Hierarchical Abstraction

coursera.org/learn/interacting-system-managing-memory/supplement/XWtO0/hierarchical-abstraction

Abstraction works hierarchically—we can combine the smallest "building blocks" of our system into larger logical units. We can then combine those larger "blocks" into larger units and repeat this process until we have a large, complex system. Of course, ideally, at any granularity where we inspect the system (*e.g.*, read the code), we have something small enough to fit into our cognitive capabilities—something we can analyze and understand.

Returning to our example of letters and words, words get combined into sentences, sentences get combined into paragraphs, paragraphs get combined into sections, and so forth. You can see the same effects of thinking about a thing in logical units at any of these granularities: think about the sentence “I am learning about why abstraction is important in programming.” and the gibberish phrase: “A in tricycle grail air discontinue my flammable to imprecision.” As with the letters/words example, you can think about the first more easily, as it has a logical meaning, even though they both have the same number of words (and letters in each word).

Designing software with hierarchical abstraction can primarily happen in two ways: *bottom-up*, or *top-down*. In a bottom-up design, you start with the smallest building blocks first, and build successively larger components from them. This approach lends itself well to incremental testing (build a piece, test it, build a piece, test it...). However, the downside is that you have to be sure that you are building the *right* blocks, and that they all fit together in the end.

The other design philosophy is top-down. In top-down, you start by designing the highest-level algorithm and determine what other functions you need in support of it. You then proceed to design these functions, until you reach small enough functions that you do not need to abstract out any more pieces. This design approach should sound rather familiar, as it is exactly what we have described to you when discussing how to translate your generalized steps into code. The advantage here is that you know exactly what pieces you need at every step (they are required by the higher-level algorithms that you have already designed), and how they fit together.

The downside to top-down design can arise in testing. If you try to write the whole thing, then test it, you are asking for trouble. However, if you implement your algorithm in a top-down fashion, you may have high-level algorithms that rely on lower-level pieces that do not exist. This problem can be overcome in a couple of ways.

First, you can still test what you have every time you build a complete piece. That is, when you finish a "small block," you can test it. Then once you have built (and tested) all the "small blocks" for a medium sized piece, you can test it. Effectively, you are building and testing your code in a bottom-up fashion, even though you have done the design in a top-down fashion (and may have some partially implemented/untested higher-level algorithms).

The second way you can address this problem is to write *test stubs*—simple implementations of the smaller pieces which do not actually work in general, but exhibit the right behaviors for the test cases you will use to test your higher-level algorithms. Such test stubs may be as simple as hard coded responses (*e.g.,* **if(input==3) {return 42;}**). You can then test your higher-level functions assuming the lower-level pieces work correctly, before proceeding to implement those pieces.

✓

Completed