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

APIs, Libraries, and Code

Diomidis Spinellis

LET'S SAY YOU want to display a JPEG-compressed image, calculate Pearson's correlation coefficient, parse an XML file, or create a key-value store. You can often choose between using the functionality of the application's platform (Java EE or .NET), calling one of several available external libraries, or writing the code on your own. It isn't an easy choice because you have many factors to consider. Specifically, you must take into account the task's complexity, as well as the licensing, quality, and support of competing options. You can narrow down your choice by eliminating alternatives at the earliest possible decision point. Here's how I recommend you go about it.

You're getting paid to deliver end results, not to reinvent the wheel.

Where to Start?

There are clear advantages in writing your own code: you control its quality, your code plays well with the rest of your system (and you can even reuse other parts of it), you don't introduce new dependencies, and you don't need to make special arrangements to support the code. The main deciding factor here is the task's complexity. You're getting paid to deliver end results, not to reinvent the wheel. Unless the task is trivial to implement, professionalism dictates looking at existing solutions. Handcrafting code to find the biggest number in a sequence is okay if a corresponding function isn't directly available in your environment. On the other hand, unless you work for a game studio or Pixar, building a 3D rendering engine from scratch is definitely a no-go area.

If some alternatives come as external libraries, start by looking at the licensing terms. Many offerings might be available as open source software. See if the distribution license is compatible with your business model. Tightly integrating GPL-licensed code with a product you'll distribute without its source code will get you into trouble. Less worrisome scenarios include using open source code that comes with a more permissive license, such as the BSD, MIT, and Apache ones, dynamically linking against an LGPL-licensed library, releasing your product's source code with a compatible license, or offering services over the Web rather than as a product. When examining proprietary packages, look at the one-off cost and per-user royalties. Only a library offering strategic functionality might justify the administrative burden and cost of paying royalties for each of your product's end users.

Next, judge the usability of the library or the platform API. Is the interface straightforward or complex and full of hidden gotchas ready to bite you? Some libraries require you to obtain an object from a factory for even the simplest operation and then initialize it with various obligatory parameters (including other objects that you need to create) before performing multiple actions, each of which can fail in several obscure ways. I've even encountered cases where the library will mysteriously fail unless I guess and setup its correct configuration. I recently saw a CSV library that would process a large file incorrectly, unless a buffer size was increased. Such behavior is pure evil. Libraries should be configured with sensible defaults that allow them to work faultlessly under all conditions (convention over configuration). Additional settings could improve their performance or specialize their operation, but they should never be required.

See www.computer.org/software-multimedia for multimedia content related to this article.

Compatibility Rules

A library's compatibility with your system will ultimately affect its usability in your hands. Are naming conventions, error handling, resource management, thread safety, and build requirements playing well with the rest of your code? Platform APIs have an edge here over external libraries because, by definition, you're already following their style. Judge the effect of each facet I listed. You can fix minor deviations by developing a small shim that adapts the library's conventions to those of your project.

Elements with diverse naming conventions coexisting within the same source code are not only an eyesore but an open invitation for more disorder and worse style abuses. If you name your methods likeThis, and a library names them LikeThat or like_that, your code won't look pretty. Sadly, in C++ and Python, various naming conventions coexist even in the standard library, at odds with recommended ones. Contrast this with the Java and .NET libraries, where names are given with Prussian-style discipline.

Error handling is another area of disagreements. A function can signal an error by returning a negative value (as do most Unix system calls), a false value (0, as is the case with the Windows API), or by triggering an exception (the standard way in the Java and .NET platforms). Mixing these idioms is like storing pesticide in a food container bottle: sooner or later, someone's going to get hurt.

Memory management can also be a potent source of problems. If each component in your project manages its own memory and the new library requires you to manually deallocate all the objects it returns (or the opposite), you're inviting trouble. At some point, an object will be freed twice or a memory leak will occur. Thankfully, modern languages with automatic garbage collection avoid this problem (by pretending it doesn't exist). There are even application domains where dynamic memory allocation is prohibited; obviously, these severely limit your selection of third-party libraries.

If your system is multithreaded, you must examine whether the library is compatible with such a setting. Sometimes this is evident from the library's interface. Routines that return results from a single statically allocated area spell trouble. In other cases, similar restrictions might be hidden behind the library's interface. It's safest to assume a library isn't thread-safe unless its documentation expressly discusses this aspect.

A final element of compatibility involves the way the library is built or distributed. Maven versus Make versus Ant versus Visual Studio versus an Eclipse project—there are unfortunately many incompatible ways in which a library's source code can be built. If the library is distributed in binary form, you have to consider whether the format is compatible with your processor's architecture or virtual machine, your operating system and package manager, the object file format, and the compiler's calling conventions. If you're building from source, invest a little time to bring the library's build under your own control. While you're at it, verify that the library is available and works correctly under all current and planned platforms where your software will run.

Final Steps

Associated with compatibility are the library's dependencies, the extended family you're marrying into. Here, you need to balance the functionality the library provides against its dependencies. For instance, dragging into your project a 200-Mbyte GUI framework or a dependency on a specific vendor's operating system just to read the screen's dimensions is probably excessive.

Having determined the library's suitability, you can now invest time to determine its quality. Are all the functions, interfaces, and error conditions well documented? (A surprising number of production systems fail that last test.) Judge correctness and robustness by using as a proxy the test cases that come with the library. A lack of test cases should raise an alarm. Run your own tests of the library, based on your particular use patterns. Look at the credentials and reputation of the library's developer. Determine who else is using the library. A large number of users form a pressure group that can aid the resolution of future problems.

Finish your assessment by looking at the offering's support. When did the last release come out? Is there a public Q&A forum? Are the library's developers participating in it? How quickly are issues resolved?

Ultimately, the choice among using libraries, writing your own code, and using an API is a matter of judgment. Thankfully, you can base your decision on the simple objective measures I've just described.

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of the books *Code Reading* and *Code Quality: The Open Source Perspective* (Addison-Wesley, 2003, 2006). Contact him at dds@aueb.gr.

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