Software Development (CS2500)

Lecture 18: Class Design

M. R. C. van Dongen

November 1, 2013

Contents

1 Introduction

This lecture corresponds to the start of Chapter 8 but the presentation is different. The main objectives are as follows.

- Study methods and why they are useful.
- Learn the *pass-by-value rule*. This rule is important because it describes how `Java` methods should be evaluated.
- Simulate the evaluation of methods.
- Learn more about class design by carrying out a case study for a simple "game." We shall learn that much of class design boils down to looking for nouns, verbs, predicates, and properties in the problem specification.

2 Why Methods?

Methods are arguably one of the most interesting things in computer science. In this section we shall study methods and why we should bother about them.

Methods are useful for several reasons:

- Methods are interfaces for parameterised computations. Using a simple method call we can carry out a complicated well-defined computation.
- Method calls provide reusable computations. Methods allow us to implement an algorithm once, and use the algorithm several times. This aspect is the basis for, possibly tedious, automation.
- Method calls are the building blocks of more complex computations. An algorithm that is written in terms of a short sequence of method calls is easier to understand and easier to develop.
- Method calls are the only mechanism to change `private` variables.
- Methods encapsulate computations. You write a method definition in a single location in a single class: the class encapsulates the definition.This lets you hide the implementation.

3 Pass-by-Value

Different programming languages have different mechanisms for evaluating methods. `Java` methods are evaluated using the *pass-by-value* paradigm. Before we can study pass-by-value we need to agree on some common vocabulary that allows us to explain the pass-by-value mechanism. The following section introduces the vocabulary. This is followed by the explanation of the mechanism.

3.1 Parameter Taxonomy

This section is a short introduction to *parameter* (or argument) taxonomy. Understanding the taxonomy greatly helps us understand the difference between on the one hand the parameters in a method definition and on the other the parameters of a method call.

The following explains the difference:

Formal parameter: The method's *formal parameters* are the "variables" inside the method's parameter list in the method definition. The following shows the basic form of a method definition and its formal parameters.

```Java
⟨visibility modifier⟩ ⟨static option⟩
⟨type⟩ ⟨method name⟩( ⟨type⟩1 ⟨formal parameter⟩1,
                     ...,
                     ⟨type⟩n ⟨formal parameter⟩n ) {
    ⟨body⟩
}
```

Actual parameter: A method call's *actual parameters* are the arguments inside the method call's parameter list. The following shows the basic form of an instance method call and its actual parameters. Class method calls are written in a similar way but you leave out the '⟨`reference`⟩`.`' part. In this example, it is assumed that the method does not return a value.

```
⟨reference⟩.⟨method name⟩( ⟨actual parameter⟩1,
                           ...,
                           ⟨actual parameter⟩n );
```
Java

It is important to understand that actual parameters may be literals, variables, as well as more general expressions.

```
System.out.println( "The answer is " + 42 + "." );
```
Java

Consider the following Java code.

```
public static int f( int a, int b ) {
    return a + b;
}

public static void g( int c ) {
    int a = f( 1, 2 + c );
    int d = f( 1 + 3, a );
}
```
Java

The formal parameters of the method f are a and b. The method g has only one formal parameter: it is c. There are two method calls and both are to f. Therefore there are four actual parameters and they are given by '1', '2 + c', '1 + 3', and 'a'.

3.2 The Mechanism

Having established the difference between formal and actual parameters we are now ready to study the pass-by-value evaluation mechanism. The following are the rules. To evaluate a method call with n parameters:

1. Create a fresh variable for each parameter. Each variable is used to represent the value of a formal parameter in the method call.
2. For i from 1 to n (from left to right):
   a. Evaluate the ith actual parameter.
   b. Assign the result of the ith evaluation to the ith fresh variable.
3. Use the fresh variables to represent the values of the formal parameters and carry out the statements in the method body.
4. If the method returns a result, then substitute the result for the method call.
5. If the method has parameters, remove the temporary variables.

Notice that we first carry out computations to evaluate the expressions that make up the actual parameters. When we're finished with these computations, we assign the results of these computations to the formal parameters. When doing this, we always assign a *copy* of the actual parameter's value. With this mechanism no assignment to the formal parameter inside the method can affect the value of the actual parameter. (Even it the ith actual parameter is a variable ⟨var⟩ and the ith formal parameter also has the name ⟨var⟩.)

Having said that, instance methods can "see" instance variables, so it *is* possible to change the value of an instance variable as a result of a call to an instance method. The temporary variables that are used to represent the actual parameters are stored on a first-in-last-out data structure which is called a *stack*.

- When the method is called, the values of these variables are represented as scratch values on top of the stack.
- When the method returns this scratch space is released.

The stack is also used to represent local variables in blocks.

- When the block is entered, the values of these variables are represented as scratch values on top of the stack.
- When control leaves the block, this temporary space is removed from the stack.

3.3 Examples

At this stage, you are invited to go to today's presentation for some examples that show the pass-by-value mechanism. These examples simply could not be included in the notes as it would be a waste of paper. Also the examples are best viewed in full-blown size. You may find the presentation at `http://csweb.ucc.ie/~dongen/cs2500/13-14/18/Lecture18.pdf`.

4 Playing with Toys

In this section we shall implement a simple toy "game." As part of the implementation of the game we shall learn about class design.

When you think about it, designing a `Java` for a complex application is not a science. For example, how do we choose the classes?Once we've decided on the classes, how do we choose the attributes, and how do we choose the methods? The case-study, that we shall carry out in a moment, shows that the problem specification provides many clues.

A common technique to find classes is to look for actors in the specification. This works, because the actors correspond to the objects, and each object is an instance of its class. We may implement the object in a class that is named after the actor: toy and `Toy`; writer and `Writer`; dog and `Dog`; . The actors do things (verbs): these are the methods. The actors own things, these are the attributes. Here "owning" things includes predicates. A predicate is a term designating a property or relation. For example, 'being tall', having 'a length', 'being old', having 'an age', and so on.

We shall now use this technique to design a "game" application. The objective of the game is to take and drop toys subject to certain rules. The following specification has been deliberately simplified.

- There are *hands* and *toys*;
- *Toys* are either used or free;
- Initially *toys* are free;
- *Hands* cannot take used *toys*;
- *Hands* can only take free *toys*;
- If a *toy* is taken by a *hand* it becomes used;

- A *hand* can drop its *toy*;
- Dropping a *toy* makes it free;
- Each *toy* has its own *name*; and
- Each *hand* has its own *type*: left or right.

For simplicity we shall assume that any `String` may act as a valid toy name or hand type. To find the classes in our program we need to look for the *actors* in the program. The reason why this is useful is because the actors usually correspond to the objects and each object is an instance of its class. To find the actors we look for nouns in the specification. This is called a *noun analysis*. The nouns in the previous itemised list are the italicised words. If we take the singular form of the nouns we have four nouns: 'toy', 'hand', 'name', and 'type'. The nouns 'name' and 'type' are not really *active,* i.e. they don't *do* very much. The active (singular) nouns are toy and hand. It seems reasonable to create a class for each of them. We shall call them `Toy` and `Hand`.

4.1 The `Toy` Class

Let's first look at the `Toy` class. To see which methods and attributes are required by the `Toy` class, it makes sense to look at properties of `Toy`s and for the actions of `Toy`s. The following requirements seem most relevant.

- Toys are either *used* or *free*;
- Initially toys are *free*; and
- Each toy has *its own name*.

It seems like a good idea to have attributes `used` and `name` and provide getter and setter methods for these attributes. Since each `Toy` has its own name and `Toy`s are initially free, it seems reasonable to have a `Toy` constructor that depends on a `String` which is the `Toy`'s name.

```Java
public class Toy {
    private final String name;
    private boolean used;

    public Toy( String name ) {
        this.name = name;
        used = false;
    }

    // Getter and setter methods omitted.

    @Override
    public String toString( ) {
        return "Toy[ name = " + name + " ]";
    }
}
```

For simplicity the getter and setter methods have been omitted. Notice however that the attribute `name` is a `final` attribute. This is reasonable because it is not stated that `Toy`s can change their name.

4.2 The `Hand` Class

Next let's look at what's required for the `Hand` class. The following are the requirements that relate to properties and actions of `Hand`s.

- Hands cannot *take* used toys;
- Hands can only *take* free toys;
- If a toy *is taken* by a hand it becomes used;
- A hand can *drop its toy*;
- *Dropping* a toy makes it free; and
- Each hand has *its own type*.

The italicised words are candidate properties and actions of `Hand`s. It is obvious that a hand has its own '`type`' and also has 'its `Toy`', so we might just as well introduce an attribute `type` and `toy`. It is also obvious that the constructor should depend on the `Hand`'s 'type'. The things a `Hand` can do is 'take a `Toy`', 'drop its `Toy`', and 'make its `Toy` free'. The verbs are 'take' and 'drop' and each operates on a single object: a '`Toy`'. Usually it is a good idea to turn the verbs into method names and the objects into method arguments.

Let's see what we end up with if we use this as a rough implementation of our `Hand` class.

```Java
public class Hand {
    private final String type;
    private Toy toy;

    public Hand( String type ) {
        this.type = type;
        toy = null;
    }

    public void take( Toy toy ) { ⟨to do⟩ }
    public void drop( ) { ⟨to do⟩ }
    // Getters and setters omitted.

    @Override
    public String toString( ) {
        return "Hand[ type = " + type + ", toy = " + toy + " ]";
    }
}
```

Note that we did not implement the method `drop( )` with a given `Toy` argument: `drop( Toy toy )`. This makes sense because a `Hand` always drops its current `Toy`.

We still need to complete to implement the proper rules for 'taking' and 'dropping' `Toy`s. Let's start with the method '`take( )`'.

```Java
public void take( Toy toy ) {
    if (this.toy != null) {
        // We cannot take a Toy if Hand is full.
        System.err.println( "** " + this + " is full." );
        System.err.println( "** Cannot take " + toy + "." );
    } else if (toy.getUsed( )) {
        // We cannot take a used Toy.
        System.err.println( "** " + toy + " is taken." );
        System.err.println( "** Cannot take it." );
    } else {
        // Take toy.
        // Formally mark toy as used.
        toy.setUsed( true );
        // Make toy our current Toy.
        this.toy = toy;
    }
}
```

The implementation of the method '`drop( )`' turns out equally easy.

```Java
public void drop( ) {
    if (toy == null) {
        // We cannot drop a toy if we don't have one.
        System.err.println( "** " + this + " is empty." );
        System.err.println( "** Cannot drop any toy." );
    } else {
        // Drop our current toy.
        // Formally mark toy as free.
        toy.setUsed( false );
        // Make hand empty.
        toy = null;
    }
}
```

4.3 The `main` Method

Now that we've implemented our classes let's "play" with our `Toy` and `Hand` objects. The following is the implementation of the `main`.

```Java
public static void main( String[] args ) {
    Hand left = new Hand( "left" );
    Hand right = new Hand( "right" );
    Toy game = new Toy( "computer game" );
    Toy puzzle = new Toy( "puzzle" );

    left.take( game );
    right.take( game ); // Results in error message
    right.take( puzzle );
    left.drop( );
    left.drop( );       // Results in error message
}
```

Notice that all the `String` literals in the `main` are magic constants. It is usually better to introduce a constant for these literals.

```Java
private static final String LEFT = "left";
private static final String RIGHT = "right";
private static final String GAME = "computer game";
private static final String PUZZLE = "puzzle";

public static void main( String[] args ) {
    Hand left = new Hand( LEFT );
    Hand right = new Hand( RIGHT );
    Toy game = new Toy( GAME );
    Toy puzzle = new Toy( PUZZLE );

    left.take( game );
    right.take( game ); // Results in error message
    right.take( puzzle );
    left.drop( );
    left.drop( );       // Results in error message
}
```

5 For Monday

Study the notes. Study Sections 7.1 and 7.2.