Wollok: language + IDE for a gentle and industry-aware introduction to OOP

Nicolás Passerini
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
npasserini@gmail.com

Carlos Lombardi
Universidad Nacional de Quilmes
Bernal, Buenos Aires, Argentina
carlombardi@gmail.com

Javier Fernandes
Universidad Nacional de Quilmes
Bernal, Buenos Aires, Argentina
javier.fernandes@gmail.com

Pablo Tesone
IMT Lille Douai
Douai, Hauts-de-France, France
tesonep@gmail.com

Fernando Dodino
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
Universidad Tecnológica Nacional
fernando.dodino@gmail.com

ABSTRACT

Students often have difficulties in learning how to program in an object-oriented style. One of the causes of this problem is that object-oriented languages require the programmer to be familiarized with a big amount of non-trivial concepts. While there exist languages that can reduce this complexity, it is difficult to find programming tools for a student who is walking her first steps into programming.

In our experience, the lack of adecquate programming languages and tools can be a significant obstacle for initial courses. Also, educative tools that differ greatly from those used in the IT industry can weaken student interest, as well as hamper the application of the learned concepts and techniques in subsequent labor experiences.

In this work we describe the *Wollok IDE*, an educative development environment conceived for learning OOP in a way that supports an *incremental learning path*, and, at the same time, facilitates the transition to industrial environments. Equally important, we briefly describe the motivations for their design and our teaching experience with these tools.

ACM Reference Format:
Nicolás Passerini, Carlos Lombardi, Javier Fernandes, Pablo Tesone, and Fernando Dodino. 2018. Wollok: language + IDE for a gentle and industry-aware introduction to OOP. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Teaching how to program has revealed itself a difficult task [6, 29]. We have individualized three specific aspects present in many initial programming courses that hinder the learning process: (a) a complex programming language, (b) too many concepts needed for a first working program and (c) programming environment that are not conceived for the specific needs of an initial student [28].

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There have been several proposals to address the difficulties in introductory OO courses by defining a specific language which provides a simplified programming model, such as Karel++ [12] and Mama [10]. This approach has been used even outside the OO world [19, 27, 31]. A step further is to provide a whole programming environment specifically designed to aid novice programmers such as Squeak [3], Traffic [23], Loop [9], and BlueJ [11].

Other environments make use of block-based or visual programming, such as Scratch [4], Etoys [17] and Kodu [26]. In our vision, these tools are suitable for stimulating interest in programming and for being used in secondary education, but not beyond that stage. Other approaches, such as Gobstones [19], focus on a first universitary programming course. Other educators propose to use industrial languages in introductory courses, such as Java [15], Eiffel [22, 23], Smalltalk [5] and Self [30].

The great differences between these programming languages and environments show that they have to be analysed in the light of the pedagogical approaches behind them. We follow the work of Lombardi *et al.,* [1, 2, 18, 25], which consists on a novel path to introduce OO concepts focusing first on objects, messages and polymorphism while delaying the introduction of more abstract concepts, such as classes, types or inheritance. This way of organizing a course provides a more gentle learning curve to students and allows them to write completely working programs from the first classes.

Also, this approach gives great importance to the programming tools used in the course, stating that they should be carefully selected and customized, taking into account the specific needs of beginner programmers, as well as the intended pedagogical view. These ideas led us to conceive a programming language, named Wollok, together with am Integrated Development Environment (the Wollok *IDE*).

The main goal of this paper is to describe how the set of tools included in the Wollok IDE[1] contribute to support a gentle and industry-aware introduction to OOP.

In Section 2 we present the problems of learning Object Oriented programming together with a brief introduction to our pedagogical approach. Section 3 describes the iterative methodology used for the development of our pedagogical approach and the tools that support

[1] http://www.wollok.org/. Source code and documentation can be found in Github (https://github.com/uqbar-project/wollok). Wollok is open-source and distributed under LGPLv3 License (http://www.gnu.org/copyleft/lgpl.html).

it, while Section 4 describes some characteristics of these tools and their role for teaching programming skills. Section 5 describes the results obtained. Section 6 analyses some key design decisions and Section 7 summarises our contributions along with some possible lines of further work.

2 WHY WOLLOK?

One cause behind the difficulties in learning OOP is the use of languages that require the student to understand a lot of abstract concepts before being able to produce the simplest program [16]. Figure 1 shows an example of a possible first program, written in Java [14]. Before being able to have a first object and send a message to it, the student has to deal with packages, classes, variable scopes, types, arrays, printing to standard output and class methods. Courses tend to spend too much time on the details of a specific language, leaving too little time to become fluent on the distinctive featuresof OOP, such as identifying objects and their knowledge *relationships*, assigning *responsibilities* and the power of *encapsulation* and *polymorphism* to build robust and extensible software.

Moreover, frequently the students do not have proper tools that could help them to overcome all the obstacles. This has even more importance if we want courses to deal with larger programs and to teach concepts such as testing, debugging and code reuse [16].

For example, writing to standard output would be considered a problem in industrial software construction and teaching the students to try out their programs in this way introduces a bad practice, which will have to be *unlearned* later. Still, some kind of user interaction is required in order to see the behaviour of the program, even if proper handling of it is beyond the scope of an initial OO course.

```
package examples;

public class Accumulator {
  private int total = 0;

  public int getCurrentTotal() { return total; }
  public void add(amount) { total += amount; }

  public static void main(String[] args) {
    Accumulator accum = new Accumulator();
    accum.add(2);
    accum.add(5);
    accum.add(8);
    System.out.println(accum.getCurrentTotal());
  }
}
```

Figure 1: Sample initial Java program which diverts student attention from the most important concepts.

Wollok provides an *simplified programming model* (SPM) that allows students to create programs containing objects, messages and polymorphism without the need for more abstract concepts such as inheritance, type annotations or even classes. Wollok also allows the incremental introduction of more abstract concepts, providing a smooth transition into a full-fledged OO programming model.

The example in Figure 2 shows a possible first program in a Wollok-based OO introductory course. Syntax has been reduced to a minimum and the basic constructs of the language match exactly the concepts we want to transmit, *e.g.,* var is used to define variables and method is used to define methods. The accumulator object is defined as a stand-alone (*i.e.,* it has no visible class) and *well-known* (*i.e.,* globally accesible) object (WKO).

```
object accumulator {
  var total = 0
  var evens = 0

  method getCurrentTotal() { return total }
  method add(amount) {
    total += amount
    if (amount % 2 == 0) { evens += 1 }
  }
  method evenCount() { return evens }
}
```

Figure 2: Sample initial Wollok object definition.

To allow the student to interact with objects while avoiding I/O a *read-eval-print-loop* (REPL) is provided. Shortly after in the course, we introduce unit testing, which slowly replaces the REPL as the main form of interacting with objects. These tools are described in detail in Sec. 4.4.

The presence of literals for lists (*e.g.,* [1,2,3]) and sets (*e.g.,* #{1,2,3}) allows us to use collections and, therefore, increase the complexity of examples that we can build before introducing classes. We also introduce *closures* at this stage.

Afterwards, we introduce classes. Wollok helps us in the transition: any pre-existent stand-alone object can be converted into a class by just changing the keyword object for class[2]. Moreover, stand-alone objects can be used in the same program and even be polymorphic with class-based objects. Examples of all these language features can be found in [25].

While neither the language itself nor the programming environment contain novel features that are unseen in industrial tools, the assemblage of selected features, each one carefully selected due to its educational value, is not found in other previous programming environments, neither educational nor industrial. Often, the rich set of tools an industrial IDE offers cannot be exploited by an inexperienced programmer. Even worst: they can cause confusion. Therefore, there is much to gain from a language and IDE that provide the exact tools a student can understand and take advantage of at each stage of his learning process.

At the same time, we have noticed that sometimes students have a hard time translating their knowledge to their professional activity. We think that a good mitigation tool ist to bring the activities in the course as close as possible to the professional practice [21]. We put special emphasis in solving the apparent contradition between customizing language and environment for student needs and at the same time keeping them close enough to their professional counterparts. At the same time, we incorporate industrial best practices such as code repositories and unit tests, adapting them to the possibilities of students with little or no programming experience.

The current study and development have been focused on university students which have had a previous subject on imperative programming. The natural extension of this work is the adaptation of these ideas to teenagers or, more generally, students without any prior programming experience.

[2]As a matter of fact, we will also change the name, as our code convention mandates lowercase names for objects and uppercase names for classes

3 METHODOLOGY

Wollok language and IDE are developed in an iterative process, guided by our pedagogical approach and at the same time providing the basis for a classroom experience, which in turn provides feedback to the process.

After each university semester, both students and teachers are surveyed. Questionnaires are designed to determine which are the topics that result more difficult for students, and to analyse the relationship to other variables, such as the order in which concepts are presented, the practice and examples used in each stage of the course and the support tools that could help grasping each concept.

Then, the results of surveys is analyzed by a board of teachers from five universities. Although each teacher has his own way in front of the course, there is an extensive basis of agreement. This consensus allows us to create shared teaching supplies, such as theoretical material, sample exercises and exams and other support tools. In the last years, a big amount of this effort has been specifically devoted to define and build the Wollok language and IDE.

During 2017, 140 students answered the surveys, including also some courses that do not use Wollok. Most of them were surveyed twice: before and after the course, in order to follow the evolution of each student and to analyse the relationship between their previous experience and their perception of the tools. Surveys are anonymous to ensure that students are not afraid of posting negative critics[3].

The information gathered from surveys has been complemented by individual interviews. To anonymity, interviews are always conducted by a teacher of another university. Interviews were guided by the same questions as in the survey, but allowing for longer, open responses and giving the opportunity for the interviewer to deepen some topics, depending of the student answers and background.

Adding up these techniques allows us to gather different flavors of data, combining the statistically more significant data obtained from surveys and with the deeper insights that can be obtained in an interview. Also, interviews allows to discover oportunities for improving the questions in future surveys.

Finally, we collected retention and approval data, along with final exams, from courses using different approaches, languages and other tools. The results of this process are detailed in Sec. 5.

4 THE WOLLOK IDE

The features that influence the experience of the learning programmer are not limited to the programming language used. The *tools* used to write, analyse and evaluate the code have a paramount relevance for this experience, and therefore for the success of the programming courses.

Beginner programmers are likely to require more guidance and make more mistakes than experienced ones. Also, the kind of support required by a experienced programmer is different from that required by a beginner. One very important feature a beginner requires from her programming environment is *discoverability*, *i.e.,* the tools should help discover possible paths of action and gently provide feedback when the student makes a mistake, helping her to understand what was wrong and how to fix the program. All programmers require tools that help them understand, navigate and explore their programs.

We decided to embed the Wollok language in an integrated programming environment, whose features are designed having in mind the specific needs of novice programmers. We consider these tools to be a fundamental part of the pedagogical approach, pursuing the following goals: (a) to guide and ease the actual code writing, (b) to detect several of the most common mistakes done by novices, providing adequate feedback, and even to suggest corrections when possible, (c) to navigate and give different perspectives of the program, (d) to test and experiment with objects, both those created by the student and those provided by Wollok.

We remark that several of the tools that the Wollok environment provides are common, in exact or approximate form, to those provided by mainstream industrial IDEs like Eclipse, Visual Studio or the Idea series. In this way, we aim to make both the programming experience more appealing to the students, and the transition to later courses and work environments softer; while giving adequate support to the learning process through the same tools.

4.1 Basic guidance for writing code

The syntactic strictness of programming languages imposes a harsh barrier on novice programmers. Errors due to mispelled keywords or lack of proper delimiter match (*e.g.,* braces) are both frequent and frustrating to them[4]. The Wollok IDE code editor offers several features that help avoiding those problems, including syntax highlighting, automatic insertion of the closing delimiter when the opening one is typed, and the proposal of a proper indentation scheme. We remark that the latter feature also aims to improve readability of code, and also to induce good code organization practices.

The Wollok IDE also provides *content assist*, *i.e.,* in certain contexts, the IDE can autocomplete an identifier name or provide a list of possible completions if there are many (*cf.* Fig. 3)[5].

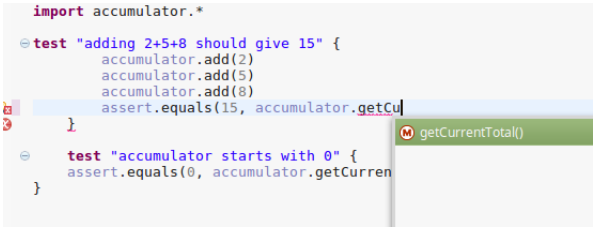

Figure 3: A list of suggestions from the Wollok IDE content assistance.

At a bigger scale, Wollok admits the definition of several objects and/or classes in the same file. This allows the teacher to go deeper in the initial examples using several objects and polymorphism without requiring the students to struggle with imports and packages. These concepts will arise later in the course, when students' programs increase their complexity to a level that demands for modularization.

[3] Students that answered both initial and final course surveys were asked to identify themselves using a nickname, allowing us to correlate their responses.

[4] Of course, block-based and visual programming tools make these problems just vanish. As we describe in the discussion, such tools would not be adequate for the intended uses of Wollok.

[5] Currently this can autocomplete variables, constants and messages sent to self, super or any WKO. Current work in progress includes a *type inferer* that will allow to extend the content assistance capabilities to any context.

4.2 Detect mistakes and help fixing them

The Wollok IDE is able to detect statically (i.e. prior to evaluation) several mistakes frequently made by students. Besides basic errors like syntax or references to undefined identifiers, the code analysis several programming errors, *e.g.,* a method must return in all execution branches or in none, or abstract classes cannot be instantiated. Other, more subtle errors are indicated as warnings, such as unused or never-assigned variables. Also, the code analysis enforces selected programming practices, *e.g.,* class definitions must follow a definite order: instance variable declarations, followed by constructors, followed by methods.

These errors are, detected while the student is typing a program and are shown in the editor. Like in several modern code editors, the source line is marked with an error sign. When the student passes the mouse pointer over that sign, an explanatory message is shown. Special care has been put in the messages, to explain mistakes in the same terms in which the teacher talks to students. In some situations, the IDE proposes possible *automatic fixes* to the detected problems. Figure 4 exemplifies this feature along with the rendering of a code error. The mispelled identifier is underlined and if we pass over the mouse on the error report, we get an error message together with possible fix. In this case, the automatic fix will insert an empty method in the adecquate position.

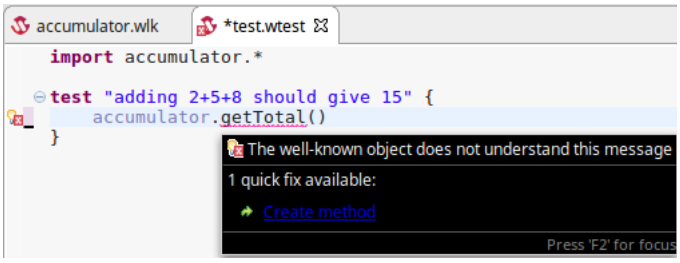

Figure 4: Example of a simple error detection.

These features aim to give a quick visualization and a proper understanding of coding mistakes. In this way, we intend to empower students to explore different ideas, providing positive feedback in case of mistakes. While this does not replace the more personalized feedback a teacher provides, in several situations, a sensitive automatic feedback helps students not to stay stuck with simple errors waiting for teacher or colleague assistance. This, in turn, allows for more agile lab classes and, therefore, for the possibility of including more exercises during a course.

Automatic detection acts as a (basic) assistant teacher, *i.e.,* some simple topics that are not crucial for the course can be left for the student to learn by herself in the interaction with the IDE, instead of to being explained in class. This is specially useful for some errors that only some students are likely to make, *e.g.,* students with previous experience in OO languages but without a good theoretical support. We prefer to avoid covering these errors in class, because it would imply to show a bad solution to the rest of the students, who had otherwise not thought about it. While anticipating errors could be a fine strategy in an advanced course, it frequently confuses beginners. By having the IDE IDE detect the mistake and show possible solutions, we can tackle the problem only for the students that effectively incur in this kind of errors.

4.3 Understanding and navigating a program

Code navigation and visualization tools can significantly improve the programming experience, both for students and practitioners. Lack of such tools frequently misleads students to avoid correct modularization of their program, as they feel more difficult to cope with a program that is divided in several small pieces.

The Wollok IDE supports basic navigation tools, such as from a class reference to its definition, from a message send to the corresponding method (when such method can be determined) and back-and-forth navigation on code portions resembling that of Web browsers.

Some code visualization tools are available as well. An outline view and automatically generated static diagrams (*cf.* Fig. 5 right and left resp.) provide streamlined views of the classes and WKOs defined. Clicking a class, WKO or method pops up the corresponding definition in the code editor. These tools are useful to induce students to abstract themselves from the details of some portions of a program, understanding objects and classes as black boxes that provide some services, instead of attempting to have all of them in their head at all times.

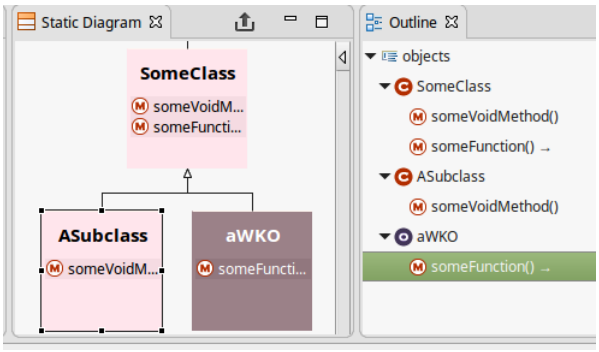

Figure 5: Static diagram and outline.

The Wollok IDE also includes some tools to help students to manage the whole set of code produced along a course. Source files are organized in *projects*, which have a predefined directory structure including both files for class/object definitions and tests. The generated file structure is adequate to the use of a source code repository such as GIT or SVN. These features alleviate the frequent difficulties novice programmers find to organize their source files and, in particular, to coordinate work in group assignments. We note to this respect that it is normal to see members of a work group sharing code by sending e-mail with *zip files* between each other. The time spent trying to reconcile different versions of the program, or even trying to understand which is the latest one, distracts them from learning the actual topics of the course. Another benefit of providing suitable source organization tools is the enforcement of good development practices.

4.4 Tests and experiments

As we described in Section **??**, Wollok provides two ways of working with the defined objects and classes: the REPL and the definition of tests.

The REPL is first introduced and provides a simple environment for direct object manipulation. Running a program in the REPL

brings all defined objects to life and allows the student send messages to the them and see how they respond (*cf.* Fig. 6). In some courses, we even *start* on the REPL by sending messages to objects provided by the teacher. In this way, students get familiarized with the most important concepts of the OO paradigm: *object* and *message*, before going into the details about how these objects have been implemented. The REPL also allows the programmer to define local variables, useful to remember intermediate results to be used in further operations, allowing for more complex object manipulations.

```
Wollok REPL Console
Wollok interactive console (type "quit" to quit):
>>> accumulator.add(2)
>>> accumulator.add(5)
>>> accumulator.add(8)
>>> accumulator.getCurrentTotal()
15
>>>
```

Figure 6: Sample usage of the accumulator in the REPL

As the REPL interaction grows, in a short time the students realize that they are doing repetitive operations there and start looking for automation; at this moment, *automatic tests* are introduced. Fig. 8 shows the test runner tool output for the test shown in Fig. 7.

```
import accumulator.*

test "adding 2+5+8 should give 15" {
  accumulator.add(2)
  accumulator.add(5)
  accumulator.add(8)
  assert.equals(15, accumulator.getCurrentTotal())
}

test "accumulator starts with 0" {
  assert.equals(0, accumulator.getCurrentTotal())
}
```

Figure 7: Sample test Wollok program.

The Wollok test runner simplifies unit test creation for for begginners by automatically providing *test isolation* [20], *i.e.,* global state is reset after each individual test case is run. For example, the messages sent to the accumulator in the first test in Figure 7 will not affect the state of the accumulator in the second test.

It is important to notice that the test requires a *higher level of abstraction* than the REPL. Now the student has to anticipate the result that some expression should yield, write both the expression and the expected result and interpretate a *green bar* as a signal that the answer was the expected, without ever seeing it.

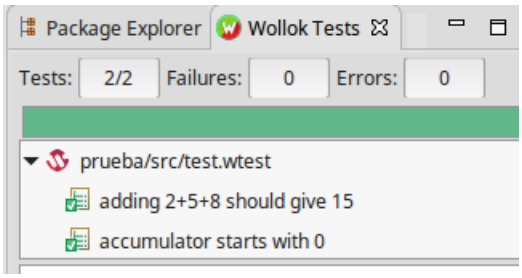

Figure 8: Test runner view after a sucessful test run.

5 RESULTS

Since March 2015 Wollok is being used to teach introductory OOP courses at university level. Until 2017 it has been used in more than 30 courses in five different universities, reaching almost 1000 students. It has also been used at highschool level. Many of the ideas in Wollok where present in other tools we had previously developed, going back to 2006.

In these 12 years, these ideas, although always evolving, have been applied by more than 100 teachers and 6000 students. In several cases, approval rates changed from 30–40% to 80–90%. At the same time, the level of the courses has been consistently increased, covering more topics and requiring the students to build bigger programs with higher design quality.

As a way to measure the knowledge level, we analysed the difficulty in final exams along 12 years. In some cases, the programs required to pass a final exam used to consist of only one or two classes, with no more than 6–8 methods, and only one (fairly simple) usage of polymorphism. Current exams require the student to write a program with 10–15 classes and no less than 20 methods (excluding getters and setters). Also, there are more usages of polymorphism frequently including one that requires to come up with a non-obvious abstraction in order to sort out the problem. Metrics from 2017 show that 88% of the students that started the course took the final exam, 96% of those where able to complete the required program (and tests) in less than four hours. The final approuval rate was 79%.

For a better understanding to the effect of using Wollok, we've realized surveys and interviews, organized according to the ideas described in Section 3. We surveyed 15 courses during 2017, obtaining 133 responses. These surveys show that 84% of students found Wollok at least significantly easier to understand than other languages (*cf.* Fig. 9), and 95% thinks that Wollok has helped them understand their mistakes and learn from them. At the same time 70% of the students consider that having learnt Wollok will help them in their professional practice.

	Abs.	Sign.	Part.	Not	N/A
Easier to learn than other langs.	33	74	16	4	4
Helps understanding errors	47	78	5	1	0
Similar to prev. known langs.	19	44	31	15	22
Helps professional activity	26	52	18	15	19

Figure 9: Student evaluation of Wollok Language and IDE. For each question, each student had to select between "Absolutely", "Significantly", "Partially" or "Not at all".

Interviews confirm that students have no problems recognizing the applicability of the concepts learned with Wollok. For example, several students with professional experience affirmed that learning OOP with Wollok led them to improve their professional practice. Others indicated that having learned OOP with Wollok was of significant help to learn other modern OOP languages, such as *Swift*.

Interviews also confirmed that the subject allows the students to get a good grasp of theoretical knowledge, that they are aware of the new theory they incorporated and that they are capable of applying these concepts in different technologies and situations.

Students assign great value to the tools of the IDE. Some of the most appreciated tools are those that help understanding and visualization, such as outline, diagrams and class catalog. The other most referenced characteristic of the IDE is its error reporting mechanism;

most students mentioned it as a great help for debugging their programs. Finally, also integrated tooling is considered valuable, such as testing facilities and git integration.

On the downside, we discovered that students sometimes have difficulties finding some of the features of the IDE. We consider that this problem can be alleviated by both improving student documentation and building better teacher guidelines.

Also, they criticized that the IDE can be confusing because of having too many tools they do not know how to use. This is a known issue, due to an implementation trade-off: we intend to build a minimalistic IDE in which each tool is precisely selected according to student needs, but it would require a significantly bigger amount of work than current, Eclipse-based implementation. Even so, Eclipse is a customizable platform and the depuration of the IDE to remove superflous tools is a work in progress.

Other critics asked for improvements in error messages, autocompletion and smart suggestions, which shows that the students perceive the added value of this kind of tools.

6 TO IDE OR NOT TO IDE

A frequent controversy between software programmers is about the convenience of using an IDE or a simpler text editor for writing code. In the last decade, several languages, frameworks and other tools have become popular for which there are fewer visual or integrated environments.

This scarcity of tools has diverse roots. In some cases, the lack of type information undermines the possibility to implement features such as code completion, automatic refactorings or code navigation. In other cases, languages and frameworks change so fast that tools can not catch up. Frequently, there is also a matter of taste. Also, some teachers argue that providing the student with too many tools will make him dependent on those tools.

In our view, tools that simplify day to day work can not be neglected. We found that professional programmers make use of a good amount of tools to program consistently and efficiently. The most popular text editors in industry are those that allow for additions in the form of *plugins*, where the programmer can create his own personalized development environment. Other tools that are not integrated into the development environment, are inserted into the development process by other means; *e.g.,* a continuous integration process may run a *linter* on each commit, check the build and run tests. So, instead of a discussion about whether we need powerful tools, we rather see an evolution from heavy monolithic environments with lots of tools onto an ecosystem of light tools that have different ways to integrate with each other, allowing a developer or team to create a unique environment which accomodates to their specific needs and taste.

Still, in our specific case, we opted for an *integrated* environment because it simplifies the set up for beginners. In more advanced courses, we think that it could be a good idea to let the students build their own environments.

7 CONCLUSION

The Wollok language and IDE have been put into practice for already two years, targeting hundreds of students. They have been successful in supporting an incremental learning path, allowing the users to train their OO modelling skills using a very simple programming model and providing a smooth transition to industrial languages and associated tools.

The IDE allows students to program in a controlled environment which helps them to avoid getting stuck, frequently guiding them to use the best programming practices. Also, it provides a controlled environment which empowers students to use their intuition, test their ideas and explore new possibilities. The combination of REPL and tests, we have succeeded in completely avoiding the need for I/O or undesired debugging practices, such as the inclusion of println expressions along the code.

By providing simplified versions of several industry-like tools, the Wollok IDE allows to introduce professional development practices early on in the curricula, helping the students in getting familiarized with the kind of practices and environment they find both in later subjects as in their professional jobs.

The main focuses of attention for the Wollok IDE development team are the detection of programming errors and bad practices, and the provision of quick fixes, content assistance and refactorings. A cornerstone to achieve these goals is the type inferer, which is one of our current main objectives. Still, providing a type inferer for a language such as Wollok has many subtleties, which deserve an independent study [24].

Other future tasks include extending the REPL to be a full-fledged editor which re-evaluates expressions after a code change, displaying the new results (such as Scala Worksheets[6]); improving team work support, and a web-based version of the interpreter.

ACKNOWLEDGEMENTS

We want to thank all the people who participated in the ObjectBrowser, Loop, Hoope and Ozono projects, as well as the teachers and students that provided feedback from their use of those tools, leading us to the ideas presented here.

REFERENCES

[1] C.Lombardi and N.Passerini. Alumnos, docentes y recorridos en una materia de programación informática. UNQ – Argentina, Oct. 2008.
[2] C.Lombardi, N.Passerini, and L.Cesario. Instances and classes in the introduction of object oriented programming. DC - FCEN - UBA, Buenos Aires, Argentina, Dec. 2007.
[3] D.Ingalls, T.Kaehler, J.Maloney, S.Wallace, and A.Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, Oct. 1997.
[4] D.Malan and H.Leitner. Scratch for budding computer scientists. *SIGCSE Bull.*, 39(1):223–227, Mar. 2007.
[5] S. Ducasse. *Squeak: Learn programming with robots*. Apress, 2006.
[6] E.Dijkstra. On the cruelty of really teaching computer science. *Communications of The ACM*, 1989.
[7] F.Nielson and H.Nielson. Type and effect systems. In *ACM Computing Surveys*, pages 114–136. Springer-Verlag, 1999.
[8] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.
[9] C. Griggio, G. Leiva, G. Polito, G. Decuzzi, and N. Passerini. A programming environment supporting a prototype-based introduction to OOP. In *Proceedings of the Intl. Workshop on Smalltalk Technologies*, IWST '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.
[10] A. Harrison Pierce. Mama-an educational 3d programming language.
[11] J.Bennedsen and C.Schulte. BlueJ visual debugger for learning the execution of object-oriented programs? *Trans. Comput. Educ.*, 10(2):8:1–8:22, June 2010.

[6] https://github.com/scala-ide/scala-worksheet

[12] J.Bergin, J.Roberts, R.Pattis, and M.Stehlik. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1996.

[13] J.Fernandes, N.Passerini, and P.Tesone. Wollok – relearning how to teach object-oriented programming. In *Workshop de Ingeniería de Software y Tecnologías Informáticas*, Universidad Tecnológica Nacional, Buenos Aires, Argentina, 2014.

[14] K.Arnold, J.Gosling, and D.Holmes. *The Java programming language*, volume 2. Addison-wesley Reading, 1996.

[15] M. Kölling and J. Rosenberg. Guidelines for teaching object orientation with java. In *ACM SIGCSE Bulletin*, volume 33, pages 33–36. ACM, 2001.

[16] M. Kölling. The problem of teaching object-oriented programming, part i: Languages. *JOOP*, 11(8):8–15, 1999.

[17] Y.-J. Lee. Empowering teachers to create educational software: A constructivist approach utilizing etoys, pair programming and cognitive apprenticeship. *Comput. Educ.*, 56(2):527–538, Feb. 2011.

[18] L.Spigariol and N.Passerini. Enseñando a programar en la orientación a objetos. UTN FRC, Córdoba, Argentina, Nov. 2013.

[19] P. López, E.Bonelli, and F.Sawady. El nombre verdadero de la programación. Aug. 2012.

[20] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[21] R. McDermott, M. Zarb, M. Daniels, and V. Isomöttönen. First year computing students' perceptions of authenticity in assessment. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 10–15, New York, NY, USA, 2017. ACM.

[22] B. Meyer. Towards an object-oriented curriculum. In *TOOLS (11)*, pages 585–594, 1993.

[23] B. Meyer. The outside-in method of teaching introductory programming. In M.Broy and A.Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 66–78. Springer Berlin Heidelberg, 2003.

[24] N.Passerini, P.Tesone, and S.Ducasse. An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types. Cambridge, England, Aug. 2014.

[25] N. Passerini, C. Lombardi, J. Fernandes, P. Tesone, and F. Dodino. Wollok: Language + ide for a gentle and industry-aware introduction to oop. In *2017 Twelfth Latin American Conference on Learning Technologies (LACLO)*, pages 1–4, Oct 2017.

[26] M. Research. Kodu.

[27] R.Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. Wiley, New York, NY, USA, 1st edition, 1981.

[28] M. T. Singh. How to teach programming languages to novice student and problems in learning of students. *Journal of Computing Technologies (JCT)*, 1(2):5, 2012.

[29] T.Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58. Citeseer, 2002.

[30] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987.

[31] W.Feurzeig, S.Papert, M.Bloom, R.Grant, and C.Solomon. Programming-languages as a conceptual framework for teaching mathematics. *SIGCUE Outlook*, 4(2):13–17, Apr. 1970.