Deverywhere: Develop Software Everywhere

Yishai A. Feldman,* Ari Gam,† Alex Tilkin,‡ and Shmuel Tyszberowicz‡
*IBM Research – Haifa, Israel; Email: yishai@il.ibm.com
†Blavatnik School of Computer Science, Tel Aviv University, Israel; Email: theprap@gmail.com
‡School of Computer Science, The Academic College Tel Aviv Yaffo, Israel; Emails: alextilk@gmail.com, tyshbe@tau.ac.il

***Abstract*—Professional programmers use desktop or laptop computers as a preference. However, they sometimes need to continue their work on the go, when they may only have access to mobile devices. Thus, mobile devices can be important but not exclusive development platforms. Therefore, it is necessary to support programming in conventional languages on mobile devices, such as phones and tablets.**

Programming on mobile devices presents two major obstacles: the lack of a physical keyboard, and the small screen space, which limits the amount of code that can be shown simultaneously. This paper addresses both challenges, and offers a method to enable programming on mobile and other devices with limited input and output capabilities, by using templates to make voice and touch input very effective for programming, and showing much more code in a limited space. These ideas are also relevant to programming on laptop and desktop systems, for people with disabilities such as repetitive-stress injuries (RSI) that limit keyboard usage, and partial vision loss, which requires the use of very large fonts.

I. Introduction

In the early days of computing, programmers had to work in offices. Personal computers allowed programmers to work at home as well. Laptops further expanded the working environment, and we often see people programming in coffee shops, terminals, trains, and airplanes. With ubiquitous mobile devices becoming increasingly popular, there is an opportunity to allow programmers to work in even more environments. While such small devices are unlikely to become the preferred working medium, they can be useful in circumstances where urgent action is required and other equipment is unavailable.

This scenario presents two major obstacles: first, the lack of a convenient keyboard; and second, the small screen space, which limits the amount of code that can be shown simultaneously. Some have advocated the creation of new programming languages for mobile platforms,[1] but the cost of adopting a new language, with its related tools and infrastructure, seems to be too great for the benefit of occasionally programming on a mobile device. This applies to the development of mobile and non-mobile applications alike; professional programmers who develop mobile applications still prefer to use large screens and physical keyboards. Instead, we focus on easy ways to use existing languages, such as Java and JavaScript, on mobile devices. Our proposed solution, called *Deverywhere*, addresses both challenges, by using templates to make voice and touch input very effective for programming, and to show much more code in a limited space. Templates, used in context, allow voice input for program creation, editing, and navigation; and allow a compact representation of programs that makes maximum use of the given screen space. Both uses require a high degree of configuration, since programmers have different preferences regarding the way they want to voice and see programs. The underlying representation is always the original language, so that each programmer can see a tailored view while seamlessly collaborating on the same code with others.

These ideas are also relevant to programming on laptop and desktop systems, for people with disabilities such as repetitive-stress injuries (RSI) that limit keyboard usage, and partial vision loss, which requires the use of very large fonts. For some programmers, no screen is large enough, and so we expect that these programmers will use the compact representation of code even on large displays.

[1]See, for example, the "Theme and goals" section of the PROMOTO 2014 Workshop, http://research.microsoft.com/en-us/events/promoto2014.

II. Programming by Voice and Touch

Dictation systems exist today, but their use for programming is extremely limited. Lacking any domain knowledge, they require most of the program to be dictated letter by letter, which is impractical. By building an understanding of program syntax and some semantics into the dictation tool, it is possible to make this process much more efficient. For example, the spoken words "for i from zero to n" can be interpreted as the Java idiom

```
for (int i = 0; i < n; i++) body
```

The current insertion point would be left at the body of the loop. Furthermore, Deverywhere will know that this place in the template is called "the body," so that a further instruction to "edit the body of the for loop" will return to that point. Touch can alternatively be used for the same purpose.

Other locations can also be associated with templates. For example, the template above can define "the loop index" as referring to the variable `i`, so that the developer can later say "rename the loop index to j." The general form of a Java `for` statement can define the "initialization," "test," and "update" locations, referring to the three parts inside the parentheses. A conditional template may have three locations, referring to the condition, the consequent, and alternative.

Similarly, an "iterate" template can be defined to generalize the use of iterators that are not accessible through the `Iterable` interface. For example, suppose that `tree` is an element of class `Tree<Element>`, which does not implement `Iterable` but provides an iterator through a

method `inOrderIterator()`. The utterance "iterate on tree in order" will create the following code:

```
Iterator<Element> iterator =
   tree.inOrderIterator();
while (iterator.hasNext()) {
    Element element = iter.next();
    □
}
```

The box represents the current insertion point. In this example, the types of the iterator and the element have been inserted automatically, based on the type of `tree`. Their names have been chosen heuristically, but they can be changed by the developer; they can later be referred to as "the iterator" and "the element," respectively.

Such templates can be created for language constructs as well as for other types of patterns, such as application frameworks. For instance, we can dictate "event key is shift enter" for the following common Dojo expression that checks the details of a keyboard event:

```
event.keyCode == dojo.keys.ENTER
&& event.shiftKey
```

As part of the template-based input method, developers will be able to specify the kind of syntactic element they are about to enter, such as class, method, variable, or constant. A suitable template will be applied; for example, a constant in Java will automatically be defined as `public static final`. Similarly, a "main method" will open with the already supplied header `public static void main(String[] args)`. The same words ("main method," "constant X," etc.) can be used to navigate to the appropriate element. In addition, it should be possible to refer to elements according to their position in the text shown on the screen; for example, "first for loop," "inner if statement," "loop on i." These templates should be recognized regardless of how the code was entered. This implies that Deverywhere should recognize templates from the original language text, without relying on any external annotations.

A convenient way to add templates should be provided, as the number of possible templates is unlimited, and may even be programmer-specific. Each template is associated with utterances to create it, with named locations, and with its compact representation. The utterances form a grammar, which need not be completely unambiguous, since the development environment can offer a choice between alternatives. This, however, should be avoided as much as possible.

The same utterance and compact representation can be associated with more than one code template, in order to support multiple programming languages. One of the significant advantages of this template-based system is its ability to treat multiple languages in a similar way (to the extent that the languages provide similar mechanisms, of course).

Context is crucial to understanding. For example, names in a program are usually limited to a relatively small set, which depends on the current scope; a number of methods can be used to select the correct one efficiently. One way is to start naming a variable (or class name, method name, etc.) either by spelling or, if it is composed of known words, by sounding them. Once the choice becomes small enough to show it on the screen, completion can be made by sounding the number of the correct choice. Another way is to assign short nicknames to variables and other named elements, then refer to them by their nicknames. The initial definition of a variable cannot be made in this way, as the space of choices is unlimited. If the name is a known word or a series of known words, they can be dictated and Deverywhere can join them in the way appropriate for the programming language and the type of the element; for example, using CamelCase with appropriate capitalization in Java.

Refactoring and other source transformations, as in Eclipse, are a must for Deverywhere, with the appropriate modification of the relevant wizards to work with voice entry. In addition, other capabilities would be useful. For instance, suppose the developer wants to use the result of a method call that is part of an expression in another context. In Eclipse, the developer would have to mark the method call expression, then apply the Extract Local Variable refactoring, then move the generated variable definition upwards if necessary, and finally use it. Instead, in the voice-programming system, the developer will be able to say "use the result of the second call to substring." This will create the variable at the correct position, and insert a use of the variable at the current insertion point.

Statically-typed languages such as Java are often very verbose, especially in the specification of types. However, type inference methods exist for these languages, and current IDEs use them to identify errors and fix them automatically. The voice-programming system will not require (yet allow) the specification of types at any point, and will try to infer as much information as possible. In the example above, "for i from zero to n," it is clear that `i` is an `int`, and this need not be mentioned at all. When type inference fails, information about the variable may be limited; for example, the system will not be able to suggest methods for that variable. Therefore, the developer should always have a verbal command that adds a type definition to the variable at the current insertion point. This will add the missing type at the point the variable is defined, but will not change the current view or the current insertion point, so that the developer can continue choosing the method to call without further distractions.

III. Programming on Small Displays

While mobile displays are getting larger, they are still significantly smaller than the displays developers are used to. Even a 7" phone/tablet provides much less screen space for programming than a typical 21" desktop display. The amount of code that developers can see simultaneously has a great effect on their productivity. Special techniques are therefore required in order for programming on small displays to be effective. The templates that are used for voice entry can also be used for display.

There are many ways to show information visually in a compact way; for example, using special symbols, colors,

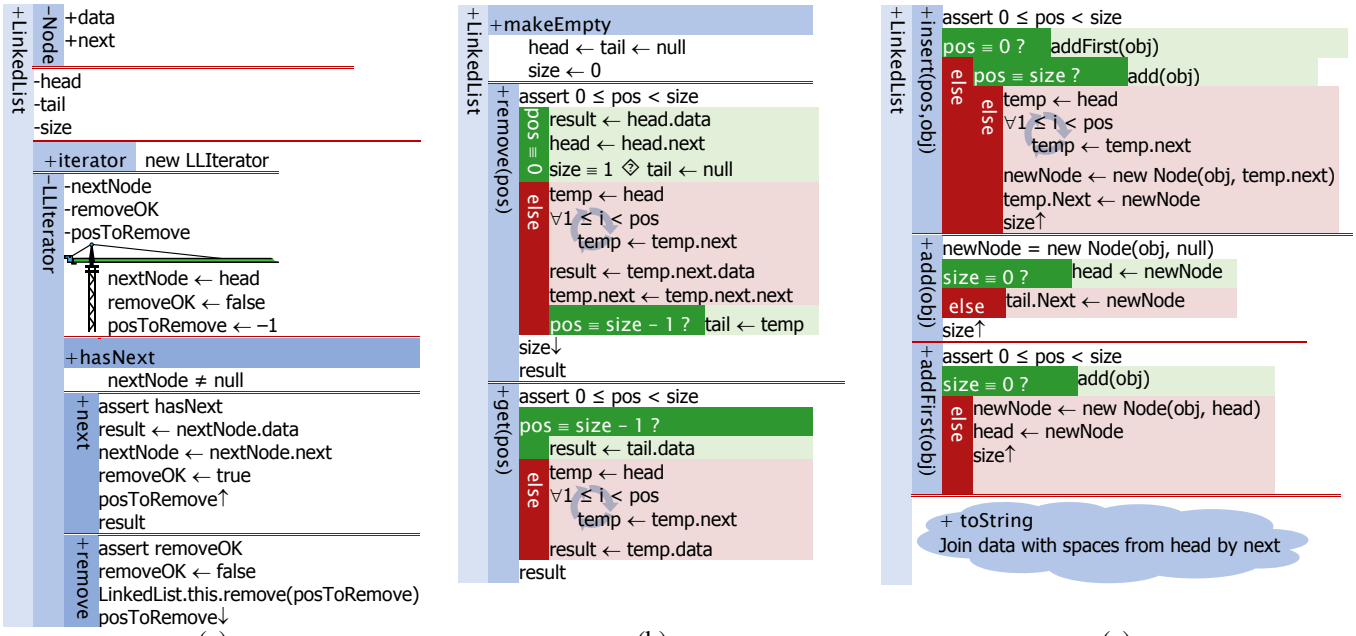

Fig. 1: The `LinkedList` class, as would be shown by Deverywhere.

fonts, backgrounds, borders, and even watermarks. Some information, such as noise words (`then`, `else`, `end`, etc.), types, throws declarations, access levels, and package prefixes, can even be omitted altogether (with an option of showing them selectively, perhaps using a touch gesture).

The example in Figure 1 shows many of these techniques; it may be extreme for some tastes, but, as mentioned above, the way program features are shown in the compact representation must be individually configurable, for the preferences of each developer and also based on screen size. This particular presentation of the program should be thought of as an example of how Deverywhere can be customized rather than as the definitive output format. The figure shows the code separated into three parts, each of which fits the size of a reasonable smartphone.

This code of example is taken from the University of Texas CS307 course of 2011 (https://www.cs.utexas.edu/~scottm/cs307/javacode/codeSamples/LinkedList.java), and implements a linked list. It has not been modified in any way except for the presentation, and the intent is for the code kept in the source-control repository to be the same as the original.

This example demonstrates many possible features of compact representations of programs. Names of enclosing classes and methods appear on the bar that also serves for the indentation, unless the bodies are very short. Many language elements are omitted (comments, types, keywords such as `class` and `return`, empty pairs of parentheses, and symbols such as statement terminators and braces); others have been replaced by short notation (+ and − for `public` and `private`, the equivalence symbol for ==, arrows for ++ and −−, and special notation for constructors, conditionals, and loops). Background colors are used to show scopes (shades of blue for classes and methods, green for the `then` and red for the `else` part of conditionals); the scope of a constructor is denoted by the extent of the crane symbol. Note that class and method names appear at the start of every relevant screen, not just at the beginning of their scopes.

Templates have been used in several places to make code more concise (and perhaps also more readable). Getters and setters have been compressed to look like field reads and writes, as is done in several languages (but not in Java). The common mathematical idiom $0 \leq x < s$ is used instead of the cumbersome notation that uses a conjunction of two inequalities. The same notation is used here for loop bounds, in the (very common) case of a unit increment.

Conditionals are shown here using the same block structure as classes and methods, except for the colors used to denote the different branches. One exception, demonstrating a somewhat more conventional way of representing a conditional, appears inside the `remove` method in part (b). This notation is similar to the three-way conditional expression of Java, but uses a different symbol to denote that it is a statement rather than an expression.

Loops are shown with a circular watermark that denotes the scope, and a special symbol (∀) to indicate the loop control. This is perhaps the most unusual notation we show in this paper; while it may be chosen by few developers, we use it to show the variety of notation that can be used for a compact representation of programs.

The last method, `toString()`, is different from the other parts of this example. In some cases, it is useful to show some

documentation (perhaps, but not necessarily, the Javadoc) instead of the method body. This offers a very concise view that shows the intent without the implementation; this is often very useful.

Other possibilities, not shown in this example, include automatically inlining a value that is only used once provided the resulting expression is not too large; this could have been done (twice) for `newNode` in part (c). Existing Eclipse templates, such as expanding `sysout` to `System.out.println`, can be used in reverse to compact programs that use the expansions of such templates. This can be extended even further; for example, given an analysis that converts imperative idioms to functional ones, such as LambdaFicator [6], the view can show the functional form without changing the underlying program.

As can be seen from this example, the representation is sometimes ambiguous. This seems to us to be acceptable, since the meaning will in most cases be obvious from the context. In any case, the developer will always be able to ask to see more details (perhaps by touching locations for which more information is desired).

Program slicing, and especially Fine Slicing [1], is a very effective technique for showing a small part of the code that is relevant to a particular purpose. When browsing code, slicing can be used to prevent the need for a lot of scrolling. The program view would normally consist of a single method, possibly with some context given as a breadcrumbs view. In some cases, it is useful to show enclosing conditionals and loops, but without intervening details; this is a kind of poor-man's slicing that is more easily implemented.

IV. Roadmap

In order to map the requirements and possibilities of development on limited platforms, we performed several dictation experiments where the speaker tried to dictate a program as naturally as possible, and the writer attempted to understand as literally as possible. In addition, we studied previous work on mobile programming environments [5], [7], compact representations [3], [8], [9] and dictation [2], [4], and abstractions used by various programming languages.

Previous work on mobile programming environments [5], [7] has focused on new languages that are more natural for the mobile environment. In contrast, we believe that such new languages will have very limited use, mostly for small applications. Professional development will continue to be done in more established languages, and a mobile solution for professional programmers should support these languages without requiring changes in the underlying technology.

We found many interesting ideas in Intentional Software [9] and registration-based abstractions [3], [8], which have discussed multiple views of the same underlying program. VoiceCode [4] and Spoken Java [2] focus on voice entry of programs. Deverywhere combines voice input with a compact presentation using the same set of templates.

The results of our study are several lists and relationships. One list contains features that a mobile programming environment can support; in addition to those discussed above, the list includes items such as the automatic application of quick fixes, renaming conflicting elements, extension methods [10], and two-dimensional expressions (as in mathematics). A second list contains features for compact representation; a third lists features for voice input; and a fourth lists configuration modes, such as the use of typographic styles, layouts, frames, and watermarks. We also created a matrix relating input and output options with the various ways each of these can be shown; this will be the basis for the developer-specific customization.

V. Conclusion

Deverywhere relies on templates that have information about their structure that supports flexible voice input and various types of compact representations. Customizability is crucial to support different personal styles as well as different sizes and capabilities of programming environments.

Once fully implemented, Deverywhere has the potential to make programming on devices with limited user interfaces, such as mobile phones and tablets, much more convenient than they are now. This will allow developers to work more comfortably in environments in which programming was very difficult.

These techniques can also provide enormous help to developers who have various kinds of disabilities that prevent them from using existing interfaces on laptop and desktop environments effectively. In fact, we conjecture that many developers would enjoy having the benefits of Deverywhere as additions to their normal working environments.

References

[1] A. Abadi, R. Ettinger, and Y. A. Feldman. Fine slicing: Theory and applications for computation extraction. In *Proc. 15th Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 471–485, Mar. 2012.

[2] A. Begel and S. L. Graham. Spoken programs. In *IEEE Symp. Visual Languages and Human-Centric Computing*, pages 99–106, Sept 2005.

[3] S. Davis and G. Kiczales. Registration-based language abstractions. In *Proc. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 754–773, 2010.

[4] A. Désilets, D. C. Fox, and S. Norton. VoiceCode: An innovative speech interface for programming-by-voice. In *Extended Abstracts on Human Factors in Computing Systems (CHI)*, pages 239–242, 2006.

[5] G. Essl. Mobile phones as programming platforms. In *Programming Methods for Mobile and Pervasive Systems (PMMPS)*, 2010.

[6] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. LambdaFicator: from imperative to functional programming through automated refactoring. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1287–1290, 2013.

[7] S. Li, T. Xie, and N. Tillmann. A comprehensive field study of end-user programming on mobile devices. In *IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC)*, pages 43–50, September 2013.

[8] J.-J. Nunez and G. Kiczales. Understanding registration-based abstractions: A quantitative user study. In *Proc. IEEE Int'l Conf. Program Comprehension (ICPC)*, pages 93–102, 2012.

[9] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *Proc. 21st Annual ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 451–464, 2006.

[10] Xtend – modernized java. http://eclipse.org/xtend.