KenyaEclipse: Learning to Program in Eclipse

Robert Chatley
Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ
rbc@doc.ic.ac.uk

Thomas Timbul
Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ
tt101@doc.ic.ac.uk

ABSTRACT

A fundamental part of a Computer Science degree is learning to program. Rather than starting students on a full commercial language, we favour using a dedicated "teaching language" to introduce programming concepts.

At the same time, we want to introduce students to popular tools that assist in the software development process. However, up until now our teaching language, Kenya, has not been supported by professional IDEs. Therefore, we have been unable to progress smoothly from first principles to the state of the art within one environment.

We present work that integrates the Kenya language into the Eclipse environment. Students can now become familiar with the major features of a professional IDE while learning to program, and experience a smooth transition to commercial languages within the same environment.

One of the hardest things to teach students is good programming style. Compilers reveal syntactic and type errors, but do not analyse style. We have harnessed as-you-type code checking, as seen in Eclipse's Java development tools, to provide advice on program style as well as correctness.

Categories and Subject Descriptors

D2.3 [**Coding Tools and Techniques**]: Structured Programming

General Terms

Design, Languages

Keywords

programming, education, style checking, Eclipse

1. INTRODUCTION

Teaching a programming language such as Java to university students is a challenging task. Java is a large and complex language with many features. This is also true of other "commercial" languages such as C# or C++. These languages are powerful, but contain many concepts that are too complicated to explain at the beginning of a programming course.

The Kenya teaching language [2] solves this by presenting students with a less complex selection of features and abstracting away from concepts that a novice programmer might not understand.

While knowledge of a programming language is important, it cannot be denied that in modern software development, productivity and efficiency is increased by the use of powerful development tools. Particularly common are integrated development environments (IDEs) that allow the programmer to quickly manipluate code and to automate repetitive tasks.

To be able to program effectively, it is desirable that students learn not only programming concepts, but also to make good use of the tools provided by professional IDEs. However, many IDEs have a steep learning curve and even experienced programmers may take some time before they can use a tool to its full advantage.

We present a plugin for Eclipse that enables users to program in Kenya using the Eclipse IDE, and to make use of a subset of the features that are currently available in professional tools for Java development. This allows them to work more effectively when moving on to the full Java language and the rich set of IDE features that exist for it.

Tools are designed to aid and support programmers, helping them to produce high quality software. As we are introducing tools to novice programmers, we aim to harness these tools to support the particular needs of a beginner. In teaching programming to students in our university, we have noticed that the main problems students have are with learning good programming style. In our Kenya plugin for Eclipse, we have developed a style critic that detects stylistic problems in students' code, suggesting improvements as they work.

In the remainder of this paper we give some background on the Kenya language and the Eclipse development environment, in Section 3 present details of our support for detecting stylistic problems, and in Section 4 describe the implementation of the tool. We conclude by discussing related work and future directions.

2. BACKGROUND

2.1 Kenya

In order to become good programmers, students need to be able to concentrate on understanding the problems that they are solving, and apply appropriate techniques and algorithms. This can be impeded by the need to understand the syntax of a complex programming language. Languages specifically designed for teaching programming provide simple syntax, but students are often not happy learning a language which they do not see as being real world.

For the past few years we at Imperial College London have had success using the teaching language Kenya [1, 2], which we devel-

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

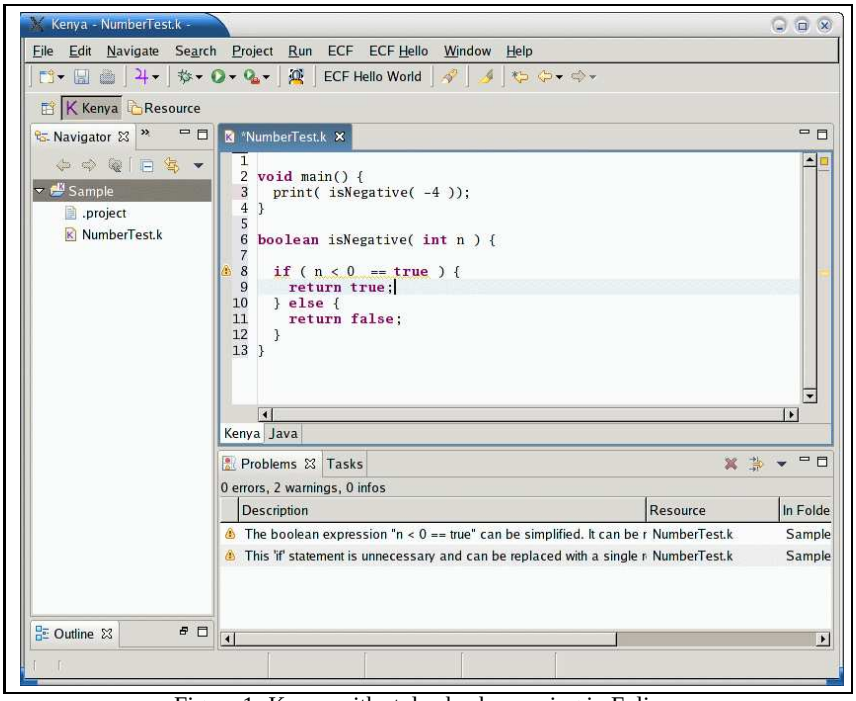

Figure 1: Kenya with style checks running in Eclipse

oped (and is now used in a number of universities around world, particularly in Italy and South America). The Kenya language is similar to Java but focuses on an imperative subset of the full language [4]. Kenya programs can be automatically translated to equivalent Java programs so that students can see what they would write if they were using full Java. This makes Kenya a useful stepping stone towards learning Java.

It has been well documented that language complexity impedes learning to program [3, 5]. Most alternative approaches to minimising this complexity involve providing a skeleton into which students plug their code. Our experience with such techniques is that, as students are only required to write fragments of code, they leave some students not knowing how to construct a complete program. Our solution to the problem is to have designed a complete but small language that enables students to gain confidence because they write entire programs.

2.2 Eclipse

An important part of modern software development is the use of tools. The popularity of IDEs like Eclipse has shown the enhancements in productivity that developers can experience by having good tool support. An IDE like Eclipse with the Java Development Tools installed provides a large amount of functionality. Unfortunately, it can be difficult even for an experienced programmer to learn how to use powerful tools effectively, even although they may be fluent in the programming language. This is especially true if they have previously been using basic command-line tools.

As teaching languages are not typically used beyond the university environment, they are often not supported by powerful tools. For example, source must often be edited in a standalone text editor, which will probably not provide features such as auto-completion, integrated help or debugging facilities. The lack of tools for teaching languages can lead to students not realising the benefit that can be gained by using the tools that are available for developing in more fully-fledged languages.

Following the approach that we have taken with programming languages, using a simplified version of the language as an introduction to the full language, we would like to introduce the basics of Eclipse early in the degree course. Students will then become familiar with the patterns and idioms of using the IDE while working with our teaching language before moving on to using its full power when working on larger projects, easing the learning curve.

We have integrated the Kenya language as a perspective in Eclipse. A perspective determines the visible actions and editor views that are available on screen. By selecting a small set of tools and ideas from the Java Development Tools (for example the notion of a project, the package explorer, the tasks/problems list) we allow students to learn a simple language in a simple but supportive environment. We advocate introducing Eclipse terminology early, so the 'small' programming language feels more real. Then as students progress from Kenya to Java they will be able to take a similar small step up to the full Eclipse IDE for Java.

3. STYLE CHECKING

We have found that many of the comments that a human tutor makes about student programs refer to coding style. We try to teach students that their programs must be read as well as written, because it is likely that other programmers will one day have to maintain their code.

Therefore, a tutor may consider a program to be 'badly written' even if its algorithm is efficient and it produces the correct results at run-time. The tutor could think this for many reasons: logical flow is unclear, literal values are scattered throughout the code rather than constants, or method definitions are overlong need refactoring.

While an IDE like Eclipse provides as-you-type detection of compilation errors, we would like it also to advise on stylistic issues. Thus, in a teaching environment, it would play the role of a human tutor, providing continuous support and advice.

For instance, stylistic comments often refer to redundancy in

Boolean expressions. The two code fragments below compile successfully and execute with the same results. We would like students to be encouraged to write the second as it is clearer, more succinct and more efficient.

```
boolean b;
...
if ( b == true ) {

  // do something
}
```

would be better written as:

```
boolean b;
...
if ( b ) {

  // do something
}
```

Fowler and Beck identify a number of structures that may appear in code that are referred to as 'code smells' [6]. These include: duplicated code, over long methods and long parameter lists. There has been some previous work on automatically detecting these features in program code [9, 10, 12]

We have built features into our tool that allow patterns of 'bad style' in programs to be detected and reported as code is written within Eclipse, in the same way that data and control flow analysis is performed. We concentrate on the good programming idioms encouraged at Imperial College having taught thousands of programmers over many years. The tool gives students continuous feedback on the quality of the code they are writing, increasing the degree to which the IDE can support the learning process.

3.1 Detection

As the user types their program source, our tool analyses the code to detect a number of common patterns of bad programming style. Whenever the user stops typing for a few seconds, the current code is parsed and checked for syntactic, type and style errors.

Before we detect stylistic errors, we require that the written code is free from syntactic and type errors. This helps to ensure reliable detection as we can process the abstract syntax tree (AST) rather than the plain source text. It is in any case sensible to have the student fix any more serious (compilation) errors before attempting to correct their style. Apparent style problems may have resulted from other errors.

Once the code can be parsed and compiled, it is analysed making use of the AST representation. The advantage of using the AST over using, say, regular expressions is that we reduce the numbers of false positives resulting from, for example, analysing code that is in fact commented out. Working with the AST also gives flexibility to do more sophisticated analysis.

We currently check for a number of different style errors including: over-complicated boolean expressions; use of "magic" strings or numbers, which should be constants; missing breaks or default cases in switch statements; over-long functions.

When one of these patterns is recognised, it is underlined in yellow in the editor, and a warning triangle is placed next to the relevant line (see Figure 1). Clicking on this displays a message explaining the problem.

Each style check is implemented as a visitor that traverses the AST looking for particular patterns of nodes. For example, when detecting redundancy in boolean expressions, all boolean expressions are located. A more finely grained visitor then traverses each expression to check for possible reduction. Any possible reductions are then offered as a suggestion to the user, which, if accepted, replaces that part of the tree.

3.2 Suggestion and Correction

Eclipse's Java Development Tools support the notion of a quick-fix. When an error is detected, Eclipse will often be able to suggest a possible fix or fixes that will alleviate the problem. Selecting a fix causes Eclipse to repair the code automatically. We have used this idiom to allow automatic correction of some stylistic problems. Here is an example. Consider the following very simple program that tests whether a number is negative. This is perfectly valid code, but an experienced programmer will see ways that the `isNegative()` function can be improved in terms of style.

```
void main() {
  print( isNegative( -4 ));
}

boolean isNegative( int n ) {

  if ( n < 0 == true ) {
    return true;
  } else {
    return false;
  }
}
```

If we write this program in our Kenya/Eclipse tool, the line beginning `if ( n < 0` is underlined in yellow. We get a style warning saying that our boolean expression contains redundancy which can be reduced. Applying the tool's suggested quick-fix changes the function to the following:

```
boolean isNegative( int n ) {

  if ( n < 0 ) {
    return true;
  } else {
    return false;
  }
}
```

We now get another warning, saying that the if statement can be replaced by a single return statement, as if the condition is true, we return true, if it is false, we return false. Applying the suggestion yields the following, much more elegant, code.

```
boolean isNegative( int n ) {

  return n < 0;
}
```

It is important to note that at each stage it is the programmer's responsibility to choose to apply a quick-fix. We want the novice programmer to go through the mental processes of having their mistake pointed out to them, and then fixing it (even if they do not change the code manually), so that they learn from their mistakes. The mistakes that are highlighted in the above example are typical of the sort of errors that we see time and again in solutions to exercises produced by beginners. The tool we have developed provides continuous feedback and support for identifying problems and suggesting improvements in students' code as they write it, rather than having to wait for feedback from a marker days or weeks after the exercise is submitted.

4. ARCHITECTURE

We have implemented a plugin for Eclipse that provides a Kenya perspective. This borrows idioms from the Java Development Tools provided with Eclipse, but trims down the workbench to provide a minimal working environment. To ensure consistency between the Kenya and Java perspectives, we have hooked our error checking and style guidance routines into the problems view, and used the

run and debug options in the toolbar to interface with the Kenya interpreter.

In developing the Eclipse plugin, we strived to maintain the ability to run Kenya as a standalone application based on the same code. This has been achieved by separating the Eclipse features from the core Kenya features, and led to a modularised architecture.

The initial requirements that had to be met in the implementation of the style guidance module were: extensibility - easy addition or removal of style checks; configurability - disabling individual style checks; independence - Kenya/Eclipse should work without the style module; efficiency - processing must be done in the background, not block the GUI; assistance - allowing users to apply automatic correction if possible or applicable.

The easiest way to achieve extensibility and independence was to make use of the ability to declare extension points in Eclipse plugins. This means that in future extra style checks could be added as plugins to our plugin, without having to recompile Kenya/Eclipse.

For each style pattern there is one StyleChecker (derived from a common base class). These are responsible for detecting errors and for defining a set of possible automated solutions (including explanation). By making use of the editor architecture, the problem location is highlighted using so-called markers and annotations. At the same time, the solutions are cached to allow fast access.

Developers who want to write their own StyleChecker only have to write code for detecting the problems and declare the solution (if any). All remaining issues (to do with markers, annotations, registering solutions) are functions maintained by the base class and the framework. This allows for little overhead when considering the addition of a new check for a different problem.

5. RELATED WORK

Other teaching languages that address the problems of teaching full Java as in introductory language include BlueJ [3] and JJ [5]. BlueJ has an interactive programming environment, where programs are constructed by interactively creating Java objects rather than explicitly writing code. This can help with the understanding of object-oriented principles, but may, as discussed earlier, lead to students not being able to code complete programs from scratch. JJ, developed at Caltech, provides an online environment for learning Java. Neither of these approaches introduce students to tools that are used in industry.

GILD [11] is an Eclipse plugin from the University of Victoria in California. The tool is aimed directly at students learning programming with Java. It has integrated support for marking lab work. GILD's emphasis seems to be on learning in a classroom environment. There seems not to be a great deal of support for helping individuals learn programming. Some advice on erroneous code is offered, but with no suggestions as to how to correct it. Similarly Class Compass [7] aims to improve student-teacher communication using internet tools, rather than providing explicit programming help.

There have been a number of projects aiming to automatically assess the quality of code, including calculating metrics on code [12], trying to detect "anti-patterns" [9], or violation of coding conventions [8]. However, we do not know of any other work that tries to address fundamental issues of style, or provide the immediate feedback that our tool supports.

6. CONCLUSIONS

We have presented a plugin for Eclipse to support the Kenya teaching language. By integrating the language that we use for teaching introductory imperative programming with an industry standard development environment, we can introduce students to tools at the same time as teaching programming concepts.

Building on the basic editor and code management functions of the IDE, we have built a system for analysing and suggesting improvements to a student's program in terms of programming style. This allows some of the support offered by human programming tutors, a comparatively scarce resource, to be given by the machine.

In future we would like to build on the range of stylistic checks that we can perform, and perhaps provide a method for users to customise the checks, as different people do prefer different styles. We would also like to adapt the style checking engine to work with full Java (or C# or C++) code, rather than just Kenya.

The tool will be available for download from summer 2005 from http://www.doc.ic.ac.uk/kenya.

7. ACKNOWLEDGMENTS

We would like to acknowledge Tristan Allwood and Matthew Sackman, who reimplemented the Kenya system in 2004. Their work made the integration with Eclipse a lot easier. We would also like to thank Susan Eisenbach, who teaches the introductory programming course at Imperial for her insights into common stylistic mistakes. For financial support, we would like to acknowledge IBM under the Eclipse Innovation Award scheme.

8. REFERENCES

[1] T. Allwood, R. Chatley, and M. Sackman. Kenya. Technical report, Imperial College London, http://www.doc.ic.ac.uk/kenya, 2004.

[2] R. Chatley. Java for Beginners. Technical report, Imperial College London, http://chatley.com/kenya/thesis, 2001.

[3] David J. Barnes and Michael Kolling. *Objects First with Java, A Practical Introduction using BlueJ, 2nd ed.* Prentice Hall / Pearson Education, 2004.

[4] S. Eisenbach. Kenya notes. Technical report, Imperial College London, http://www.doc.ic.ac.uk/˜ sue/121/index.html, 2004.

[5] D. Epstein and J. Motil. JJ. Technical report, Caltech, http://www.publicstaticvoidmain.org/, 2004.

[6] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] M. Mervis. Class Compass. Technical report, Mervis Learning Designs, http://www.classcompass.com, 2005.

[8] H. Oak. Three tools that make Java code review painless and effective. Technical report, Builder.com, http://builder.com.com/5100-6370-5031836.html, 2003.

[9] Scott Grant and James R. Cordy. Automated Code Smell Detection and Refactoring by Source Transformation. In *IEEE Working Conference on Reverse Engineering*, 2003.

[10] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *CSMR*, pages 30–38, 2001.

[11] M.-A. Storey, D. Damian, J. Michaud, D. Myers, M. Mindel, D. German, M. Sanseverino, and E. Hargreaves. Improving the usability of eclipse for novice programmers. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 35–39, New York, NY, USA, 2003. ACM Press.

[12] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, Oct. 2002.