Java-Meets Eclipse - An IDE for Teaching Java Following the Object-later Approach

Lorenzo Bettini, Pierluigi Crescenzi

Java--Meets Eclipse
An IDE for Teaching Java Following the Object-later Approach

Lorenzo Bettini[1,*] and Pierluigi Crescenzi[2,†]
[1]*Dipartimento di Informatica, University of Turin, Turin, Italy*
[2]*Dipartimento di Ingegneria dell'Informazione, University of Florence, Florence, Italy*

Keywords: DSL, Java, IDE, Eclipse, Xtext, EMF.

Abstract: In this paper, we introduce a new Eclipse-based IDE for teaching Java following the object-later approach. In particular, this IDE allows the programmer to write code in Java--, a smaller version of the Java language that does not include object-oriented features. For the implementation of this language we used Xtext, an Eclipse framework for implementing Domain Specific Languages; besides the compiler mechanisms, Xtext also allows to easily implement all the IDE tooling mechanisms in Eclipse. By using Xtext we were able to provide an implementation of Java-- with all the powerful features available when using an IDE like Eclipse (including debugging, automatic building, and project wizards). With our implementation, it is also straightforward to create self-assessment exercises for students, which are integrated in Eclipse and JUnit.

1 INTRODUCTION

Java-- is an application that allows undergraduate students to learn programming by using a smaller version of Java without object-oriented features. This way, students can focus on the basic programming concepts without being distracted by complex constructs. The original implementation of Java-- comes without any IDE; it provides a GUI, but without the advanced tooling mechanisms that are typical of an IDE like Eclipse.

In this paper, we introduce a new implementation of Java-- that includes a full-featured Eclipse-based IDE. Our IDE provides an Eclipse based editor with syntax highlighting, navigation, code completion and error markers, not to mention automatic integrated building and debugging. This way, the students will immediately become familiar with Eclipse.

By using our implementation of the IDE for Java--, it is straightforward for the teachers to create exercise projects that students can use for self-assessment. The teacher can rely on existing frameworks such as JUnit, which is already integrated in Eclipse.

*Author partially supported by MIUR (proj. CINA), Ateneo/CSP (proj. SALT), and ICT COST Action IC1201 BETTY.

†Author partially supported by MIUR under PRIN 2012C4E3KT national research project "AMANDA Algorithmics for MAssive and Networked DAta".

The Java-- IDE is available as an open source project at `https://github.com/LorenzoBettini/javamm`.
We also provide an Eclipse update site and preconfigured Eclipse distributions with Java-- installed, for several architectures.

In the following we will use the term Java-- for denoting both the smaller version of the Java language and the application (with or without IDE) for writing code in this language.

1.1 Educational Motivation

Every undergraduate program in computer science introduces its first-year students to programming. Even if choosing the most suitable First Programming Language (in short, FPL) to be taught is still an active debate topic, there is a general agreement on the fact that such a choice has a significant influence on students' learning performances (see, for example, (Koulouri et al., 2014)) and, clearly, on the development of students' programming abilities, both in terms of the programming style and in terms of the coding techniques they will adopt during their professional life. Some common factors that are taken into account while choosing the FPL might be its simplicity, its industry relevance, its programming paradigm, and the available development tools (see, for example, (Pears et al., 2007; Mason and Cooper, 2014)).

As shown in (Farooq et al., 2014), in 2011 C++,

Java, and Python were the most frequently used FPLs, with Python gaining more and more popularity (see, for example, (Leping et al., 2009)). This latter trend has been confirmed in the last three years, as witnessed, for example, in (Guo, 2014), where the top 39 computer science departments (as ranked by U.S. News in 2014) have been considered, and, for each department, the CS0 and CS1 courses have been analyzed in order to determine which language was used. In particular, 8 (respectively, 5) of the top 10 departments, and 27 (respectively, 22) of the top 39 departments teach Python (respectively, Java) in introductory computer science courses. Python, Java, and C++ turn out to be also among the most "lucrative" programming languages, as stated in (Nisen, 2014), where the author analyses the data compiled (starting from thousands of American job ads) by Burning Glass with Brookings Institution economist Jonathan Rothwell, in order to determine which language might get the worker the best salary.

A framework to evaluate several existing programming languages, for their suitability as an appropriate FPL, has been proposed in (Farooq et al., 2014). By applying this framework to Ada, C, C++, C#, Fortran, Java, Modula-2, Pascal, and Python, the authors show that Java obtains the overall highest score and, thus, conclude that it is the most suitable programming language (followed by Python and Ada). Even by considering the "demand in industry" and "easy transition" parameter more important than the other parameters, Java turns out to be the language with highest score (followed by Python and C#).

As observed in (Koulouri et al., 2014), even if Java has been widely used as a FPL, its complexity "may be overwhelming for learners". In particular, one of the main complexity sources of this language is the fact that it is "heavily coupled with object-oriented concepts", thus making more difficult to implement an object-later strategy. A typical example of such complexity is the implementation of the well-known "hello, world" program, which would require, in addition to the print statement, the definition of a class, containing a `main` method (with its "strange" syntax, both in terms of modifiers and in terms of parameters).

Several tools have been proposed in order to deal with the complexity of Java for novice programmers (see Section 4). Some of them were designed in order to teach basic programming concepts (such as primitive data types, arrays, control structures, methods and recursion), before exposing the students to classes and objects. An example of such tools is JOSH (Diehl, 2003), which is a Java interpreter designed to ease teaching Java to beginners. With JOSH programmers can interactively evaluate simple expressions, execute program statements, define variables and methods, and invoke methods. Inspired by JOSH, which was based on a command line interaction, the Java-- application (see Section 2) has been successively developed and used at the University of Florence, along with an Italian text book (Crescenzi, 2015). This application allows the user to perform the same tasks of JOSH through a simple Graphical User Interface (GUI), which contains three tabbed panels that allow the user to edit the code, see any mistakes that the Java compiler detected, and look at the standard output (Cecchi et al., 2003). Successively, Java-- has been extended with a self-assessment module, which allows the user to implement one or more methods solving simple problems, immediately verifying the correctness of the proposed solution (Bettini et al., 2004; Crescenzi et al., 2006).

As already said, the development of JOSH and of Java-- referred to the "structured programming before object oriented programming" teaching paradigm, as described in (Gibbons, 1998). According to this paradigm the first part of a CS1 course should be taught in order to develop skills with the usual low-level procedural mechanism, that will allow students to gradually build up programs from primitive types and basic control structures. Only at a later stage students should implement object features and test objects, and, thus, be exposed to the conceptual load related to the new abstract data type issues. Moreover, as stated in (Lewis, 2000), it is worth observing that the object oriented approach does not abandon typical concepts of a procedural approach: indeed, it augments and strengthens them, since implementing an object almost always requires a good knowledge of structured programming. Finally, not having objects in a language does not necessarily decrease its educational power and usability; for example, Processing (Reas and Fry, 2014), a programming language and a development environment initially created to teach programming, allows the programmer to easily do professional computer graphic and animations without object-oriented constructs.

Even though Java-- allows the programmer to interact with a GUI, that includes some features, such as syntax coloring, typical of an Integrated Development Environment (in short, IDE), this application cannot be considered an IDE, since it lacks many other features, such as compiler and debugger integration, build automation, code completion, and easy navigation to definitions. On the other hand, all existing Java IDEs (such as Eclipse and Netbeans) require the programmer to deal, since the very beginning, with the object-oriented programming paradigm. That is why,

in this paper, we propose a new Eclipse-based IDE (see Section 3), that has the same features of Java--, and, thus, allows the programmer to experiment with the basic programming concepts (without necessarily knowing any notion of object-oriented programming), and that, at the same time, has all the advantages of being an IDE (see Section 3.1). This Java-- IDE may also ease the transition task which arises when the student is asked to switch from the Java-- environment to a more sophisticated IDE (such as Eclipse), since the GUI to be used will be exactly the same.

1.2 Technology Used

Developing a compiler and an IDE for a programming language is usually time consuming even when relying on a framework like Eclipse, which already provides typical IDE mechanisms. After implementing the compiler components like the parser and the validation, we then need to connect the compiler to the IDE components and this requires lot of manual programming. Xtext (Itemis, 2015; Bettini, 2013) is a modern language workbench (such as MPS (Voelter, 2011) and Spoofax (Kats and Visser, 2010)) that solves all the above issues in a uniform way: starting from a grammar definition Xtext generates not only a parser and an abstract syntax tree, but also all the typical Eclipse-based tooling features (e.g., editor with syntax highlighting, code completion and static error markers). Xtext comes with good defaults for all the above language implementation mechanisms and for the Eclipse integration; however, the language developer can customize all such mechanisms in a straightforward way. Thus, Xtext makes language implementation and its integration into Eclipse really easy (Eysholdt and Behrens, 2010).

For all the above reasons, we chose Xtext for the implementation of the Eclipse IDE presented in this paper. By relying on Xtext, our implementation provides all the typical Eclipse features: automatic building, error reporting, outline view, navigation to declarations (e.g., variables, parameters, methods) and debugging, just to mention the main ones, mimicking the same features of Eclipse JDT (Java Development Tools). This means that Java-- will make the transition to the complete Java language and its Eclipse integration easier for the students who already got familiar with our Java-- IDE.

Since Xtext uses EMF, the Eclipse Modeling Framework (Steinberg et al., 2008), for storing and representing programs, another important consequence of using Xtext is that we will be able to use all the modeling tools of the EMF ecosystem (Gronback, 2009), including graphical modeling frameworks. With that respect, integration of graphical views and Xtext implementations have already been proved possible (see, e.g., (Koehnlein, 2014; Brun, 2014)).

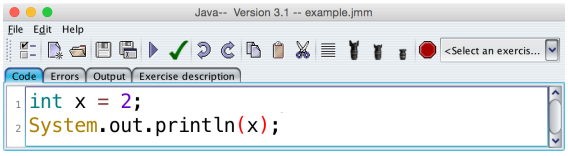

Figure 1: Writing a simple program with the original Java-- application.

2 THE ORIGINAL JAVA-- APPLICATION

In this section, we recall the main features of the original implementation of Java--. As stated in the introduction, the goal of this tool is to allow the user to focus on the basic programming concepts, without encumbering the novice student with unnecessary complex constructs. Indeed, the user may write Java code outside of any method body as shown in Fig. 1.[3] In order to appreciate the advantage of using Java--, the code shown in the figure can be compared with the code that a student should have written if Java-- was not used, that is,

```
public class Example {
  public static void main(String[] a) {
    int x = 2;
    System.out.println(x);
  }
}
```

Apart from the fact that the student is exposed since the very beginning with the object-oriented specific syntax of Java, as stated in (Westfall, 2001) the above code can even be "harmful to development of object-thinking", since "it communicates virtually nothing about the concept of user-created objects". On the contrary, the code shown in Fig. 1 turns out to be very similar to the corresponding Python code, that is,

```
x = 2
print x
```

[3]As it is shown in the figure, the student is required to make a leap of faith concerning the use of the static methods to print to the standard output (such as `System.out.println()`). Although this could have been avoided by implementing a `print()` method that invokes the corresponding Java method, we preferred to ask the students to use the standard Java methods and to profess their faith in them, rather than give them a solution that is not pure Java, and which could have confused them later on.

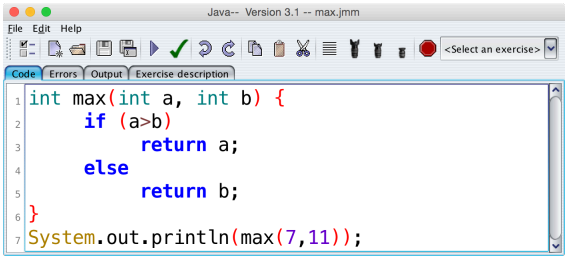

Figure 2: Defining and invoking methods in Java--.

Clearly, Java-- allows the programmer to define and invoke methods, as shown in Fig. 2. Once again, it is interesting to compare the code shown in the figure with the corresponding Python code defining the `max` method, that is,

```
def max(a,b):
  if a>b:
    return a
  else:
    return b
```

When Java-- is asked to execute a code, the tool generates a temporary class where it defines the `main` method and copies all the methods specified by the user. Successively, the temporary class is compiled and executed: the compilation errors are shown in the `Errors` pane, while the output and/or the execution error are shown in the `output` pane. The original implementation of Java-- also includes a self-assessment module, that allows the student to write methods solving simple programming problems. The behavior of this module is very similar to other tools available on the web, such as, for example, the CodingBat tool (Parlante, 2011). The distribution of Java-- already includes dozens of pre-defined exercises, but new exercises can be easily added by teachers.

3 THE JAVA-- IDE

We now briefly describe the new IDE, by first emphasizing the advantages of using an IDE, by then explaining the technology which has been used to develop the IDE, and by finally showing some examples of the IDE interface.

We would like to stress that the aim of Java-- as a programming language is to target algorithmic aspects of programming, so that students can concentrate on implementing algorithms without being distracted by OO features. Thus, Java-- does target other programming contexts such as GUI programming. However, as described in more details in the conclusions (Section 5), Java-- can access any existing Java type, such as container classes.

3.1 On the Value of an IDE

Although an IDE is not a strict requirement to develop applications, it surely helps programmers to increase productivity with features like syntax coloring in the editor, compiler and debugger integration, build automation, code completion and easy navigation to definitions, just to mention a few. In an agile (Martin, 2003) and test-driven context (Beck, 2003) the features of an IDE like Eclipse become an essential requirement. Indeed, languages such as Smalltalk have been tightly coupled with an IDE from the beginning (Goldberg, 1984).

The ability to see the program colored and formatted with different visual styles (e.g., comments, keywords, strings, etc.) gives an immediate feedback concerning the syntactic correctness of the program. Moreover, colors and fonts help the programmer to see the structure of the program directly, making it easier to visually separate the parts of the program.

The programming cycle consisting of writing a program with a text editor, saving it, switching to the command line, running the compiler, and, in case of errors, going back to the text editor is surely not productive. The programmer should not realize about errors too late: the IDE should continuously check the program in the background while the programmer is typing in the editor, even if the current file has not been saved yet. The longer it takes to realize that there is an error, the higher the cost in terms of time and mental effort to correct it. For example, the Eclipse Java plugin highlights the parts of the program with errors directly in the editor: it underlines in red only the parts that actually contain the errors; it also puts error markers (with an explicit message) on the left of the editor in correspondence to the lines with errors, and fills the Problem view with all these errors. The programmer will be able to easily spot the parts of the program that need to be fixed.

With that respect, in our experience as teachers, we noted that most of the students that fail programming exams are not able to fix compilation errors since they do not use an IDE: they tend to write the whole program and then try to compile it; when they get lots of compilation errors, they are not able to understand how to fix them.

3.2 Software Technology: Xtext and Xbase

As anticipated in Section 1.2, we chose Xtext as the framework for implementing the Eclipse IDE for Java--. In this section we will provide some implementation details, and further motivate our choice.

Xtext allows to start from a library grammar for the implementation of a language, so that one does not have to define the grammar rules for typical language elements such as strings, integers, comments, etc. One of such grammars to start from is Xbase (Efftinge et al., 2012), an extendable and reusable expression language. Xbase integrates tightly with the Java platform and JDT (Eclipse Java development tools). In particular, Xbase reuses the Java type system without modifications, thus, when a language uses Xbase it can automatically and transparently access any Java type. For these reasons, Xbase makes it straightforward to develop languages with Xtext that need to interoperate with Java, its type system, all the Java libraries and Eclipse JDT.

One of the goals of Xbase is to provide an expression language which is not strictly Java, but a version with less "syntactic noise". For instance, types are not required in Xbase expressions if they can be inferred from the context (e.g., the type of a declared variable can be inferred from the initialization expression). Moreover, in Xbase everything is an expression, even "if", "while" and "switch" statements. Thus, Xbase aims at providing a readable Java-like code which is smaller and more compact like dynamically typed languages, while retaining all the advantages of statically typed language checks.

Indeed, the above advanced and clean features of the Xbase expression language have been a drawback for us, since our main goal is to have the Java syntax for expressions and statements (and Java statements should not be used as expressions). However, we leveraged the complete customizability of Xtext and modified some of the Xbase grammar rules to comply with Java syntax[4]. Moreover, we had to customize some other parts of the Xbase implementation. This required some work, but the advantages of reusing the existing Xbase implementation for the Java type system and all the Xbase IDE tooling payed back. Indeed, implementing Java-- from scratch with Xtext without Xbase would have surely been much harder, even without having to deal with Java object-oriented features. To the best of our knowledge, Java-- is the first project that customizes Xbase (i.e., its grammar, type system and code generator) in order to be able to deal with Java expression syntax.

Summarizing, our implementation provides all the typical Eclipse features: automatic building, error reporting, Outline view, navigation to declarations (e.g., variables, parameters, methods) and debugging, just to mention the main ones, mimicking the same features of Eclipse JDT. This means that Java-- will make the transition to the complete Java language and its Eclipse integration easier for the students that already got acquainted with our Java-- IDE.

Our compiler will then generate Java code, that will be compiled by a standard Java compiler. In Eclipse, this will take place transparently: saving a Java-- file in an Eclipse project will generate the corresponding Java code, and this will in turn trigger the compilation of the generated Java code by the Eclipse Java compiler.

3.3 The IDE Interface

In Figure 3 we show a screenshot of the Java-- Eclipse IDE (note the complete integration of our tooling with the Eclipse mechanisms, which are basically the same as Eclipse JDT). First of all, in the Eclipse project, the Java-- compiler automatically generates Java code into the source folder `src-gen`; such generation is integrated with the Eclipse building mechanisms: if a Java-- file is modified, re-generation is automatically triggered and if a Java-- file is removed, the corresponding generated Java file is automatically removed. Error markers are placed on the editor's left ruler, on the corresponding file in the "Project Explorer", and in the "Problems" view (note that warnings are generated as well, just like in Java, e.g., when a declared variable is not used). Moreover, the regions in the editor corresponding to the errors are underlined (e.g., for a type mismatch error like in the screenshot). The "Outline" view on the right reflects the one of Eclipse JDT. Code completion works as well; with that respect, note that the content assist mimics the one of Java: Javadoc comments, if present, are displayed as well.

The use of Xbase implies another important feature: when running the generated Java code in debugging mode in Eclipse, we can choose to debug directly the original Java-- code (it is always possible to switch between the generated Java code and the original code). In Figure 4 we show a debug session of a Java program generated by our Java-- compiler: we have set break points on the Java-- file, and the debugger automatically switches to the original Java-- code (note also the file names in the thread stack, the "Breakpoint" view and the "Variables" view). Indeed a well-known problem with implementations which generate Java code is that for debugging, the programmer has to debug the generated code which is usually quite different from the original program; our implementation does not have this drawback.

[4]When an Xtext language reuses an existing grammar, it does so with a mechanism called *grammar inheritance*, which has basically the same semantics of object-oriented class inheritance: a rule in your grammar can override a rule in the "parent" grammar.

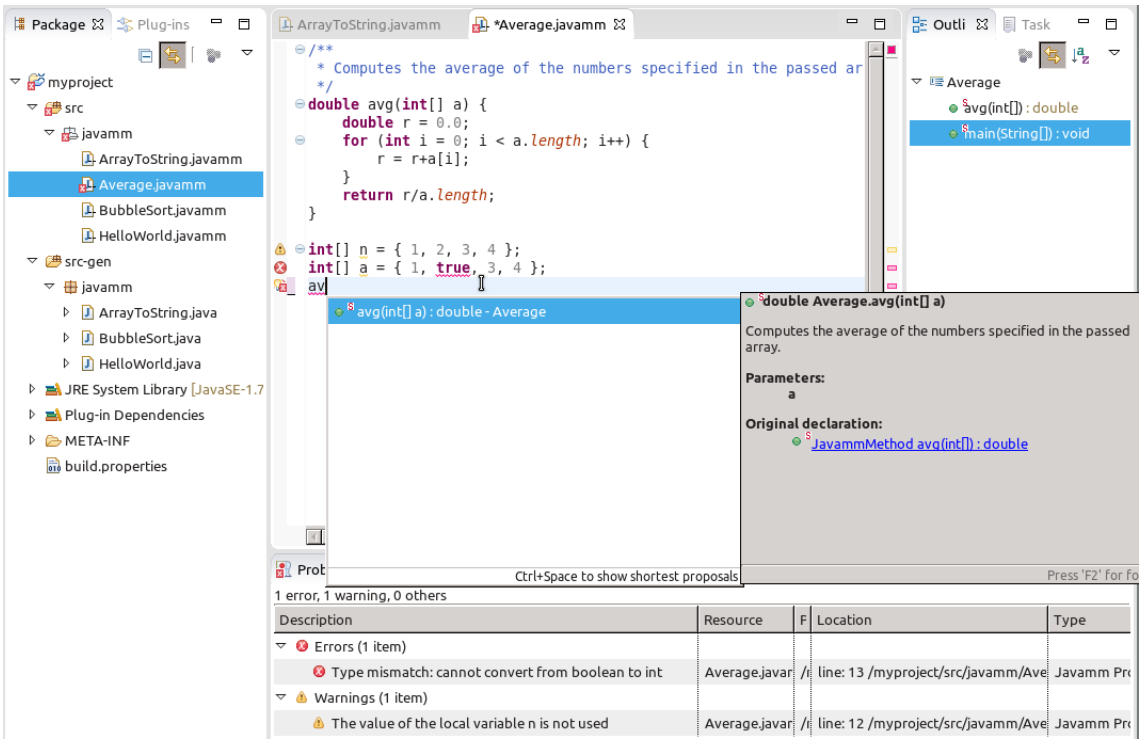

Figure 3: Java-- Eclipse IDE.

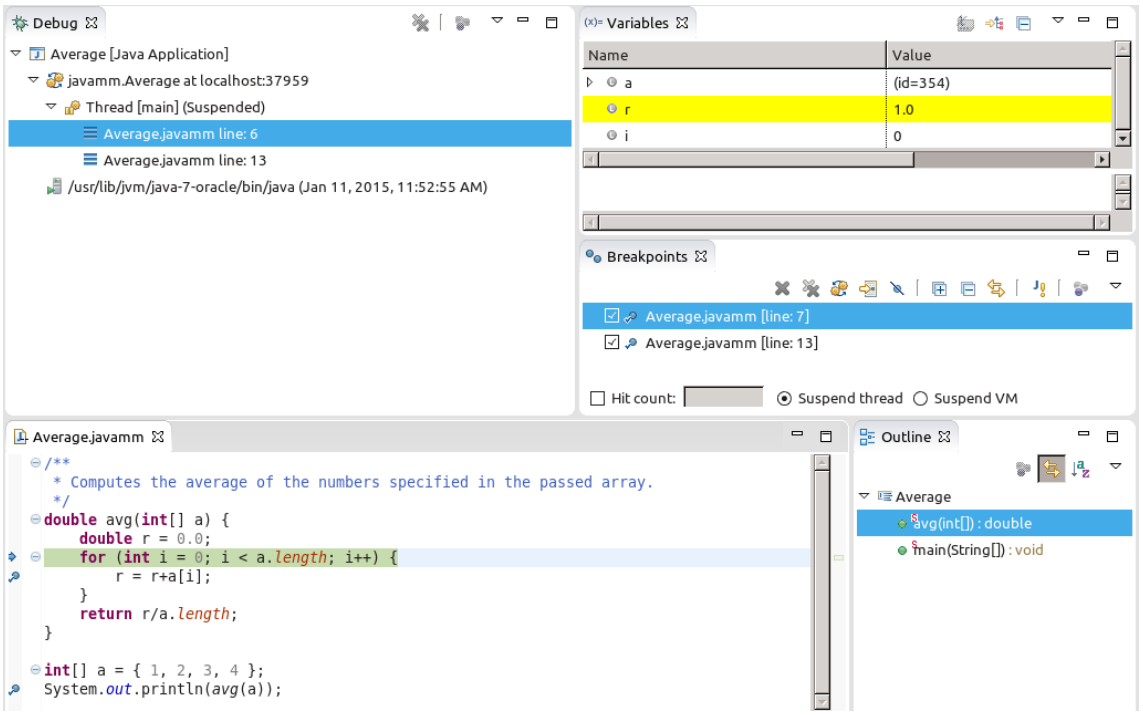

Figure 4: Debugging a Java-- program.

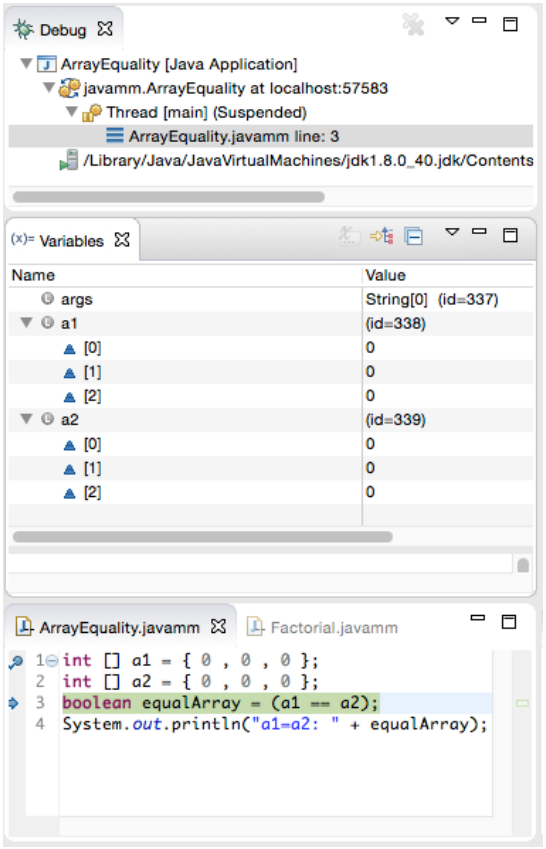

Figure 5: Using the debug for teaching array references.

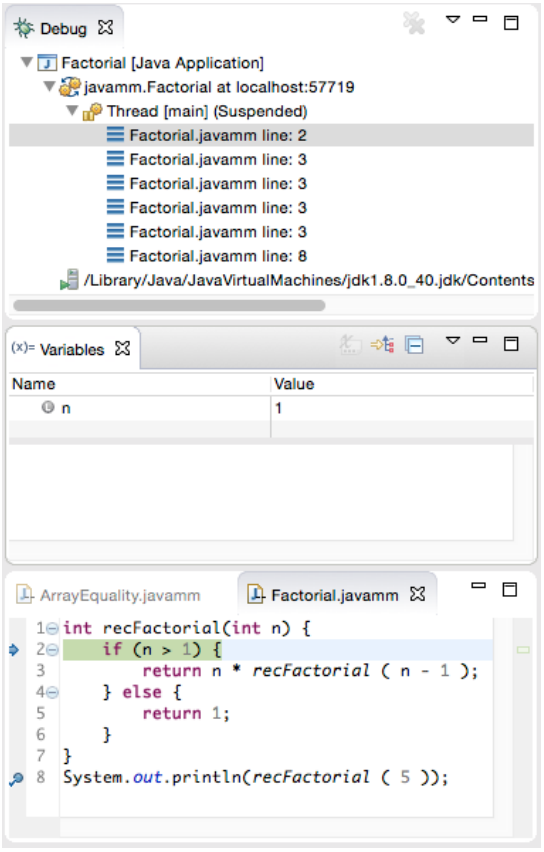

Figure 6: Using the debug for teaching the price of recursion.

In our opinion, the debugging feature can also be useful, during a lecture, in order to support explanations (usually done by using the whiteboard) of a program execution. For example, in Figure 5 it is shown how the debugger can be used in order to explain that an array variable is a reference variable and, hence, that the equality between two different array variables cannot be established by simply using the == operator. Indeed, in the "Variables" view it is explicitly shown that the two variables `a1` and `a2` are distinct, even though the elements of the two arrays they refer to are all the same. Another example of such a "pedagogical" utilization of the debugger is given in Figure 6. In this case, the "Debug" view explicitly shows the activation records corresponding to the five invocations of the `recFactorial` method, so that it is easier to explain to the students that recursion (and, in general. method invocation) has a "small" price to be paid, that, in certain cases, can be excessively high (just imagine the invocation of `recFactorial` with a large integer number as argument). Indeed, we think that this usage of the debugging mode of the Java-- Eclipse IDE has another advantage: besides supporting the teacher explanation, it may also allow the students to get used to the debugger before using it for developing their own programs.

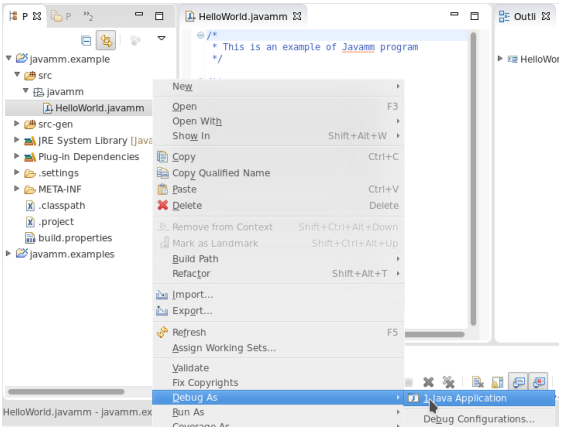

Figure 7: Context menu on Java-- sources for running and debugging.

Running or debugging the generated Java code can be done using context menus available directly on the original Java-- source, as shown in Figure 7.

The Java-- IDE also offers a project wizard to create an Eclipse project with the structure and require-

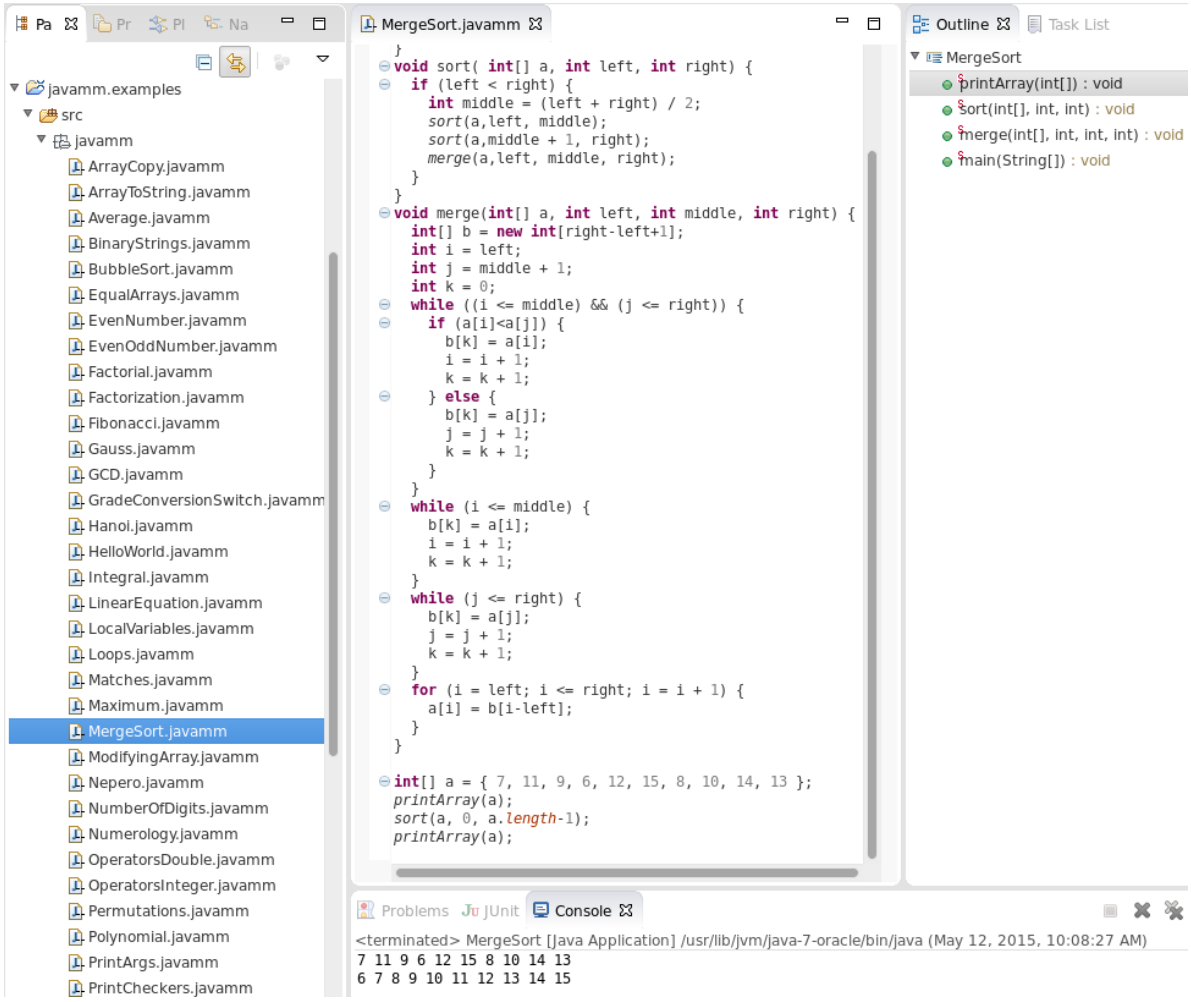

Figure 8: Some examples shipped with Java--.

ments to start editing, compiling and launching Java-- programs. We also provide a wizard to import a Java-- project with about 40 examples (Figure 8).

3.4 Self-assessment Tool

The original Java-- application had some mechanisms for self-assessment. In our implementation such mechanisms can be straightforwardly implemented by relying on JUnit and its integration in Eclipse. The basic idea is that the teacher provides the students with Java-- Eclipse projects with:

1. a Java-- file where the student implements his/her solution to the requested problem;
2. the solution of the teacher in binary form (so that the student cannot have a look at it);
3. a JUnit testcase that checks whether the output of the student's implementation corresponds to the teacher's implementation.

An example is shown in Figure 9. In this example the student must implement the `max` function in Java--. Note that the student has made a mistake in the implementation, since within the boolean expression of the `if` statement the code compares the value of the first parameter with the successor of the value of the second parameter. The JUnit testcase tests the student's implementation with random inputs using the teacher's implementation as the expected output. The student can then see the failed tests (in case of mistakes in the implementation). In our example, it is shown that when the first parameter is 3 and the second parameter is 2, the answer given by the student's code is 2 instead of 3, because of the wrong comparison performed in the control expression of the `if` statement.

Currently, the setup of such projects has to be done manually by the teacher; Eclipse makes such setup easy, but we are working on a more automatic mech-

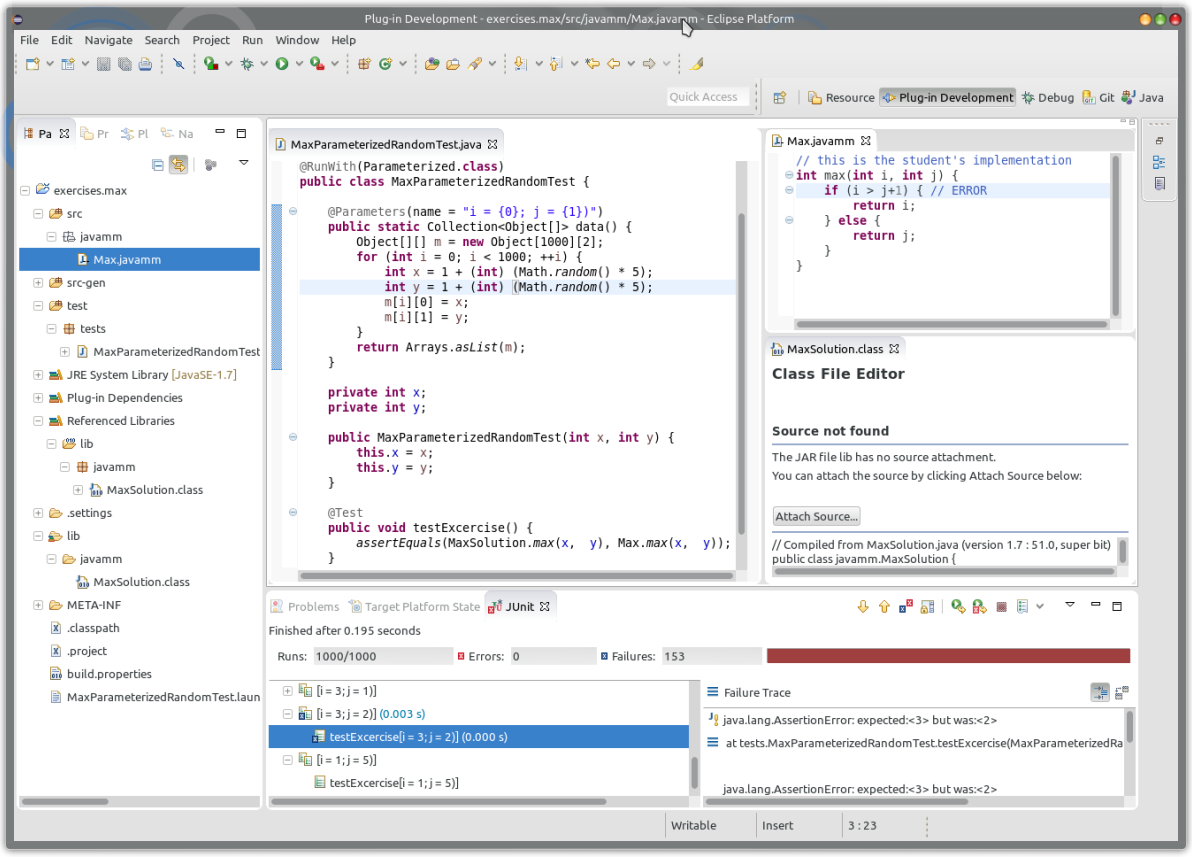

Figure 9: An example of exercise for Self-assessment.

anism for such setup (e.g., with wizards and Eclipse commands for exporting the teacher's solution in binary form into the student's project).

4 RELATED WORK

Some related work has already been discussed in the paper. In this section we will discuss some further related work, both concerning the educational context and the implementation technology.

4.1 Educational Related Work

An excellent survey of programming languages and environments for making programming accessible to beginners is contained in (Kelleher and Pausch, 2005). For what concerns Java, there is now a vast range of tools, which have been especially designed for educational purposes, in an attempt to create an environment that can help in teaching programming; we mention some of the popular ones. Alice (Dann et al., 2011) is an interactive programming environment that establishes an easy, intuitive relationship between program constructs and 3D graphics animations. BlueJ (Barnes and Kölling, 2011) is a teaching environment strictly linked to the development of object-oriented programs by means of a framework which is focused on objects (hence, applying a teaching paradigm opposite to the one followed by JOSH and Java--). JEliot 2000 (Levy et al., 2003) is a program animation system intended for teaching computer science especially to high school students but does not hide the object-oriented feature of the Java language. As far as we know, however, the Java-- IDE is the first tool which combines the pure procedural Java syntax learning with the utilization of all the powerful features of an IDE like Eclipse.

4.2 Implementation Framework Related Work

There are other tools for implementing DSLs and IDEs (we refer to (Pfeiffer and Pichler, 2008) for a

wider comparison). Tools like IMP (The IDE Meta-Tooling Platform) (Charles et al., 2009) and DLTK (Dynamic Languages Toolkit) only deal with IDE features, thus the compiler of the language has to be implemented separately, while Xtext unifies all the implementation phases. TCS (Textual Concrete Syntax) (Jouault et al., 2006) is similar to Xtext, but with the latter it is easier to describe the abstract and concrete syntax at once, and it is completely open to customization of every part of the generated IDE (besides, TCS seems to be no longer under active development). EMFText (Heidenreich et al., 2009), instead of deriving a metamodel from the grammar, does the opposite, i.e., the language to be implemented must be defined in an abstract way using an EMF metamodel.

In general, we chose Xtext since it is basically the main standard framework for implementing DSLs in the Eclipse ecosystem, it is continuously supported, and it has a wide community. Moreover, Xtext is continuously evolving, and the main forthcoming features will be the integration in other IDEs (mainly, IntelliJ), and the support for programming on the Web (i.e., an implementation with Xtext should be easily portable on the Web, allowing programming directly in a browser).

5 CONCLUSIONS

We have described a new IDE for teaching Java following the object-later approach. In particular, by using Xtext, this IDE combines the "structured programming before object oriented programming" teaching paradigm (already implemented in Java--) with all the powerful features available when using the Eclipse IDE. Observe that our IDE also allows to use all Java types, such as, e.g., the `String` and the Java collection classes; furthermore, the syntax for types already supports full Java generics, including wildcards. Finally, the syntax of Java-- expressions corresponds to the syntax of Java expressions, except for `this`, `super`, anonymous classes and Java 8 lambdas. This means that copying Java expressions into a Java-- program is allowed (of course, if the original Java expressions rely on specific OO features, like, e.g., a field reference on `this`, Java-- will issue a validation error). An example is shown in Figure 10: we use Java list classes, with generics (including wildcards) and imports; concerning imports, we support the automatic import statement insertion during content assist, and the typical "Organize Imports" menu. All these features turn out to be useful before migrating to the full Java syntax, in order to allow the students to familiarize with method invocation of already

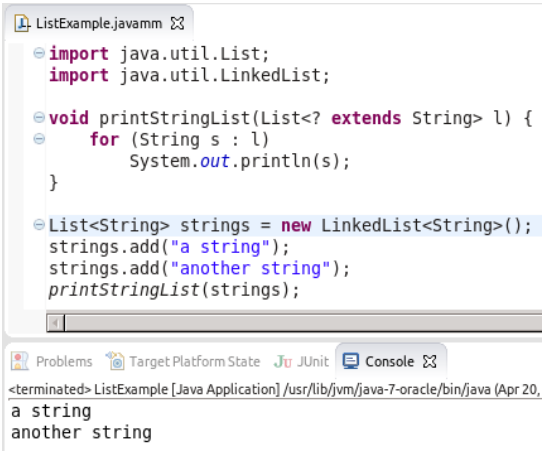

Figure 10: An example showing the use of library classes, generics and imports.

existing classes and with Java generics.

Adding Java 8 lambda expressions into Java-- should not be a problem: Xbase supports lambda expressions with its own syntax (which we have removed from the grammar), so we would just need to introduce in the Java-- Xtext grammar the syntax for Java 8 lambda expressions and then reuse Xbase's type system for lambda expressions. This is the subject of future work that would allow us to experiment with the functional programming approach as an intermediate step between object-later approach and the OO paradigm. The final step towards full Java would consist in adding anonymous classes and OO Java constructs such as interfaces and classes. This is feasible in Xtext, as proven by the programming language Xtend[5], but we think that it would not make much sense, since at that point we can directly switch to the full Java programming language and its Eclipse tooling.

As hinted in Section 3.4 we are working on making the creation of self-assessment exercises easier for the teacher. We are also planning to provide a dedicated DSL for such task, again implementing it with Xtext and Xbase. We will take inspiration from similar testing frameworks implemented in Xtext, such as, e.g., Xpect (Eysholdt, 2014) and Jnario (Benz and Engelmann, 2014).

From an experimental point of view, instead, we observe that, whenever a programmer is asked whether an IDE should be used, it is very likely that the answer would be an obvious one, that is, "yes". However, as stated in (MacDonald, 2014), it might be that, for novice programmers, it can be useful "to

[5]Xtend, `https://eclipse.org/xtend/`, is a Java dialect implemented with Xtext and Xbase.

be able to trace through the execution of the code by hand", even because, at this stage, the programs to be written will likely be pretty short. We conjecture that this is not the case. For this reason, we now plan to execute a controlled experiment in order to evaluate the efficacy of starting learning Java, by following the object-later approach, with and without an IDE (that is, by using the original Java-- application and by using the IDE described in this paper).

As hinted in Section 3.2, Java-- is the first project that customizes Xbase grammar, type system and code generator in order to be able to deal with Java expression syntax. We believe that such customizations should be easily factored out in a more general and reusable framework: this customized Xbase expression syntax for Java expressions can be reused in other DSLs that rely on Xbase for achieving the integration with the Java platform. Indeed, our customized syntax for expressions does not depend on any specific feature of Java--: the syntax for methods is simply built on top of the syntax for expressions. It will be interesting to perform such refactoring, to extract this part from Java-- and to experiment its use in other DSLs (one of such DSLs we plan to experiment with is the one described in (Bettini and Damiani, 2014)).

ACKNOWLEDGEMENTS

We are grateful to the reviewers for their insightful comments and suggestions for improving the presentation.

REFERENCES

Barnes, D. and Kölling, M. (2011). *Objects First with Java: A Practical Introduction Using BlueJ, 5/E*. Prentice Hall.

Beck, K. (2003). *Test Driven Development: By Example*. Addison-Wesley.

Benz, S. and Engelmann, B. (2014). Jnario, Executable Specifications for Java. http://jnario.org.

Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing.

Bettini, L., Crescenzi, P., Innocenti, G., Loreti, M., and Cecchi, L. (2004). An Environment for Self-Assessing Java Programming Skills in Undergraduate First Programming Courses. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies*, pages 161–165.

Bettini, L. and Damiani, F. (2014). Generic Traits for the Java Platform. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 5–16. ACM.

Brun, C. (2014). Sirius + Xtext: Love. https://www.eclipsecon.org/france2014/session/sirius-xtext.

Cecchi, L., Crescenzi, P., and Innocenti, G. (2003). C : C++ = JavaMM: Java. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, pages 75–78.

Charles, P., Fuhrer, R., Sutton Jr., S., Duesterwald, E., and Vinju, J. (2009). Accelerating the creation of customized, language-Specific IDEs in Eclipse. In *OOPSLA*, pages 191–206. ACM.

Crescenzi, P. (2015). *Gocce di Java. Un'introduzione alla programmazione procedurale ed orientata agli oggetti (nuova edizione)*. Franco Angeli Edizioni.

Crescenzi, P., Loreti, M., and Pugliese, R. (2006). Assessing CS1 Java skills: A three-year experience. *SIGCSE Bull.*, 38(3):348.

Dann, W. P., Cooper, S., and Pausch, R. (2011). *Learning to Program with Alice*. Prentice Hall.

Diehl, S. (2003). The java old-style shell. http://www.rw.cdl.uni-saarland.de/ diehl/JOSH/.

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., and Hanus, M. (2012). Xbase: Implementing Domain-Specific Languages for Java. In *GPCE*, pages 112–121. ACM.

Eysholdt, M. (2014). Xpect. http://www.xpect-tests.org.

Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *SPLASH/OOPSLA Companion*, pages 307–309.

Farooq, M. S., Khan, S. A., Ahmad, F., Islam, S., and Abid, A. (2014). An evaluation framework and comparative analysis of the widely used first programming languages. *PLoS ONE*, 9(2):e88941.

Gibbons, J. (1998). Structured programming in Java. *ACM SIGPLAN Notices*, 33(4):40–43.

Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley.

Gronback, R. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley.

Guo, P. (2014). Python is now the most popular introductory teaching language at top u.s. universities. http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext.

Heidenreich, F., Johannes, J., Karol, S., Seifert, M., and Wende, C. (2009). Derivation and Refinement of Textual Syntax for Models. In *ECMDA-FA*, volume 5562 of *LNCS*, pages 114–129. Springer.

Itemis (2015). Xtext. http://www.eclipse.org/Xtext.

Jouault, F., Bézivin, J., and Kurtev, I. (2006). TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE*, pages 249–254. ACM.

Kats, L. C. L. and Visser, E. (2010). The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463.

Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137.

Koehnlein, J. (2014). Graphical Views for Xtext. https://www.eclipse.org/community/-eclipse_newsletter/2014/august/article4.php.

Koulouri, T., Lauria, S., and Macredie, R. D. (2014). Teaching introductory programming: A quantitative evaluation of different approaches. *ACM Trans. Comp. Educ.*, 14(4):Article 26.

Leping, V., Lepp, M., Niitsoo, M., Tõnisson, E., Vene, V., and Villems, A. (2009). Python prevails. In *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, pages 87:1–87:5.

Levy, R. B.-B., Ben-Ari, M., and Uronen, P. A. (2003). The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15.

Lewis, J. (2000). Myths about object-orientation and its pedagogy. *SIGCSE Bull.*, 32(1):245–249.

MacDonald, B. (2014). To IDE or not to IDE? http://radar.oreilly.com/2014/01/to-ide-or-not-to-ide.html.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

Mason, R. and Cooper, G. (2014). Introductory Programming Courses in Australia and New Zealand in 2013 - Trends and Reasons. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, pages 139–147.

Nisen, M. (2014). These programming skills will earn you the most money. http://qz.com/298635/these-programming-languages-will-earn-you-the-most-money/.

Parlante, N. (2011). Codingbat code practice. http://codingbat.com.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, pages 204–223.

Pfeiffer, M. and Pichler, J. (2008). A comparison of tool support for textual domain-specific languages. In *Proc. DSM*, pages 1–7.

Reas, C. and Fry, B. (2014). *Processing: A Programming Handbook for Visual Designers*. MIT Press, 2nd edition.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition.

Voelter, M. (2011). Language and IDE Modularization and Composition with MPS. In *GTTSE*, volume 7680 of *LNCS*, pages 383–430. Springer.

Westfall, R. (2001). Technical opinion: Hello, world considered harmful. *Commun. ACM*, 44(10):129–130.