How Big Is That System?

Whil Hentzen

Most Fox developers these days are running into situation of being asked to take over another system. One of the first things you'll need to do is evaluate that system. This article describes a tool that gathers a variety of statistics about a system and then discusses what to do with that information.

One of the first questions you should ask is 'How big is the system'? Understanding the size of the system is the first step in appreciating the complexity. Unfortunately, most system owners have no idea, and I mean *literally, NO IDEA*. After all, the system was developed over years, in more than a few cases, decades. And often by more than one developer. It's a sure bet that no metrics on the development have been kept, so the number of hours put into the system is worse than a mere guess. However, you can get a rough size and general scope of the system by determining how many files and lines of code are in it, as well as a few metrics on what's in the code. These metrics may give you an insight on the difficulty in working with the code.

What comprises the system?

The first issue to consider in this endeavor is defining what comprises "the system". At first glance, one might assume that simply scanning through all of the files in a VFP PJX would be sufficient, but au contraire.

* There might be files in the system folder that aren't included in the project, such as files named by macro expansion, indirect references and in meta-data and dictionary tables.

* Or it might be known ahead of time that the system's files aren't contained solely in a project, but are found in the folder that contains the project.

* Or there may be system-related files scattered throughout a folder and subfolders.

* Finally, there may be multiple projects involved in the system, so examination of an entire folder might be in order.

Additionally, once the issue of scope has been determined, there may still be files that are part of the system that aren't caught by this tool. .APP and .EXE files, for example, need to be located and analyzed separately.

There isn't much that's all that interesting about identifying the scope past determining what it is, so for the time being, let's bypass that step and just assume we're going to look at a project.

Once the scope of the system has been defined, the goal is to gain an understanding of the size of the system, and 'close enough' will be sufficient. Precision is not important, so the calculations don't have to be exact. More important are the types of information gathered.

For example, whether there are 71,502 or 75,102 lines of code in the PRGs that make up a system isn't important. Whether those 70K lines of code are in 10 PRGs or 1100 PRGs is much more interesting and telling of what you might find when you get into the system.

Similarly, whether there are 71,502 or 75,102 lines of code in a class library is unimportant. Whether there are 3 class libraries that support 150 forms, or 25 class libraries and 3 forms is much more important, because those two cases describe two completely different approaches toward the application architecture.

Using the How Big Tool

Included in this issue's Downloads is a HowBig.scx form (and supporting base classes.) Running the form displays the empty form shown in Figure 1.

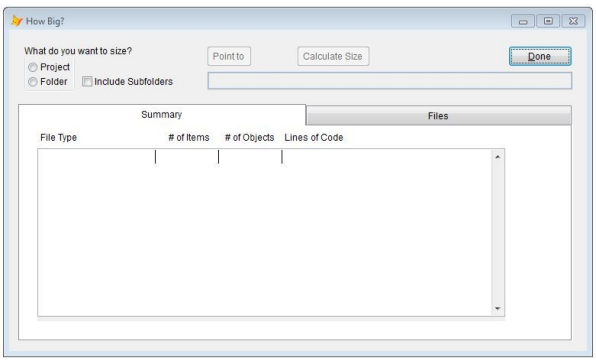

Figure 1. The HowBig system size and scope tool.

To use it, click the Project option button, click the Point to command button to select a PJX file,

and then click the Calculate Size button to generate two sets of statistics on the project.

The first set of stats is a summary, grouped by file type, shown in Figure 2.

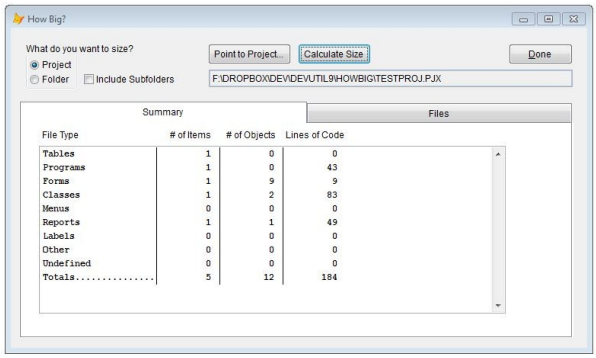

Figure 2. Summary of system file counts.

The second set is a list of files in the project, together with counts of specific attributes for each file, shown in Figure 3.

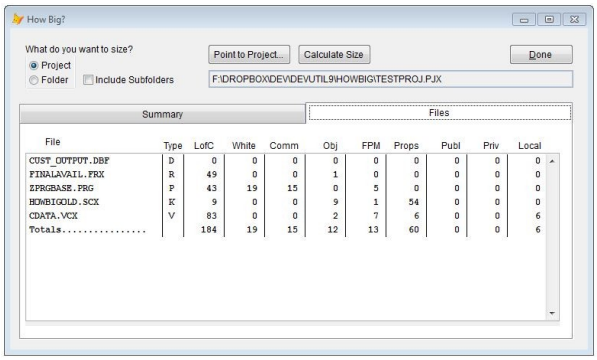

Figure 3. List of project files and file attributes.

Both the Summary and Files lists include grand total rows at the bottom.

This article won't go into the code underneath; it's simple and straightforward, with enough comments to obviate the need for lots of head-scratching. However, it's worth explaining the reasoning for the types of data that is collection.

Types of Data Collected

First of all, we want to know how many 'things' are in the system. On the macro level, 'things' are simply types of files - tables, forms, reports, and so on.

Summary tab

These are broken down and their counts listed in the Summary tab. Beyond the sheer number of files, though, we want to know how complex each object is, so I counted the number of objects in each type of file, as appropriate.

For example, for forms and reports, I drilled through how many controls were on the form or report, and how many lines of code were in its methods.

At first blush, a form with 30 or 40 controls would have seemed to be more complex than a form with one or two controls, but that's not necessarily so. A form with 2 controls but thousands of lines of code in its methods is likely going to be more complex than a form with 40 controls but only 100 lines of code.

Obviously, this doesn't take into account inheritance – a form could inherit from a complex class but appear to be simple, since it had only a couple of its own objects. Still, if the instance itself doesn't have a lot of extra objects or lines of code, it'll likely be simpler than if it had a lot of 'things' in it.

The summary page provides the totals for each type of file, so that at a glance, you can see how many files and how 'busy' those files, on average, are going to be.

Files tab

Once done with the summary, we're going to want to be able to drill down more granularly. For example, suppose there are 10 forms with a total of 200 objects and 4000 lines of code. That's 20 objects and 400 lines of code per form. That sounds like a reasonable level of complexity for a form. However, suppose your customer has told you that there are two particular forms that they need work on. Use the Files tab to look at the counts for those two forms. If they both feature over 60 objects and a thousand lines of code each, you know that these two are way more complex than the rest of the system.

The Files tab lists counts nine attributes for each file. These attributes can be segmented in three groups.

The first three attributes, Lines of Code, Lines of White Space, and Number of Comments, all have to do with how dense the code is and how hard it might be to understand. While not a cure all, code with white space is generally going to be easier to read than code that's jammed all together, and code with comments may give you a better chance than uncommented. Indeed, I once ran into a system where each PRG had no comments or whitespace, and, even worse, didn't even have indentation, because the programmer didn't want to waste the characters needed to tab or space logic structures. REALLY easy to work with.

The next three, Number of Objects, Number of Functions/Procedures/Methods, and Number of Properties, give you an idea of how many 'things' there are to deal with. A PRG with hundreds of functions or procedures is likely a

library file, but if it's not, it could be trouble. As far as properties, absence or minimal use of these, particularly in conjunction with a large number of methods in a form or class, or a large number of publicly declared variables, could spell trouble.

The last group, the Number of Variables explicitly declared as Public, Private or Local, particularly in conjunction with the number of lines of code, tells you about the discipline of the developers of the program. If there are a large number of lines of code but few or no explicit declarations, there could, again, be trouble.

What To Do With The Data

While it may seem from the discussion so far that the purpose of this data gathering is strictly personal, that is, so that you can get your arms around the system, there's a second purpose as well – to show and educate your customer about what will be required with their system. All too often, the owner of the system has forgotten the amount of knowledge hidden in its bowels, and thus assumes trivial amounts of effort will be needed to perform what they perceive as simple tasks.

There aren't any absolute or generally accepted principles when it comes to values of any of these counts. You can't take a form and absolutely declare, "Oh, this form has 20 controls on it but only one property, therefore (some conclusion)." What you can do is appreciate how much 'stuff' you'll have to wade through when working with an entity. That report that has 75 controls and hundreds and hundreds of lines of code, and not a single comment anywhere, is likely going to be challenging.

What do you do with these numbers? First, you should run this tool against all of your past projects, to get an idea of the values, both average as well as the min and max extremes, of projects you're familiar with. If a new application varies greatly from those values, you will know better what you're getting into.

Second, at some point, your customer is going to ask you for an evaluation of their system. If you have to shrug your shoulders and agree that you have no idea what you're getting into either, they're going to wonder if they want to hire you on. However, if you can quickly tell them things about their system that they don't even know, you'll appear more authoritative and professional. And if you can then follow up with, "In my experience..." and compare and contrast with other systems you've analyzed, you'll be in a much stronger position to provide and sell your recommendations.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com