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

Tool Building on the Shoulders of Others

Holger M. Kienle, Adrian Kuhn, Kim Mens, Mark van den Brand, and Roel Wuyts

Software engineering requires adequate tool support. Software engineering research is no different. Many researchers in this field build and use advanced tools and prototypes to either validate their own research ideas or advance the state-of-the-art in software engineering tools and techniques. The 2008 European Conference on Object-Oriented Programming introduced the International Workshop on Advanced Software Development Tools and Techniques, the first workshop dedicated to academic tool building. At that workshop, researchers shared their experiences and discussed how to build tools more effectively and efficiently. The 15 tools we looked at covered a broad range of topics, including refactoring, modeling, behavioral specification, static and dynamic program checking, user interface composition, and program understanding.

In this column, we identify four emerging trends in academic tool development.

Standing on the Shoulders of Others

Perhaps the most important observation was that even for research tools, you can gain a significant advantage by building them on top of existing tools and frameworks. Although you might think that every researcher makes his or her own prototype tool in isolation, most of the presented tools relied on external code. In fact, tool builders can leverage many frameworks, libraries, generative languages, and open source tools to build on or to integrate in their own tool. For example, program-understanding tools can reuse functionality to parse and analyze the target code, transform and store it, and visualize it.

One tool that builds on the shoulders of others is IntensiVE (Intensional Views Environment), a tool suite for documenting structural source code regularities (such as design patterns and coding conventions) in object-oriented software systems and verifying their consistency during those systems' evolution. It builds on the shoulders of many other tools, including the SOUL (Smalltalk Open Unification Language) logic metaprogramming language as a query mechanism, StarBrowser to present and browse through the regularities, JavaConnect to access Java parse trees in Eclipse, and Mondrian to visualize the source code entities.

A big part of IntensiVE's strength comes from how it relies on and combines those other tools. It not only considerably speeds up development but also enhances the tool's overall power and quality, sometimes in ways the original tool developers didn't foresee. Such code reuse is advantageous for both developers and users. Tool developers can focus on their tool's novel features without getting bogged down in low-level plumbing. Also, external code written and packaged by domain experts is often superior to greenfield, greenhorn coding. On the receiving end, tool users can become more productive if they already know an abstraction that the tool reuses, such as a graph description language or a query language.

Dynamic Languages on the Rise

Our next observation was the prominence of tools using dynamic languages, primarily Smalltalk but also Tcl and JavaScript. This is because experimental prototypes and tools are subject to continually changing requirements and research insights. The development of such tools is thus highly incremental, continually requiring rapid adaptations and evolutions. Dynamic-language environments meet these requirements while offering adequate tool performance.

For example, CodeCity is a highly interactive tool that visualizes software in 3D, following a city metaphor. The city's buildings represent classes, which are placed in districts representing the packages. CodeCity is implemented in Smalltalk. Furthermore, it provides a scripting abstraction based on Smalltalk that lets you easily reconfigure a visualization and experiment with different visualization approaches on-the-fly. For research on programming-language design and software development environments such as the Hopscotch framework, having a dynamic language as both the implementation and development language results in an ideal playing field for experimentation.

Tools as Web 2.0 Bandwagon Jumpers

Third, we noted that several tools deliberately either represent themselves as Web 2.0-like applications (Churrasco and the Small Project Observatory) or take inspiration from Web metaphors to provide novel user interfaces (such as Hopscotch's use of a back button, history, links, and so on for browsing source code). Churrasco supports collaborative program understanding based on Web 2.0. Once developers set up a project by pointing Churrasco to the project's SVN (Subversion) repository, they can explore the code through a variety of interactive visualizations. They can annotate software views with textual notes that are immediately visible to other developers. By exploiting recent Web 2.0 technologies and Scalable Vector Graphics with embedded JavaScript for rendering, Churrasco incorporated all this functionality in a Web browser. For tool users, this means that the tool is available without installation (aside from missing plug-ins), is accessible from different platforms, and stores application data in the "computing cloud."

However, tools don't necessarily have to go Web 2.0. The CScout refactoring browser tackles the complex task of refactoring untamed C code. The tool has a vanilla HTML interface that supports renaming identifiers. However, it also provides hyperlinked code browsing and form-based querying and metrics calculations on identifiers, functions, and files. CScout generates Web pages identified by unique URLs, so users can bookmark them for easy access.

Moore's Law Helps Not Just Fancy Office Suites

Our final observation was that many tools used to struggle with large, real-world code bases, not only because such code exhibits many idiosyncrasies but also because of performance problems in terms of space and time. Even though the targeted systems' complexity has also increased, Moore's law seems to have come to many tools' rescue. Until recently, processing large real-world systems would have been beyond most program-understanding tools. Such targets are now in reach, as demonstrated by CScout, which processes the Linux kernel (4.1 MLOC) in a bit less than 7 1/2 hours. To accomplish this, CScout places high demands on computing resources (multigigabyte memory and a 64-bit CPU).

Established tools also profit from Moore's law. Over the last 10 years, the Rigi reverse-engineering environment has been able to process code bases of increasing size without rewriting a single line of code.

In addition, Moore's law helps make tools more interactive. Integrating CScout into an integrated development environment (for example, for interactive refactoring) seems feasible for smaller (up to 10 KLOC) projects. CScout can process awk (6 KLOC) in less than a second. CodeCity is another example where increasing computing power translates into a more powerful tool. Even though 3D rendering is demanding and based on a Smalltalk OpenGL library, CodeCity manages fluid visualization of, and interaction with, a megacity comprising nine systems that make up more than 17,800 classes.

We invite you to go to the workshop Web site (http://smallwiki.unibe.ch/wasdett2008/tools), take a closer look at the exciting tools that we could only briefly mention here, and try them out. You can even be bold and stand on their shoulders!

Even though the targeted systems' complexity has also increased, Moore's law seems to have come to many tools' rescue.

Holger M. Kienle is a postdoctoral researcher in the University of Victoria's Computer Science Department. Contact him at kienle@cs.uvic.ca.

Adrian Kuhn is a PhD student in the University of Bern's Software Composition Group. Contact him at akuhn@iam.unibe.ch; www.twitter.com/akuhn.

Kim Mens is a full-time professor at the Université catholique de Louvain-la-Neuve's Department of Computing Science and Engineering. Contact him at kim.mens@uclouvain.be.

Mark van den Brand is a full-time professor of software engineering and technology at the Eindhoven University of Technology. Contact him at m.g.j.v.d.brand@tue.nl.

Roel Wuyts is a senior research engineer in the independent not-for-profit research institute IMEC and a professor in Katholieke Universiteit Leuven's Computer Science Department. Contact him at wuytsr@imec.be.