Editorial Preface

Steve Goschnick, School of Design, Swinburne University of Technology, Melbourne, Australia

Although not a special issue, this Issue of IJPOP nonetheless brings together 3 papers and a book review that share something of a common thread: they are each about consolidation of software development and programming practices, through improvements to existing tools, methods or practises. In all cases, they address some aspect of broadening the cohort of people that use, design or program the ubiquitous digital devices in our lives, and in so doing, underline a key quality of People-Oriented Programming.

The first paper, *Natural Shell: An Assistant for End-user Scripting* by Xiao Liu, Yufei Jiang, Lawrence Wu and Dinghao Wu, is about turning natural language describing a task, into a shell script that does the deed. Shell scripting is widely used for automating tasks at the Operating System level - sometimes tasks involving multiple applications from different sources or vendors. Despite the popularity of shell scripting, it remains difficult to use for both beginners and experts alike, but for different reasons. Beginners are confronted by a barrage of cryptic commands and concepts, while experts struggle with the often incompatible syntaxes across different systems, made worse by seemingly similar syntax, that can be used for some completely different tasks. The authors solution is a system they call *Natural Shell* that transforms the natural language describing a task we want done on our computer, into the appropriate shell scripts that make it happen. It is a multi-step process. Firstly, it translates the natural language of the user into an intermediate, system-agnostic script called Uni-shell. In the remaining steps, Natural Shell transforms the Uni-shell script into the appropriate underlying native script, executes it, then reports details about the resulting action. In an affirming and reinforced-learning manner, the user then gets to see all the steps involved: the natural language they entered, the generated Uni-shell script, the converted native script, together with the system-reported results. Their system is a prototype that they trialled with test subjects, applying it to tasks that together require some 50 common shell commands. Natural Shell generated the correctly needed shell script in over 80% of the test cases.

As our user interfaces move further towards natural language and voice input, solutions such as Natural Shell are becoming increasingly important to people-oriented programming. At the major vendor level we already have intelligent agents such as Apple's Siri, Google Assistant, Amazon's Alexa, Microsoft's Cortana and Wolfram Alpha - that extensively draw upon data, services and knowledge out on the Internet. However, it is at the Operating System level, that we tend to organise our own data, and where we service the requests and tasks, asked of us by others. We ourselves are a part of a complex human-agent system, each supported by algorithms and AI, and inputting our own specialised skills (Goschnick & Sterling, 2003) into the system. Natural Shell is addressing issues that are physically closer to our end of the deal, as the human in these human-agent collaborations of which we are all steadily becoming a part.

PROTOTYPING MULTIMODAL INTERACTION

The second paper in this issue, *Hasselt: Rapid Prototyping of Multimodal Interactions with Composite Event-Driven Programming* by Fredy Cuenca, Jan Van den Bergh, Kris Luyten and Karin Coninx - reports on an advance in prototyping applications involving multimodal interfaces, the sort of interface at home in the discussion around the first paper above.

A composite event is best introduced by example. A well-known example of multimodal interaction is the *put-that-there* interaction (Bolt,1980), which can be represented by a composite event, in turn made up of discrete primitive interaction events (e.g. like speech acts, hand gestures or mouse movements). The spoken "put" identifies the intended action. The spoken "that" can be identified by a mouse click or touch event on a selected onscreen target object. Then the cursor is moved to another onscreen location, where another click or touch event identifies the "there" - the place where the program needs to put the onscreen object.

When interaction designers/developers are prototyping such multimodal interactions, the resulting code in conventional event-driven languages, ends up being a "callback soup" according to the authors, adding complexity and increasing the rate of errors. The numerous iterations of the prototype by a competent development team exploring different options and solutions, amplifies these problems that hamper the multimodal interaction designer/developer. Their solution is a technology they call *Hasselt*, which uses a clear and simple mini declarative language to describe the composite events. It raises the level of abstraction, and renders the code produced by each design iteration, devoid of that 'callback soup'. The researchers ran a qualitative trial of their approach, comparing Hasselt with a traditional programming language that supports event-driven coding, involving 12 participants each with a solid background in software development. Hasselt lead to higher completion rates, lower completion times, and less code testing than with the traditional event-driven language.

NAMING CONVENTIONS FOR PROGRAMMERS

The understandability of code that has been written some time ago or by somebody else, has a significant value that is often under-appreciated. Whether it is commercial team members or open source collaborators that need to comprehend each others work, naming conventions play a large part in such communication. Its a case of, the left hand knowing what the right hand is doing, or has done. Large swathes of a program are usually made up of aspects other than the command syntax of the language itself. The names a programmer gives to newly coded methods and functions will usually be given verbs as names, to emphasise the underlying actions they perform. Variable names will usually be nouns. As the limits on the length of such names in modern language have grown longer, there is now much emphasise on choosing names that are self explanatory, in-situ, reducing the need for further commenting away from where the action is. (In the book reviewed in this issue, the author tells us "Good code is like a textbook. … Code should explain what is happening; it should be self-explanatory.") And with those larger names along came *camelCase* - a mixture of upper and lower case character in the one name, to distinguish words without the need for spaces - and other notations with similar intent. Hence, the nouns have grown to become noun-phases, and verbs extended to verb-phases, often including prepositions, adverbs and adjectives. E.g. A method that can draw a circle of any size might be called: DrawCircleOfAnySize (real theRadius), in camelCase. Into this new microcosm of meaningful expression, all within the program source code, came more extensive naming conventions for modern programming languages.

The Java language started life in a very professional environment within the company Sun Microsystem, in 1995. Java was the first mainstream language to embrace several modern concepts such as: making unicode the standard intrinsic character set rather than ascii, to facilitate the internationalisation of programs; building it with the Internet in mind from the ground up; and including low maintenance memory management through automated garbage collection. So, the laying down

of a *naming convention* for Java programmers to adopt, wasn't too far behind the introduction of the language itself (Sun Microsystems, 1999). Then Microsoft, an early adopter and promoter of the Java language, yearned for its very own language of similar modernity and influence. They made it so with the launch of the C# language in 2002, seven years after Java made its debut. They too saw the professionalism in making a naming convention available for programmers to follow, as early as possible (Wiltamuth and Hejlsberg, 2003).

Compliance with naming conventions is a strong indicator of source code quality. By adhering to them a programmer improves the code comprehension, maintainability and reusability of their code. The third paper, titled *An Empirical Comparison of Java and C# Programs in Following Naming Conventions*, by Shouki Ebad and Danish Manzoor, examines the relative adherence to the respective naming conventions, by open source programmers. They evaluated 120 arbitrarily selected open-source Java and C# classes, each by different programmers. Their results indicate that Java and C# programmers, unsurprisingly, do not always follow naming conventions. However, Java developers are more likely to do so than C# developers. Four positive observations were derived from their data: (1) most of the Java subjects followed most naming conventions, (2) most C# subjects followed *class* and *method* naming conventions, (3) most Java and C# subjects that violated the conventions had only one violation, and (4) there is a positive correlation between the violations found in a C# class and its size (but not so for Java programmers).

While the programming languages chosen for their study are but two of many, C# and Java each have significant cohorts of professional programmers, and share an overlapping history, competitiveness and rivalry. The paper adds to a small but growing body of such comparative studies regarding the usage of naming conventions by programmers, and is valuable for it.

BOOK REVIEW

The book reviewed is titled *Speaking JavaScript* by Alex Rauschmayer, and is an excellent book to learn the JavaScript language from, if you already have some other programming language under your belt. Modern JavaScript is far from the light-weight scripting tool it started out as, in 1995. It is a language that has been extruded between the tectonic plates of the commercial and open source worlds. A place where open standards have persisted, despite much commercial competition and pressure, to undo them. It is also a language that has advanced dramatically with support through innovation and investment from many of the commercial players involved. Modern JavaScript is a language that has been updated and refined to remain essential in the web browser, increasingly so on the server and the mobile device, and useful elsewhere too. The broad-ranging uses of JavaScript highlight its success. For example:

- Beyond the browser, Web apps can be used as native apps on several platforms including Chrome OS, Firefox OS and Android;
- JavaScript can be added to digital books, giving them interactivity within EPUB3, the standard that lies beneath several ebook formats including those in Apple's iBook store;
- JavaScript can be used to code player-character-environment interactivity in the Unity 3D game engine;
- The innovative Node.js environment lets JavaScript programmers, code in it, server-side;
- There are cross-platform development environments such as PhoneGap and Cordova, that allow programs coded in JavaScript to run natively on iOS, Android, Windows, Linux and so on;
- More recently React Native, a Facebook developed technology, is freely available to define user interfaces for both iOS and Android devices, for apps coded in JavaScript.

And on the list goes.

JavaScript is now a mainstream programming language. If you haven't given it a thought for some years or at all, this book might entice you to consider learning it? Michael Bloomberg, the billionaire philanthropist and former New York mayor did in 2012. As have millions of high school students in recent years, around the planet.

Steve Goschnick
Editor-in-Chief
IJPOP

REFERENCES

Bolt, R. A. (1980). “Put-that-there”: Voice and gesture at the graphics interface. Computer Graphics, 14(3), 262–270. ACM.

Goschnick, S. B., & Sterling, L. (2003). Enacting and Interacting with an Agent-based Digital Self in a 24x7 Web Services World. *Proceedings of the Workshop on Humans and Multi-Agent Systems*, *AAMAS-2003 Conference*, Melbourne, Australia.

Rauschmayer, A. (2014). *Speaking JavaScript*. Sebastopol, USA: O’Reilly Media.

Sun Microsystems. (1999). Code conventions for the Java programming language. Retrieved from http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

Wiltamuth, S., & Hejlsberg, A. (2003). C# Language Specification. Retrieved from http://msdn.microsoft.com/en-us/library/ff926074.aspx