KIT101 Notes — 13 — Functional Decomposition

James Montgomery

Revised: 2017-04-06

References: L&L 4.4, 7.1, 7.7

Functional Decomposition

This is a brief guide to breaking down an algorithm into a number of methods that will each solve one part of the problem. It is not a set of fixed rules, so it is up to you to consider how each programming problem you encounter may be divided into subproblems, a skill which will improve as you gain experience.

First step: Organise the code into modules (methods)

The term 'module' is used to describe any sub-part of a larger programming solution. Most commonly it refers to an individual function, procedure or, in OO programming, *method*.

When applying **problem solving** in general, it helps to divide the problem in sub-problems. When developing a program, write a method to solve each sub-problem.

A guiding principle when writing code is that each method should **do *one* thing only**.

Advantages of this approach:

- Related code is found together
- Can solve the problem a bit at a time (can test solutions to each part separately)
- Can reuse parts of solutions to solve other problems

Next steps: Implement and test

Once you've decided what methods are required (and what their parameters and return types are), write **method 'stubs'**, which means everything for the method *except* the code inside its `{ }` block. Note that if the method is intended to return a value you may need a single statement inside the block to return some dummy default value. It is also common to put a single comment inside a stub method of the form `//TODO What's left to be done`.

Here is are two example method stubs (from two very different classes):

```
/** Moves the Turtle the given distance; will draw a line if the pen is
down. */
public void move(int dist) {
    //TODO Implement this method
}

/** Returns a string containing only letters a-z from message, all in
upper case. */
public String cleanText(String message) {
    //TODO Remove non-letters, convert to upper case, return the result
    return ""; //dummy return value until method is actually implemented
}
```

Sharing data between modules (methods)

Once a problem has been broken down into smaller parts, the methods to solve each part often need to work on the same data, so we need a way of sharing data between modules/methods.

- **Within an object**, we can use instance data. Each method within an object
  - can use (read) instance data and
  - can change the value of instance data
  - which is useful if we want to changes to persist (stay around) after each method is called
- **Within a single-file program** (main calling other `static` methods), we must use parameters and return values
  - parameters (arguments) pass *in* necessary data
  - return values pass data *out* of the method
- **Between objects**, we must use parameters and return values
  - parameters (arguments) pass *in* necessary data
  - return values pass data *out* of the method
- And also **within an object** we can also use parameters and return values
  - when doing the same task with different inputs (general purpose methods)
  - and when we the primary purpose of a method is not to change an object's state

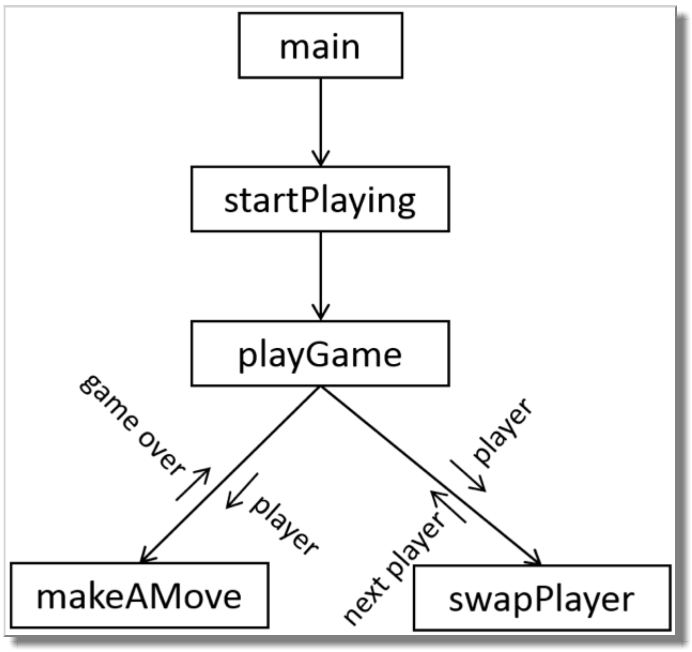
Structure Charts: A visual tool to document method breakdown

Structure charts are a useful graphical tool for describing the breakdown of a solution into smaller parts (modules). They can be used to document an existing program or to plan a new one. While they can be used at varying levels of abstraction from the final software system, we're going to suggest you use them to model the pattern of methods calls in your programs.

Structure charts contain boxes for each method linked by arrows to the method(s) it calls. Data going into a method, or a value being returned, are shown by small arrows with labels describing the data (either its role/purpose or data type, whichever you find most useful).

Here is an example for a possible implementation of a two-player game (it doesn't matter which one) with the following methods:

- main(): entry point for the program, calls startPlaying()
- startPlaying(): does initial setup then enters a loop to allow zero or more games to be played
- playGame(): conducts a single game, alternating between allowing the current player to makeAMove() and then, if the game is not over, calling swapPlayer()
- makeAMove(): allows the current player to make a move and returns **`true`** if that move has ended the game, **`false`** otherwise
- swapPlayer(): performs housekeeping tasks to switch the currently active player



When drawing a structure chart start with *main* at the top and then expand the tree of method calls down the page. Although you may choose to indicate the order in which methods are called by their relative positions left to right, it's not a requirement of the approach.

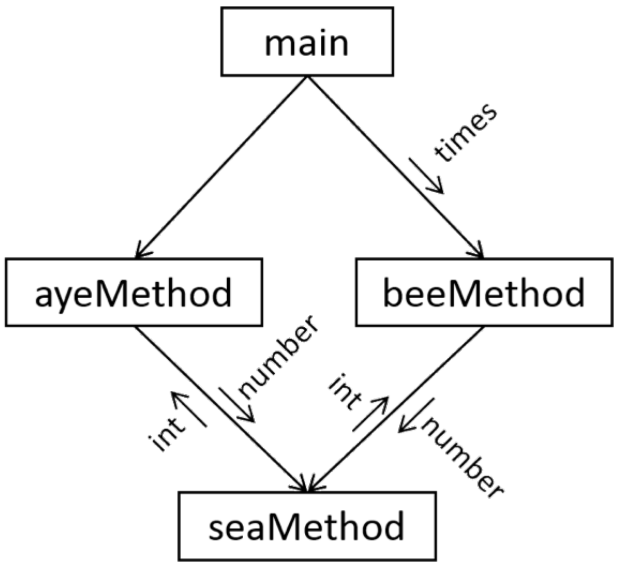
Tip: As you start to produce more complex software spread across multiple source files you can prefix the method names with the name of the class that defines them. For instance, if the above were defined in `Game.java` then we could label the boxes as Game.startPlaying, Game.playGame and so on.

Activity: Create a structure chart

Create a structure chart for the following program. It's deliberately abstract (to make it smaller, really) so is actually a little more challenging than a real program).

```
public class Demo {

    public static void ayeMethod() {
        int i = 0;

        i = seaMethod(i);
    }

    public static int seaMethod(int number) {
        return number + 2;
    }

    public static void beeMethod(int times) {
        while (times < 10) {
            times = seaMethod(times);
        }
    }

    public static void main(String[] args) {
        ayeMethod();
        beeMethod(3);
    }
}
```

There is no one ‘true’ answer to this question, given the abstract nature of the code. In this solution we’ve labelled return values by their type (since we have no other meaning to give them) and parameters by their names).



Note that it is not the order in which the methods are defined that determines their level in the structure chart, only which method calls which other method(s).

Tip: Structure charts can form part of your program documentation.