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

Portability: Goodies vs. the Hair Shirt

Diomidis Spinellis

I don't know what the language of the year 2000 will look like, but I know it will be called Fortran. —Tony Hoare

WRITING CODE THAT can run on any platform used to be a gold standard, as attested by the tens of books with the word "portable" in their title. But staying true to the faith of portable software is becoming more challenging as mighty ecosystems amass resources to tempt us into their platform-specific version of heaven. We can write nonportable code out of laziness or ignorance because we can't be bothered to verify or check that our code follows a standard. We can also decide to write nonportable code following a pragmatic cost-benefit analysis. Let's follow this approach and examine portability as a tool, looking at what we gain through it, the price we pay for it, and how we can cope with the challenge of staying faithful to it.

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

The Goodies

The key reason to favor portable code is that it opens up the selection of resources available to a project. An ideal portable project can be compiled using diverse compilers and libraries, store its data on an arbitrary relational database, and be hosted by a variety of application servers and operating systems, which in turn run on several CPU architectures. These choices free us from vendor lock-in, allowing us to select the best technology in each area based on quality and price. In addition, as our project and business evolve, we can move from one technology to another to keep the infrastructure we use in sync with our needs. Thus, the first iteration of our project can be based on widely available commercial off-the-shelf components, but when its market share grows, we can choose to lower its unit costs by running it on an embedded platform that's based on a specialized processor and an open source operating system. Or, conversely, we can initially store our data in the open source MySQL database because we could install it for free, but as our performance requirements grow, we might decide to splash out for a more powerful commercial offering.

Vendor independence also strengthens our negotiating position. Merely having the option to choose a different vendor allows us to ask for better pricing, additional functionality, bug fixes, and improved service. Guess what happens when a vendor knows that we're locked to their offerings? I've been there, and, trust me, it's an ugly place.

Platform neutrality minimizes our project's technology risks. In our fast-evolving sector, companies and technologies flourish and die at an amazing rate. If you're wed to a proprietary technology, you face the constant risk of a messy unanticipated divorce when the technology's vendor stops supporting it. In contrast, with portable code, you can choose the most beneficial technology at each point of time. Standardized technologies also tend to last longer, supporting your technology investment in the long term. Consider as examples Fortran and C versus some of the 1980s proprietary darlings such as Clipper, SQLWindows, and NatStar. There's no magic behind this phenomenon: selection bias ensures that mature technologies get standardized

0740-7459/13/$31.00 © 2013 IEEE

and widely adopted, and thus they outlast proprietary offerings.

Adopting widely used technologies will also help you in other, nontechnical areas. You'll be able to choose coworkers or employees from a deeper pool: advertizing a post for a Java programmer will yield many more candidates than opening one for an AcmeScript developer. Similarly, you're more likely to find good books, a vibrant support community, and training courses if you stick to standardized offerings.

The Hair Shirt

Sadly, striving for portability can sometimes be a thankless calling. In some domains, such as native applications with a graphical user interface, what you can write with portable code is laughable, if not entirely useless. At best, you might have to choose between delivering a system that requires platform-specific libraries (as is the case with Java's Standard Widget Toolkit) or one that doesn't quite follow the platform's native look and feel (think of Java's Swing). Performance will also suffer because vendors tend to offer their hottest code through nonportable bindings, like those of Microsoft's DirectX.

As another example, vendor-specific database bindings tend to perform better than vendor-agnostic ODBC/JDBC bridges. Adding insult to injury, portable code can be less expressive than code written using some nonportable extensions. Consider the nifty process and command substitution features of the bash Unix shell. To do the same things with the standard Unix Bourne shell requires ugly contortions involving temporary files and back-tick escaping. Similarly, you can simulate some of Oracle's analytical database query functions by means of nested queries. However, the result can be unreadable and could well perform worse than a query using the nonportable extensions.

Draw Your Lines

With most systems software implementing a standard and then helpfully also adding everything but the kitchen sink to it, writing portable code can be treacherous. An obvious solution is to disable all extensions; many compilers offer a flag that makes them standards-compliant. When this isn't possible or feasible, another practical solution is to code for one system using as documentation the official standard or that of another one. For instance, when writing code for MySQL, read the documentation for the corresponding SQLServer commands. To avoid hidden gotchas, strive to continuously compile, run, and test your code on a variety of platforms.

Wearing the portability hair shirt will deprive you and your customers from many benefits. One way around this conundrum is to draw boundaries around the nonportable code to isolate it from the rest of the application. If you're lucky, you may find a library or even a complete platform that offers you the functionality you need. For instance, HTML 5 lets you deploy sophisticated GUI applications through any modern Web browser. If you can't find a suitable portability layer, you'll have to do the heavy lifting by hand. Create a separate directory or file for the routines of each platform's code. If your language supports it, define the code's interface and implement a separate class for each platform. Often, vendors will provide a particular nonportable extension that uses slightly varying syntax or semantics, so the extra work might not be onerous.

Another approach is to admit defeat and go wild writing code that gives the best native experience. Keep in mind that we program to serve our business and customers, not just to satisfy our lofty ideals. Writing platform-specific code isn't as crazy as it sounds, even if you have to support multiple incompatible platforms. The idea is to let each platform's code base develop separately rather than introducing a platform's complexity into a single code base, for a unified base could well become exponentially complex. Worse, when some platforms (inevitably) die, you might end up having to maintain their complexity because it will be difficult to pry away their code from the integrated system. Choose your side!

> We program to serve our business and customers, not just to satisfy our lofty ideals.

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.

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