Block 2: Software Design

Kelly A

25th January 2016

1 Top-down Design (Step-wise Refinement)

Writing a program that is non-trivial (it is complex or very large) can be a daunting task. Even writing a small program for beginner programmers can be overwhelming. Where does one start? A top-down design (also known as step-wise refinement) is a disciplined model for designing programs that breaks down the program into manageable pieces. It emphasises a complete understanding of the program and can be considered as the *plan* for the program.

So how do we use top-down design to write a program? First, we identify the main subtasks or steps of a program, specifying them in plain English. We call this the *top level*. Then, we take each of these subtasks individually and again divide them into several smaller subtasks, creating a new level. We continue this process of subdividing and creating new levels until it becomes clear how each part of the problem can be implemented in code. We refine each of these steps by thinking about the details involved in each subtask. We continue to refine until we approach a point at which the algorithm for each of these steps becomes both concrete and easily translatable into a particular programming language. One important point to remember is that a top down design should not be described in any programming language, but rather in English. This is in order to keep the design separate from the programming language i.e. there should be no references to syntactic elements or library functions specific to a particular programming language.

Two mistakes are commonly made when designing programs using this method. Firstly, the design stage is terminated prematurely, i.e. not all tasks are divided until they are simple enough to implement. Even for simple programs, it is often the case that you need to go beyond level 1. Secondly, the design is written *after* the program has been implemented. This is clearly a waste of time and can lead to programs being inefficient, disorganised and poorly designed.

Let's consider an example. Write a program that draws the picture of a house as shown in Figure 2.

The first step is to work out what the specification is asking. We have to understand what is meant by "to draw a house". This is clarified as the picture of the house to be drawn is given.

The top-level (level 1) is:

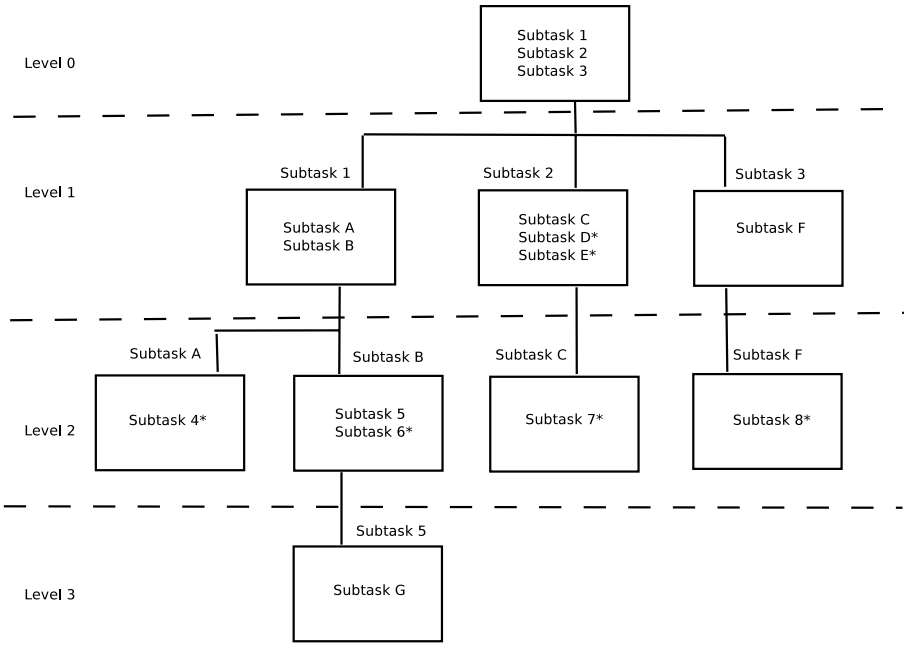

Figure 1: A pictorial view of a top-down design (structural diagram). It has a “tree-like" design, where each branch is subdivided into smaller branches at the next level, unless all the subtasks are simple enough to be implemented easily (these are indicated by a *).

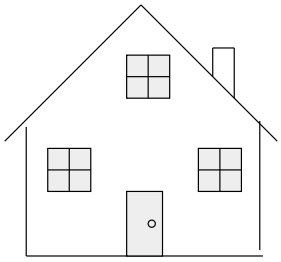

Figure 2: A picture of a house.

- Draw the outline of the house
- Draw the chimney
- Draw the door
- Draw the windows

Notice that the door has both a frame and a handle and therefore the *"Draw the door"* task can be further subdivided into two steps (level 2):

- Draw door frame
- Draw handle

The house consists of three windows, so the subtask *"Draw the windows"* can be divided into (level 2):

- Draw window 1
- Draw window 2
- Draw window 3

The three windows are identical - they only differ according to the location where they are placed. Therefore, we can reuse the code, but just ensure to provide the correct location each time. Now we just need to find out how to draw in the particular programming language of our choice, which moves us out of design phase and into the implementation phase.

Exercise 1.1 Top-down design

1: Consider the following program. Write a program for implementing a phone book. The phone book consists of names and phone numbers. A user can search of a phone number, given a name, and can also print all the names and numbers for a given a letter of the alphabet (e.g. given the letter B, all the names that start with B and their respective phone number will be printed). Produce a top-down design model for this specification.

2: Develop a top-down design for the following program. Write a program that prints the sum of all the even numbers in a given list, as well as the smallest and largest number in that list. Assume that the list only contains numbers that are smaller than 1000.

3: Develop a top-down design model for your GUI game.

2 Problem Solving

Poyla [Pol57] published a book providing techniques on how to solve mathematical problems. The following paper (https://math.berkeley.edu/ gmelvin/polya.pdf) outlines the key principles of problem solving described in this book, that can also be applied when solving problems in computer science. Given a programming task, the best place to start is to follow these key principles:

Understand the problem: what are the inputs (arguments)? What are the outputs (values returned or results)? What is the specification of the problem? Ask questions and try to understand its scope. Is there any part of the specification that is unclear, redundant or contradictory? Are there any special conditions on the input? Can you break up the problem into parts. Drawing diagrams and writing things down in pseudo-code could help. Think of sample inputs, and these can then be later used to test the code.

Make a plan: Think about the connections between the input and the output. If you cannot think of such a connection, see if you know of any related problems (e.g. a more specific one or a more general one) or any programs/functions that could be useful. Maybe you can solve part of the problem? Or get something useful from the inputs? Make some sort of plan of how to write the program.

Carry out the plan: Write the program in a particular language e.g. Racket based on your plan, checking each step of the design. You can write the program in stages, e.g. divide the problem into different cases and then determine how all the different parts will be put together to obtain the final results. Another approach would be to solve a smaller input, and then using the result to get your result i.e. recursion. You could also write a more general or specific solution and then this might give you ideas on how to solve the actual problem. Also, see if you can use (by modifying/extending) existing solutions or programs that you have written to build the solution.

Reflect on what you did: Test your program on a variety of inputs. Think how you could improve your solution or what you would do differently if you had to start again.

Exercise 2.1 *Consider the following problem: find the maximum of three integers. Get into groups of 4 or 5 and apply the principles above to make a plan on how to solve this problem in Racket. Make sure you understand the problem first, e.g. what happens if two of the integers are maximal? and when making a plan consider if there are any simpler problems that are related.*

We have considered how to sort three integers. What if we had a list of integers? How would we go about sorting the list? A sorted list would make it easier to find

specific elements and to know which elements are near each other. There is a great deal of research on the sorting problem - i.e. finding efficient sorting algorithm (reducing the amount of resources required to compute algorithm, e.g. time and space).

References

[Pol57] Polya, George (1957) *"How To Solve It"*, Princeton University Press, 2nd Edition