Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Start with the Most Difficult Part

Diomidis Spinellis

There's not a lot you can change in the process of constructing a building. You must lay the foundation before you erect the upper floors, and you can't paint without having the walls in place. In software, we're blessed with more freedom.

I recently experienced this when I implemented *wpl*, a small system that extends arbitrary Web pages with links to Wikipedia entries. (Try it at www.spinellis.gr/wpl.) The system has many parts: tools that convert the Wikipedia index into a longest-prefix search data structure; an HTML parser; code that adds links to phrases matching Wikipedia entries; and a Web front end that fetches the page, adds the links, and returns it back. As I was adding the finishing touches, I reflected on the process I used to construct the system. (I find a postmortem examination deeply satisfying, but this is probably because I'm not a medical doctor.)

What struck me were the different approaches I used to construct each of the system's main parts. For the search data structure, I worked bottom-up: I first read about and experimented with a couple of alternatives, learning about Bloom filters, tries, and Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric, and I'm not making this up) trees. Next, I designed the data layout and the low-level bit-twiddling code to add and locate entries. Only then did I write the tools for building the index and an API for searching entries in the data structure.

For the HTML parser and word-linking code, I started somewhere in the middle. I wrote a state-transition engine to parse "tag soup" HTML (HTML that isn't necessarily well formed), and then I extended it with code to add links to suitable text, and I added an appropriate interface.

As you'll probably expect, for the Web front end, I followed yet another approach, working in a top-down fashion. I first wrote down the server's main tasks and their precise descriptions; once these were in place, I started working on their implementation.

As I reflected on my construction process, I wondered what made me choose three different approaches within a week: bottom-up, middle-out, and top-down. Are my programming work habits really so chaotic as they appear? If so, what am I doing writing columns in a magazine for software professionals?

The Hidden Logic

Fortunately, I quickly realized there was logic behind my choices. On each subsystem, I started on the most difficult part; the one with the largest number of known unknowns. For term searching, this was the data structure and the associated code. I needed a structure where I could quickly search (thousands of times for each processed page) for a page's words and phrases matching one of the millions of Wikipedia entries. For this, the structure should be compact enough to fit in memory, yet flexible to allow for longest-prefix matching—a tall order. Once I had solved this low-level problem (by using a space-optimized version of a Patricia tree), designing an API and building the indexing tools were easy.

Similarly, the most difficult problem in the page-processing part was the HTML parsing. On the basis of Postel's law, "be conservative in what you do; be liberal in what you accept from others," code that processes Web pages should be able to parse

 0740-7459/09/$25.00 © 2009 IEEE

anything that remotely looks like HTML. In my system I had to do this efficiently for the very narrow task of recognizing word sequences that could be linked to Wikipedia. In this case, the high-level interface and the lower-level word search and replacement code were considerably easier, so I ended up starting in the middle.

Finally, when the time came to implement the Web front end, I initially experienced writer's block, not knowing where to start. I realized I had no clear understanding of how the various pieces would fit together: how to obtain the Web request's parameters (to which language's Wikipedia to link, and what character encoding to use), when to retrieve the page, what to do with the HTTP header I received, and how to construct the response's header. I escaped this impasse only when I designed the functionality in a top-down fashion, a choreography that I expressed in just nine lines of high-level C++ code. Once I had those lines in front of me, I could easily see what each part would need to do. When I was ready to code the low-level functionality, the actual mechanics of setting up socket communications and talking HTTP were (almost) boring details.

... And Its Advantages ...

Starting with the most difficult part has three main advantages. The most important concerns the management of design constraints. On a blank sheet of paper, the constraints we face are minimal, but each design decision imposes new restrictions. By starting with the most difficult task, we ensure that we'll face the fewest possible constraints and therefore have the maximum freedom to tackle it. When we then work on the easier parts, the existing constraints are less restraining and can even give us helpful guidance.

For instance, when I designed the search data structure in a bottom-up fashion, I realized that choosing the UTF-8 variable-length character encoding had significant advantages in terms of space and time efficiency. Had I started my work top-down by specifying an API that mandated the use of wide characters, I might have needlessly constrained my choice of the data structure. In contrast, when designing the Web front end, having various ill-fitting pieces of functionality would make it more difficult to assemble them into an efficient whole. So in that case, top-down design was appropriate.

The second advantage is the early shrinking of the project's cone of uncertainty. Any development project involves elements that we don't know at the beginning and discover as we progress. By putting the most difficult (and consequently risky) part quickly behind us, we rapidly minimize the project's unknowns and can therefore make more informed and intelligent decisions regarding budget, progress, staffing, and functionality.

The final advantage has to do with human nature. At a project's beginning, we have the highest levels of enthusiasm and motivation; at later stages we can expect some disillusionment and even burnout. By starting with the most difficult part, we ensure that we undertake it with a positive can-do attitude rather than a defeatist spirit. We can even apply this principle on a smaller scale by scheduling difficult work early in the day when the mind is clear and distractions are minimal. Plan boring meetings where they belong: at the end of the day.

... Further Applied

You can also apply the principle I've described when ordering elements of the software life cycle: requirements elicitation, high- and low-level design, coding, debugging, testing, and maintenance. Yes, if you're venturing into an unknown application territory, start by gathering requirements. Change your plan, however, if you know approximately what you need to do, you have a good communication channel with your users, and the most difficult part of the project is, for instance, how you'll display a complex data set. In this case, your problem is not the project's requirements but its rendering functionality. Therefore, spend some time experimenting with what you can achieve with, say, OpenGL or Ajax. Then, when you're discussing requirements with your users, you can steer them toward the technically achievable directions rather than letting them specify functionality that will be costly to implement.

Along the same lines, if you think that testing will be more difficult than low-level coding (it frequently is in many application domains), start by writing your test cases. If you believe that maintaining your system will involve a lot of effort, plan ahead by designing a domain-specific language that will simplify the most common maintenance tasks. If your design involves a large API, sketch its conventions before specifying its actual elements. If you think your code will need a lot of debugging, liberally sprinkle assertions, logging, and hooks.

The list goes on. Whatever you do, just remember: start with the most difficult part.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business. He recently coedited *Beautiful Architecture* (O'Reilly, 2009). Contact him at dds@aueb.gr.

Post your comments online by visiting the column's blog:
www.spinellis.gr/tools

Any development project involves elements that we don't know at the beginning and discover as we progress.