Imagine you are a programmer tasked with fixing a software bug. Depending on the scale of the system involved, you could be faced with millions of lines of code, so your first task is to identify the small subset of lines where you need to make changes. You then study this code to ensure that the changes you intend to make are appropriate. Finally, you try to identify where the changes you made might have unintended effects in the rest of the system. These effects might be many thousands of lines of code away and thus not easy to identify. Familiarity with the system might provide a crutch for your endeavors; however, in large systems, the chance that you are familiar with the specific code in question is remote.

**THE NEED FOR SOFTWARE VISUALIZATION**

One possible solution to this problem lies in software visualization. Visualization is the process of forming a mental image of something not readily apparent in sight. Examples include Underground maps, which allow users to plot a rail route from station to station, and Geiger counters, which allow users to perceive aurally the amount of radioactivity a certain source emits. As a research discipline, visualization typically refers to the efforts researchers make to present complex information to users in a form that facilitates performing specific tasks.

Software is a prime target for visualization research, as the introductory scenario illustrates. Software systems are not often “readily apparent in sight” due to the scale of code; the manner in which subtle interrelationships can spread specific functionalities over the code base; and the fact that the most reliable representation of the software (that is, the source code) provides only an indirect, static representation of a system’s behavior at runtime.

In addition, many users work extensively with source code. Students learning to program typically study, hack, and create programs. Likewise, professional programmers who develop, maintain, and evolve software systems work with source code intensively, albeit on a much larger scale. Several studies have shown that programmers who maintain and evolve software systems rely extensively on the source code, while also relying, to a lesser degree, on observing the executing system, consulting with trusted colleagues, and viewing system documentation.

**RESEARCH AND ADOPTION**

Researchers have carried out a large volume of academic work in the area of software visualization. Dedicated forums like VisSoft and the Program Visualization Workshop are coupled with forums such as the *International Journal of Visual Languages and Computing*, the Psychology of Programmers Interest Group, and the IEEE Workshop on Information Visualization to form a large body of knowledge in this area.

Given the amount of academic research, the adoption rate of academic software visualization tools...
into mainstream commercial software maintenance is surprisingly low. In “Reverse Engineering: A Roadmap” (ICSE, 2002), Hausi Muller and colleagues ascribed this low adoption rate to several factors: programmers’ lack of knowledge of these tools and their capabilities, interoperability issues between the tools and programmers’ existing development environments, and familiarization overhead.

Programmers typically use integrated development environments (IDEs) like Eclipse, Microsoft’s Visual Studio, and NetBeans to develop software in commercial organizations. Such IDEs often provide plug-in capabilities that researchers can use to add their own tools for software visualization. Such facilities explicitly address the interoperability issue and lessen the familiarization overhead, as the programmer’s environment typically changes only incrementally.

The suggestion that the adoption rate is low because programmers lack knowledge of software visualization tools is based on the premise that programmers would adopt these tools if they knew about their capabilities. This, in turn, implies that these tools’ capabilities are desirable. So ultimately, as in any software system, getting the software visualization tools’ capabilities correct becomes a core issue in the adoption and success of these tools.

Requirements acquisition

To ensure that software systems offer the correct capabilities, software traditionally undergoes a substantial requirements acquisition phase. In this phase, researchers observe users carrying out their work, ask about the benefits and limitations of their current systems, and establish their wish list for the new system. However, this phase does not often occur in software visualization tool development, resulting in the problem that “designers may only have an intuitive notion of what features are beneficial” (M.A. Storey et al., “How Do Program Understanding Tools Affect How Programmers Understand Programs?” Science of Computer Programming, Mar. 2000, pp. 183-207).

Researchers can address this issue in retrospect by thoroughly evaluating existing tools. However, the combined difficulties of acquiring professional programmers to participate in these studies, making the context sufficiently realistic, and ensuring that possibly biasing factors are controlled means that “most [software visualization] system developers perform little or no empirical evaluation,” according to John Stasko and colleagues (Software Visualization: Programming as a Multimedia Experience, MIT Press, 1998). Vic Basili echoed this sentiment in his 2005 keynote address to ICSE, claiming that researchers too often evaluate academic software tools with respect to their correct functioning, rather than their utility to the user.

More recently, however, the academic community has begun to focus more on programmers’ requirements. Several studies have focused on how programmers seek information as they work, with some interesting findings. For example, in “Information Needs in Collocated Software Development Teams” (ICSE, 2007), Andrew Ko and colleagues categorized the information that commercial programmers seek by frequency, perceived importance, and availability. They found that programmers frequently query other team members’ activities but perceive information on these activities as readily available. In contrast, they found that the programmers spend a lot of time wondering about the causes of specific program states and behaviors, including failure-type behaviors. The programmers deem this information important but often unavailable, resulting in a lack of resolution to their information searches.

Similarly, programmers often inquire about the intended purpose of the code, again deeming this information important but not readily available. Analogous work in the open source domain suggests that noncommercial programmers also have a high proportion of team-based queries, frequently wondering about the purpose of specific pieces of code and wondering how and where the code base achieves certain functionalities.

Programmer IDEs

In a related area, researchers have recently begun to probe the effectiveness and inadequacies of current IDEs in providing the information that programmers seek. For example, in “Using Visual Momentum to Explain Disorientation in the Eclipse IDE” (VLHCC, 2006), Brian de Alwis and Gail Murphy reported that Eclipse, while typically supportive of programmers’ efforts, can disorient professional programmers by not providing explicit code navigation paths. They also noted that some programmers navigate between different parts of the code base frequently and repeatedly, a practice known as thrashing. They interpret this thrashing as a desire to juxtapose code through navigations, suggesting that programmers don’t want to use their limited screen space by employing multiple code-browsing windows simultaneously.
As Figure 1 shows, such studies are vital to the generation of well-formed software visualization tools. Only by studying professional programmers in practice, as well as the IDEs they use to develop and maintain software, can we begin to identify the relevant yet obfuscated information programmers need to maintain and evolve software systems. Once identified, visualization experts can then generate congruent representations of this relevant information for the programmers to view.

TOWARD REQUIREMENTS-BASED SOFTWARE VISUALIZATION TOOLS

Already, some visualization tools are appearing that align with these studies’ findings. For example, programmers who are browsing code for Eclipse, to expand method calls and class declarations as they come across them, in situ. The method or class expands in the current browsing window, just below where it is referenced in the source code. Thus, this tool partially addresses both the navigation trail and juxtapositioning needs that de Alwis and Murphy reported.

Another such tool is Mylyn (www.eclipse.org/mylyn), also an Eclipse plug-in. This tool keeps track of the small working subset of code that programmers refer to in the context of a specific maintenance task by creating a visible, organizing entity in the IDE for each task. When completing a task, the tool tracks and highlights the recently edited code through the package explorer view, so programmers can easily locate the code they have already edited in the context of their current task—code that is more likely to be relevant to their current work. In addition, a team of programmers can use Mylyn to share their task histories, allowing them to locate relevant code quickly if other team members have previously addressed similar tasks. Thus, this tool partially addresses the team awareness and code localization issues raised in the studies described here.

These visualization tools illustrate that requirements acquisition work in this domain needs to expand. But there are difficulties. In vivo studies of professional programmers in their naturalistic work contexts are difficult to perform. Acquiring participants is hard, and commercial sensitivities often render certain data inaccessible to researchers. In addition, controlled experiments are largely impossible in these contexts, as researchers can’t control all possible impacting factors. But without empirical studies in this domain, the visualization community runs the risk of building pretty representations of information that maintenance programmers don’t really need.

Jim Buckley is a researcher at Lero—the Irish Software Engineering Research Centre and a faculty member of the Computer Science and Information Systems Department at the University of Limerick, Ireland. Contact him at jim.buckley@lero.ie.

The author thanks Clem O’Donnell for his comments in reviewing this work.