CUTTER CONSORTIUM Access to the Experts

Business Technology Trends & Impacts Advisory Service

EASY AS IMPLEMENTING A PACKAGE, PART 2

by Michael Mah, Senior Consultant, Cutter Consortium

In Part 1 of this article series (see "Easy As Implementing a Package ... Part 1," 1 March 2007), I described the productivity characteristics of large IT package implementations, including enterprise resource planning (ERP) applications. I took on this subject in response to an article by my fellow Cutter colleague Steve Andriole (see "Sourcing Today and Tomorrow," 15 February 2007), who said that many CIOs and CTOs are often extremely frustrated by cost and schedule overruns in projects like this. In worst-case scenarios, some have even lost their jobs.

There are several reasons why companies struggle greatly with project estimation when it comes to implementing packages, both large and small. Here are a few that are frequently cited:

- Implementing a software package means you're buying code (off-the-shelf) that someone else wrote and then attempting to make it work in your organization. Much of this work is about configuring and customizing the application, not writing your own code.
- More of the work can be about trying to figure out how the package works, what to use, how to then configure those parts (setting up rules engines and tables, for example), and deciding what parts NOT to use. There's a lot of thinking before any actual work gets done.
- Since you've typically had to buy the whole enchilada, it may take work to stub out parts that you decided not to use in order to set them aside.
- A centralized database is an essential aspect of this work; database conversions and migrations are frequently involved, as well as setting up tables and creating reports; little code work might be involved.
- Some organizations can't use a package strictly "as is." Rather than change how their organization works to suit the software, they feel they have to customize the software instead. That involves code work.
- Customizing software that someone else wrote is hard.
- After customization, there's still the task of retiring old applications, connecting the new system to legacy applications that are kept, and passing data across new interfaces that have to be written. Some are simple; some are complex ...
- ... and that very frequently involves -- guess what? -- writing code.

No wonder people's eyes glaze over at the prospect of implementing a package. How do you deal with something that involves code and, at the same time, doesn't involve code?

In the old days, frustrated by project overruns on new applications built from scratch, many believed that simply implementing COTS software would be the ticket out of project overrun hell. Just buy functionality, was the mantra. However, as packages got more complex and spanned more functional domains, even these projects became more difficult than anyone imagined.

But wait! Now that there's more and more industry data on projects like these, large and small, we're discovering how they are similar yet different than other software development projects. What does that mean to us? In a nutshell, it allows us to better estimate, plan, and manage this kind of work, upon which millions (or sometimes hundreds of millions) of dollars is often at stake. We've found that the productivity pattern of this work is often very similar to other complex IT undertakings, but SIZING these projects is what's different.

If you solve the sizing puzzle, it's possible to combine that knowledge with well-established productivity assumptions to predict more reliably the time and the work effort (person-months, hours, or days) that these projects could entail. The trick is sizing the work when some of it involves code (we're still talking about SOFTware, folks), while some of it doesn't, and then combining the two.

My friend and fellow Cutter colleague Ed Yourdon once said, "If you underestimate the size of your project, it doesn't matter which methodology you use, what tools you buy, or even what programmers you assign to a job." In other words, if you think a project is the equivalent of a little league ballpark and it turns out to be more like Yankee Stadium, your goose is cooked. Many package implementation projects start out looking small and then turn out huge, mostly because teams fail to size them well.

So here are a few examples of how others have sized package implementation projects successfully:

- You can count the number of high-level and then detailed business processes that are being automated. These are sometimes called "configuration items." When exploring how a package might suit these business processes, there are often cases where the functionality "fits," and in other cases, there are "gaps." You can also count these.
- Producing the desired functionality often involves creating items such as reports, tables, interfaces, database conversions, enhancements, and input forms. You can count these.
- Creating components such as interfaces often involves writing software. Simple interfaces might involve fewer (100 or so) lines of code (LOC), while complex interfaces often require more (1,000+).
- Components that don't involve code often require "programming actions" of some sort. While they're not about typing an instruction in a given programming language like C++, they are about producing an instruction nonetheless, like mouse clicks/drags for setting up tables, business rules, or invoking macros. Simple components require fewer of these "programming actions" or LOC equivalents, while complex components require more.
- There are often 10, 20, or so LOC equivalents on the smaller end of the scale, and maybe about 100 or so on the larger end. Since it's not really code but instructions, some teams count them as "logical instructions," "programming actions," or "implementation units."

Now comes the estimating part. Tallying up the size of all the work products can be done on a spreadsheet! You estimate the number of reports, interfaces, database conversions, enhancements, forms, and tables: 20 of these, 45 of those, and so on. Each of these components has a "currency conversion," such as the amount of code per interface or the number of programming actions/implementation units (LOC equivalents) for tables, reports, and the like. Then, adding it all up is simple, and when you look at the sum total, that's something that developers can handle. You've successfully estimated the size of your package implementation, in LOC equivalents.

Combining that with targeted productivity assumptions (low to high) based on how difficult the project might be can enable you to run a Monte Carlo simulation of how long the project should take and how much it might cost. If you have a reasonable handle on both the productivity and sizing assumptions, as well as the expected range, you can bet that you have a solid basis for a reasonable project estimate. As a result, there's far less risk of a disastrous overrun and/or slippage. That means CIOs can get to keep their high-paying jobs. If you successfully protect your CIO, that often is a good thing.

I welcome your comments on this *Advisor* and encourage you to send your insights on current business technology trends to me at mmah@cutter.com.

-- Michael Mah, Senior Consultant, Cutter Consortium

Cutter Consortium, 37 Broadway, Suite 1, Arlington, MA 02474, USA. Phone: +1 781 648 8700, Fax: 781 648 1950