Module 0049: Top-down design

Tak Auyeung, Ph.D.

September 26, 2006

1 About this module

- Prerequisites: 0012, 0013, 0016
- Objectives: This module introduces the concept of top-down design.

2 Top down design in real life

2.1 What it is

We do top-down design all the time in real life. For example, when we write a paper in an English writing class, we first identify the major sections of the paper (like introduction, background, and summary). Then, we write the content for each section. Sometimes, we even need to outline each section with subsections first.

Maps are also perfect examples of top-down design. If you visit any mapping website, a map shows different types of details depending on the "zoom" level. A map for the globe, for example, is unlikely to identify streets. Instead, it probably identifies nations and national capitals. A map for a country, on the other hand, may include states (provinces), major cities and state capitals. We may also see major highways, but probably not intrastate roads. A regional map will identify smaller towns and cities, as well as intrastate roads. Even at this level, local streets cannot be displayed.

Most books are also organized in parts, and then chapters within a part. This is especially the case for college-level textbooks.

Documents are usually organized in an office or even at home. A file cabinet contains drawers, a drawer contains hanging folders, a hanging folder contains regular folders, and a regular folder contains stapled documents.

Essentially, the term "top-down" means a hierarchical structure of details. At the top level, very little detail is revealed. At the bottom level, very fine detail is displayed. Between the top and the bottom, however, are levels that display intermediate detail.

Can you find more real-life examples of top-down hierarchies? In the computer context, how files and folders are organized is also a prime example of a top-down hierarchy.

2.2 Rationale

Why do we use top-down design? The answer is quite simple: we cannot manage too many things at the same time.

Imagine you are managing photographs. Due to the "no-cost" nature of digital cameras, many people shoot thousands of pictures per year. If all pictures are thrown into the same folder for a few years, the folder will end up with so many pictures that it is impossible to find a particular one.

That's why we all have different ways to organize digital pictures.

3 Top down design for algorithms

3.1 Algorithm specific considerations

There are some special considerations that are specific to algorithms. One primary concern is how much can we fit on paper or the computer screen without flipping pages.

Even with correct indentation to help us identify the beginning and ending of blocks, if a construct spans across physical pages, it will still be difficult to match the beginning with the end of a construct. Based on this limitation,

an algorithm should have no more than 40 lines or so of code. The 40-line limit is based on what most people feel comfortable with respect to font size and display size.

A 40-line algorithm can be quite complex for beginning programmers or even intermediate programmers. In reality, an algorithm should not have more than 20 lines of code if it is intended for beginning programmers.

3.2 But how do we fit...

Not all algorithms can be specified in 20 lines or less. As a result, we need to "abstract" out chunks of code with "abstract operations". An "abstract operation" simply means "we don't know/express how to do this exactly". An abstract operation is one that is not composed of the basic types of statements (assignment, sequence, conditional statement or loops). This way, we can make algorithms *appear* shorter by hiding details.

3.3 An example: factoring

Let us consider an example to find prime factors of an integer. Let us assume that $n > 1$ is an integer that we want to factor.

3.3.1 Do it by hand, first

Let's factor a number by hand first, just so that we understand the process. Let's say we want to find all the prime factors of 20.

First, we see that $20 = 2 \times 10$, and 2 is a prime number.

Then, we see that $10 = 2 \times 5$, and 2 is a prime number.

Lastly, we see that $5 = 5 \times 1$, and 5 is a prime number.

At this point, the number is reduced to 1, which means there is no more factor.

3.3.2 Assumptions

Let us be "lazy" and make some assumptions. For example, I'll assume, for now, that there is a way to find a prime factor f of n. Given this ability, I can write the logic as in algorithm 1.

```
1: given n > 1 is an integer to be factored
2: repeat
3:     find a prime factor f of n
4:     print f on the screen
5:     n ← n/f
6: until n = 1
```

Algorithm 1: A factoring algorithm.

As an exercise, assume $n = 15$ and trace this algorithm.

3.3.3 Bad assumption!

As it turns out, most programming languages do not have a method to find a prime factor f of a given integer n. Oh, well. That means we need to specify an algorithm just to do this. Again, it is best to try to do this by hand first.

Let us assume $n = 15$, and we are to find a prime factor of it.

Although not necessarily the most efficient method, one way is to try out all integer greater than or equal to 2 until we find one that is a factor. This requires the explanation of the "mod" operator. $a \mod b$ is the remainder of $\frac{a}{b}$. In other words, $10 \mod 3 = 1$, $23 \mod 10 = 3$ and etc.

Given this operator, we can now explain how we can find a prime factor of 15:

- $15 \mod 2 = 1$, therefore 2 is not a factor of 15
- $15 \mod 3 = 0$, therefore 3 is a factor of 15!

3.3.4 Finding a factor

Now we are ready to express the logic of finding a factor in algorithm 2.

```
1: f ← 2
2: while (n mod f ≠ 0) do
3:     f ← f + 1
4: end while
```

Algorithm 2: An algorithm to find a prime factor f of an integer $n > 1$.

3.3.5 To integrate or not to integrate

Now that we have separate pieces of the algorithm, we can choose whether to leave algorithms 1 and 2 just the way they are. If you want to see the entire algorithm, we can integrate the two. The integration is nothing more than filling in the statement "find a prime factor f of n" in algorithm 1 with the entire algorithm 2. The result of integration is algorithm 3.

```
1: given n > 1 is an integer to be factored
2: repeat
3:     f ← 2
4:     while (n mod f ≠ 0) do
5:         f ← f + 1
6:     end while
7:     print f on the screen
8:     n ← n/f
9: until n = 1
```

Algorithm 3: The integrated algorithm to factor a number.

The integrated version shows all the details of how to factor a number. However, it is more difficult to read because of the distracting detail of finding a factor.

3.3.6 Where is the top-down design?

The top-down design starts with my assumption that "someone else has already figured out how to find a factor f of an integer $n > 1$". This "laziness" actually works in my favor because it permits me to focus on the main logic of factoring a number. In other words, the "top" part of top-down is algorithm 1.

Once I traced algorithm 1, and am convinced that it works, I shift my attention to the assumed operation. Only at this time did I consider the logic of finding a prime factor f of an integer $n > 1$. Note that when I tried to figure out the logic of finding a prime factor of an integer, I don't need to worry about the logic of factoring a number anymore.

4 Top down isn't everything!

Although top-down design is taught in every algorithm design class and throughout all computer courses, it is *not* the only way to write programs. It is nearly impossible to apply the top-down method perfectly in the real world.

In other words, it is nearly impossible to use the correct abstract operations to make the top level algorithm "just right" when you are writing an algorithm (or a program). In real life, programmers rely on a few techniques to keep program code manageable.

4.1 Top-down

Whenever it is possible, convenient and make sense, use the top-down method. "Think in big steps" is one way to think about it.

4.2 Bottom-up

Seldom mentioned is the technique called bottom-up. This technique is very useful for more experience programmers. This method is the inverse of top-down design, because small building blocks are first created, then larger blocks are created from smaller blocks. The process is repeated until a program accomplishes what it needs to do.

The reason why the bottom-up technique is used only by more experienced programmers is because beginners cannot guess what components are needed first. More experienced programmers can look at the overall objectives of a program, and quickly determine a set of operations that must be implemented and utilized.

4.3 Abstraction

More often than not, even with experienced programmers, an algorithm grows and grows until it becomes "messy" and too lengthy. At this point, we can "abstract" portions of the algorithm as an abstract operation, and thus hide the details in the definition of the abstract operation.

The key to abstraction is to identify a block of code (within an algorithm) that is somewhat self contained and related. This process, once again, requires some experience. There are methods to evaluate how self contained is a block of code based on the usage of variables. We will revisit this topic in the module that discusses subroutines.

As an example, sometimes we come up with algorithm 3 first, then decide that it is too much to read. *Then*, we see a self contained block of code as algorithm 2, and extract that out of the main algorithm. Once we extract the self contained block of code, we replace it with a one sentence description "find a prime factor f of $n > 1$". We also end up with a second algorithm as algorithm 2. The main algorithm, then, becomes algorithm 1, which is considerably easier to read than algorithm 3.