Wollok – Relearning How To Teach Object-Oriented Programming

Javier Fernandes
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
javier.fernandes@gmail.com

Nicolás Passerini
UTN – F.R. Buenos Aires
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
npasserini@gmail.com

Pablo Tesone
Universidad Nacional de Quilmes
Universidad Nacional del Oeste
Universidad Nacional de San Martín
tesonep@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. For several years we have been teaching introductory OOP courses usind an *incremental learning path*, which starts with a simplified OOP model consisting only of objects, messages and referencies. This learning path is supported by a customized development enviroment which enabled the creation of programs using this simplified programming model, and allows us to postpone the introduction of more abstract concepts like classes or inheritance.

In this work we propose an enhancement to this learning path focusing on the transitions between the stages of the course. We also present a new educationa programming language named Wollok, which allows to maximize the accuracy in the selection of concepts to present. Finally, Wollok is accompanied by a programming environment which has lots of tools to guide the student and help detecting mistakes, but at the same time is in line with the most common professional practices.

1. Introduction

Object-oriented programming (OOP) has become the *de facto* standard programming paradigm in industrial software development. Therefore, in the last years software engineering curricula have put more emphasis in object-oriented courses. Still, students often have difficulties in learning how to program in an object-oriented style which show up both in academic and in industrial environments.

Several causes have been blamed for the difficulties in OOP learning. First, OO courses tend to focus too much on syntax and the particular characteristics of a language, instead of focusing on OOP distinctive characteristics. Second, many OO languages used in introductory courses do require to grasp a lot of quite abstract concepts before being able to build a first program. Finally, poor programming environments are used, although we are at a time where an unexperienced programmer could be making great use of the guidance a good programming environment could provide. Nico ►*En realidad algunos de estos problemas no son exclusivos de OOP, habría que ver si queremos decir algo al respecto.*◄

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++ [5] Nico ►*¿otros?*◄ . This approach has been used even outside the OO world [9, 17, 23]. A step further is to provide a whole programming environment specifically designed to aid novice programmers such as Squeak [12], Traffic [19] and BlueJ [4].

The great differences between these programming languages and environments show that they have to be analyzed in the light of the pedagogical approaches behind them. The tools are of little use without this pedagogical view. For example, some educational languages and environments are designed to be used in *object-first* approaches, *i.e.,* for students without any previous programming knowledge [3, 7]. Other languages are focused on teaching to children or teenagers, such as Scratch [18] and Etoys [14]. Finally, there are many approaches which emphasize the importance of *visualization tools* to simplify the understanding of the underlying programming model [8, 25].

Previous works from our team [11, 15, 16, 27] have described an approach consisting of (a) a novel path to introduce OO concepts, focusing on objects, messages and polymorphism first, while delaying the introduction of classes and inheritance and (b) a reduced and graphical program-

ming environment which supports the order in which we introduce the concepts, by allowing to build OO programs without the need of classes. Our approach focuses on the concepts of object, message, reference and object polymorphism, while delaying the introduction of more abstract concepts such as types, classes and inheritance. These way of organizing a course provides a more gentle learning curve to students and allows them to write completely working programs from the first week.

While this approach proved to be successful in providing the students with a more profound knowledge of OOP at the same time as raising pass rates, we feel that there is still room for improvement, in four areas: (a) integrating object-based and class-based programs together, (b) creating automatic ways to produce a graphical user interface (GUI), (c) enhancing the programming environment with a type inference tool that helps avoiding some common mistakes and (d) narrowing the distance between our language and tools with those more frequently used in industrial development.

Nico ►*No sé si hablar de los temas de colecciones, son un poco particulares de Ozono. También del environment se podrían decir más cosas que sólo el sistema de tipos.*◄

Nico ►*falta paper structure*◄

2. Problem Description

On cause behind the difficulties in learning OOP is that courses often focus on syntax and the particular characteristics of a programming language, leading students to concentrate on these relatively unimportant details rather than the underlying object-oriented modelling skills. Also, the use of industrial languages to introduce OOP requires the student to understand several concepts before being able to run his first program. [13]

By focusing on those details, many students fail to comprehend the essential model that transcends particular programming languages [1]. Figure 1 shows a typical first program, written in Java [2]. To get this program running, the student has to walk through a minefield of packages, classes, scoping, types, arrays and class methods among other complex concepts. However, he is still not able to do more basic OO programming, such as sending a message to an object.

```
package examples;

public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

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

For all these reasons, courses tend to spend too much time concentrated on the mechanistic details of programming constructs, leaving too little time to become fluent on the distinctive characters of OOP, such as identifying objects and their knowledge *relationships*, assigning *responsibilities* and taking advantage of *encapsulation* and *polymorphism* to make programs more robust and extensible.

To make things worse, frequently the students do not have proper tools that could help them overcoming all the obstacles. Already in 1999, Kolling *et al.,* had established the importance of environments in introductory courses [13]. He stated that earlier introductory courses focused on the development of algorithms in procedural or functional languages. To do this, an editor and a compiler was all that was needed for the practical part of the work. Modern courses now use object-oriented languages and subject material taught includes testing, debugging and code reuse. This creates the need to deal with multiple source files and multiple program development tools from the very start. To give a beginning student a chance to cope with this increased complexity, better environment support is needed.

The failure of students to understand the essential object-oriented concepts shows both in academy as in industry. In academic environments we find very low completion rates in introductory OO courses. Moreover, students in their very beginning of an informatics career which fail their introductory programming courses, are often likely to give up their studies[20]. In industrial development, we find that these hindrances reduce the opportunity of students to apply the concepts of the paradigm effectively in their further professional practice, resulting in several IT-projects not taking advantage of the possibilities offered by the potential of good object-oriented practices. [15]

Nico ►*Contar algo de otras propuestas anteriores a la nuestra*◄

Our previous work in this area is based on the proposal to change the order of subjects in OOP introductory coures [15]. To enable this we provide a simplified OO language with an ad-hoc educational programming environment named Ozono[1], in which students can create objects and define their behavior directly, without the need for classes and inheritance [11]. Ozono is based on Smalltalk[2]. and thus it is a *dynamic language*.

We take special advantage of the dynamic characteristic of the language because it allows very simple uses of polymorphism, *i.e.,* any two objects that understand a common set of messages can be treated polymorphically without worrying about inheritance or *interface* implementation. This enables the students to exploit polymorphism already in their programs after only two or three lectures. Nico ►*Qué otras ventajas queremos contar?*◄

[1] The name has been changing along these years. Other names for our programming enviroment have been ObjectBrowser, Loop and Hoope

[2] The latest version currently in use in universities is based on a Smalltalk dialect named Pharo [6].

This approach produced impressive results and therefore has been adopted in other courses and universities. The simplified programming model, allows the students to spend more time working with the essential concepts of the OO paradigm. This, in turn allows for more practice and more complex exercises, even incorporating more advanced OO techniques such as *design patterns* [10], which in traditional approaches are normally postponed to a second OOP course.

Still, the experience of eight years in four universities[3] and thousands of students has provided us with new insights of the learning process, which is what lead us to introduce several adjustements in our approach.

In the first place, starting with a *classless* language mandates a change of language and enviroment in the middle of the course, which is confusing for some students. To solve this, we propose to integrate *class-based* and *self-defined* objects in the same language. Nico ►*Ver bien qué nombres les ponemos y definir los conceptos antes.*◄

Second, we think that the absence of static typing information (which Ozono inherits from Smalltalk) prevents our programming environment to aid the students in finding some typical beginner mistakes. Still, we do not want to force the students to add type annotations to their code, because that would distract them from the concepts we want to focus on. We propose to settle this apparent controversy by enhancing our programming environment with a type inferer. By doing so, the student is not required to care about type annotations, and at the same type we can detect some programming mistakes and aid the student in solving them.

Last, we have detected that sometimes the students which seem to understand the main concepts and can apply them in interesting ways to create medium to complex program, then have a hard time translating this knowledge to their professional activity. We think that a good mitigation plan for this problem starts with bringing the activities in the course as close as possible to the professional practice. For that matter, we propose to incorporate industrial best practices such as code repositories and unit tests, adapted to the possibilities of students with little or no programming experience.

The renewed approach is supported with a new programming language, named Wollok, and a programming environment which aids students to write, test and run programs. Wollok is designed to give support to our pedagogical approach; it allows to define both classes and standalone objects, incorporates a basic type inferer and provides a simple syntax to define unit tests.

A big difference with Ozono and other educational OO languages is that Wollok is a *file-based* language. While we recognize the value of the *image-based* approaches, we also are aware that most industrial programming enviroments are file-based. Therefore, we think that a file-based approach will shorten the distance between the classroom and the professional activity, which is one of our main goals. Moreover, image-based enviromnents tend to blur the distinction between the programming environment and the program under development, which can embody great possibilities for advanced programmers but often does not more than confuse beginners. Also, being file-based, Wollok allows for integration for the most popular code repositories. Javi ►*As well as other potential usages still not proven like code-reviewing tools ?*◄

Finally, the programming enviroment incorporates several advanced characteristics from professional environments, which improves the coding experience, and at the same time allows the student to familiarize himself with the kind of tools he will face in his later professional activity. For that matter, the environment incorporates content assist, automatic refactorings, advanced code navigation, language semantics-aware search tools and error highlighting *while-you-type*

It is important to notice that neither the Wollok language nor the programming environment contain novel features that are unseen in industrial tools. Therefore, the distinctive characteristic of these tools is the search for a programming environment which (a) supports our pedagogical approach, (b) feeds the student with a set of tools which are adequate to his current knowledge and (c) gently prepares him to be using industrial-level tools.

A big amount of effort in our research has been put in looking for solutions that can solve the apparent controversy between the objectives (b) and (c). Often, the rich set of tools an industrial language or programming environment offers, cannot be exploited by an unexperienced programmer or even confuses him. On the other end, we think that poor programming environments fail to help the students to make his first steps in programming and so trims the possibilities of introductory courses. Therefore, there is much to gain from a language that has the exact features a teacher wants to teacher and a programming environment which provides the exact tools a student can take advantage off at each time of his learning process.

Nico ►*Esto tenía ganas de decir pero no me doy cuenta dónde ponerlo... no sé si volarlo y ya.*◄ So far we have focused on university students which have had a previous subject on inperative programming. It is a pending job to adapt these ideas to teenagers or more generally students without any prior programming experience.

3. Wollok: The Language

Wollok is a brand new language, built to specifically give support to our pedagogical approach. Many ideas have been inherited from our previous projects, such as Ozono or Loop. The most important of these inherited characteristics is the ability to create objects and treat them polymorphically without the need of type annotations, classes or inheritance. Also, as its predecessors and unlike other pedagogical tools,

[3] Ozono is currently used in Universidad Tecnológica Nacional, F.R. Buenos Aires and F.R. Delta, Universidad Nacional de Quilmes, Universidad Nacional de San Martín and Universidad Nacional del Oeste.

Wollok is a *general purpose* language, *i.e.,* it is not tied to any specific domain.

One of the main objectives of building a new language is to provide a smoother transition from the first phase of the course, in which students use a simplified programming model and the second phase, in which they use a full-fledged OOP language. With our previous programming tools students were required to discard, in the middle of the course, the programming model and enviromnent they already knew and were used to. This transition has sometimes been traumatic, because the process to define an object has to be relearnt and the tools they had been using up to that point are no longer available. With our new approach, the tools they first learnt are going to continue being available through all the course, together with the new ones that are incorporated in later lectures. This is also consistent with some modern industrial OO languages that allow to define both classes or standalone objects, such as Scala [22].

Also we reduced to a minimum the syntax and the most basic constructs of the language. While this objective was already present in our previous work, the implementation strategy of Wollok allows us to go much further in accomplishing this goal, *cf.* Sec. A. The example in figure 2 shows how the classical hello world would look in Wollok. To build this first program students are not required to know about typing, scoping or packaging. The only required construct is the **program** and the only command is a message send. Both the receiver and the parameters are built-in objects which will be handled in the same way as user-defined objects. The concepts required to understand this program are no more than program, object, message and argument passing.

```
program {
  console.println("Hello World!")
}
```

Figure 2. Sample initial Wollok program.

Nico ►*Esto requeriría una mención en la intro, o en general una explicación más detallada.*◄ Another concept we propose to emphasize in the first programming courses is the control of side effects, *i.e.,* a programmer should be aware of the potential side effects of each portion of code. The most basic feature in Wollok to controlling side effects is the ability to differentiate variables (defined using `var`) from constants (defined using **`val`**). One step forward is to incorporate an *effect system* [**?**], *cf.* Sec. 7.

To sum up, there are several simple features which help structure the way a student sees his programming activity. *Object literals* and *collection literals* reduce boilerplate on object creation, since we think that the excess of bureaucracy to create an object helps to build up the belief that using objects or collections is far more complicated than using numbers or strings, which in turns leads to *primitive obsession* [**?**].

Each Wollok file has to be defined as *program*, *library* or *test*. Only programs and tests can be run. Libraries can only be *imported* from programs, tests or other libraries. This concepts push students onto modularizing their programs into smaller units that can be reutilized.

Figure 3 shows some of the mentioned features. The programs includes two objects which are treated polymorphically, collections and block closures. Students should be able to build such a program after four lectures. In the first lecture we introduce objects, messages, methods and references; in the second one we focus on polymorphism; and in the following two we work with blocks and collections.

Nico ►*Acá falta hablar de la diferencia entre objeto y referencia*◄

```
program myProgram {
  val optimistic = object {
    method hiThere() {
      "Hi, what an amazing day !"
    }
  }

  val pessimistic = object {
    method hiThere() {
      "Don't talk to me, it's a terrible day!"
    }
  }

  val all = #[optimistic, pessimistic]

  console.println(all.map[p| p.hiThere()])
}
```

Figure 3. Simple polymorphism example.

4. A Customized Programming Environment

Beginner programmers are likely to require more guidance and make more mistakes than experienced programmers. Therefore, we think that is much to gain from a good programming environment which structures the programming experienced and helps the students to identify common mistakes.

The Wollok programming environment includes a lot of features that provide guidance to the student. *Content assist* shows the students what are his possibilities at any moment and feeds automatically into the code the most usual constructs, allowing the student to concentrate less on syntax and more in the modelling of the exercise problem. *Quick-fixes* allow Wollok not only to highlight problems in the student's code but also to propose automatic solutions for some usual mistakes. *Advanced code navigation* and *smart reference searches* allow the programmer to better understand the dependencies in his program. Nico ►*¿Se les ocurre cómo mejorar eso?*◄ Moreover, *automatic class diagrams* provide a high level view of the program and also helps understanding.

Also, the programming environment has many tools intended to help detecting mistakes, even while the student is writing code. *Syntax highlighting* helps identify the most simple mistakes by providing immediate feedback when something is not right. Moreover, the environment provides *real-time highlights* for syntactic mistakes. Finally, the *type inferer* allows to detect more subtle mistakes. All these tools allows the student to gain more control of his code, keeping him away from feeling lost, which is otherwise a common situation for a student walking his first steps into programming.

The type inferer is one of the most distinctive characteristics of the Wollok programming environment. We think that type inference is key to a simple programming environment. On one side, it allows to detect lots of common mistakes *before running the program*: if an object does understand a message, if a wrong argument is passed, if incompatible types are mixed or even miss-spellings. In environments without this capability it takes more time to detect errors. Moreover, it is not uncommon that a type mistake produces a runtime error in a place different from where the mistake was done, producing confusion.

Still, providing a type inferer for a language such as Wollok has many subtleties, which deserves an independent study [**?**]. On one side we require it to be able to work without type annotations and at the same time provide feedback useful for an inexperienced programmer. On the other side, the type system is rather complex; for example, the presence of stand-alone objects requires the type system to handle *structural types*, since a named type system would not allow them to be treated polymorphically. Also, we want to be able to treat polymorphically stand alone objects with class-based objects.

4.0.1 Checkeos y validaciones

Todos los checkeos y problemas generados se muestran agrupados en una vista dedicada a tal fín (Problems).

- **De estilo**: para promover uniformidad y consistencia de código. Ejemplos:
  - *Nombres*: variables camelcase comenzando en minúscula, nombres de clases camelcase iniciando mayúscula, packages en minúsculas, etc.
  - *Orden y agrupamiento*: dentro de un objeto o clase, primero se definen sus referencias internas, luego constructores y finalmente los métodos.
  - *Separación de programas*: las clases sólo se pueden definir en archivos de tipo *librería*, no dentro de un *program*.
  - *Evitar referencias duplicadas*: no se puede definir una referencia con nombre ya utilizado en alguna otra referencia del contexto (local, método, clase/objeto, etc.). Ni tampoco si ya está definida en la superclase.
- **De resolución de referencias**: para evitar referencias a variables inexistentes y, en la medida de lo posible (por ser de tipado implícito) de envío de mensajes. Ejemplos:
  - *Referencias inexistentes*: a variables locales, parámetros, o internas (clase/objeto).
  - *Constructores inexistentes*: evaluando existencia de la clase, y compatibilidad en el número de parâmetros.
  - *Envío de mensajes (a this)*: al ser a this se pueden realizar checkeos por la existencia del método y compatibilidad de parámetros, incluso sin involucrar al sistema de tipos.
- **De uso de referencias**: para la detección de código erroneo o bien desactualizado. Por ejemplo: warnings por referencias nunca utilizadas, nunca asignadas, o utilización de variables en lugar de valores.
- **De estructura**: evitan por ejemplo inconsistencias en las estructuras creadas por el alumno. Por ejemplo, se checkea que un método marcado como *override* efectivamente esté sobrescribiendo.

Nico ▶*override*◀

5. Discussion

Se podría hablar de la discusión sobre si proveer formas de compartir comportamiento sin clases, como clonado. También de la necesidad de crear colecciones independientemente, aunque es un poco específico de Ozono.

Acercar la experiencia de aprendizaje a las prácticas industriales: (acá el palo de que la imagen sólo existe en smalltalk, y en la industria nadie la usa. Atrás de eso, la idea de archivos, y poder compartir con SVC. Por último la idea de actualizarse a un lenguaje con influencia de lenguajes modernos como xtend, scala, ruby, etc.)

6. Related Works

The first aspect to analyse is the way an introductory course has to be, Vilmer *et al.,* [28] presents a work exposing the advantages of the implementation of object-first introductory courses. Also, Moritz *et al.,* [21] presents a way of starting the learning of a programming language using an object-first way using multimedia and intelligent tutoring. Another interesting work in this area is the one from Sajaniemi *et al.,* [26] who presents another way to introduce the main concepts. All this authors propose the idea of using a commercial language like Java or C#, they are not addressing the problems of using this industrial level languages as an start point. On the other hand, Lopez *et al.,* [17] presents a successfully way of teaching using functional-first in an introductory course.

Another aspect to analyse is the use of an industrial programming language or a custom one. In this subject, [17] *et al.,* have chosen the same approach as us, but in a functional-first solution. Both languages centre in the main concepts

of the paradigms. Another custom language centring in the main concepts is BlueJ [4]. This implementation shares with Wollok the idea to simplify the language, but it is only class centred. Wollok is both class and object centred, so there is no need to teach classes to start learning basic concepts of the paradigm.

There are interesting works in the Visual languages as a way of teaching OOP: Scratch [18], Etoys [14] and Kodu [24]. But all of them are far away of a professional development environment, so the transition to a industrial level work is not so easy as with Wollok.

7. Conclusion

Wollok is an educative, object-oriented programming language which is accompanied by an advanced programming environment. Both tools are highly customized to give support to an introductory OOP course. Our approach consists of the combination of these three cornerstones.

First, an *incremental* Nico ►*Revisar la idea de Javi*◄ *learning path* choosing exactly which are the concepts we want to teach and the order we want to teach them. The learning path starts with a *simplified programming model* (SPM), *i.e.,* one which uses less concepts than a full-fledged OOP language. The SPM allows the student to build simple programs without requiring more advanced concepts. The path should attach the SPM with a good set of programming exercises, specifically oriented to be easy to build in the selected SPM. Once the student has mastered the concepts on the SPM, we can go a step further and introduce the next set of concepts. Nico ►*Acá se podría hablar de constructivismo.*◄

Next, defining our own programming language, allows us to give full support to the selected learning path, avoiding the need of explaining complex concepts too soon in the course or forcing the student to write boilerplate code which he cannot yet understand.

Finally, a good programming environment, helps detecting errors, provides guidance and most significantly allows the student to *explore*. We have found that often students are afraid to search for solutions not seen in the class or test their own ideas, which leads them to restricting themselves into a smaller set of concepts and tools they feel more secure about. A controlled environment empowers students to look around and explore new possibilities.

One major objective in our future work is the integration of more *automatic user interaction* tools into the Wollok environment. Our objective is to enable the students to have visual and interactive programs without requiring them to learn the subleties of GUI building, extending the ideas in Gobstones [17] to object-oriented domains. Some advances in this area can be seen in our previous work named Hoope [**?**].

The second major objective is to continue improving the detection of programming errors. A cornerstone to achieve this goal is the type inferer, which is our current focus. The other half of our future work in this area is a powerfull *effect system* [**?**].

Another characteristic of programming in the real world is the need to work in teams. The success of object-oriented languages is partly due to their advantages in group projects. It is necessary teach our students about the techniques needed for teamwork, right from the beginning. To do this, it is essential that the environment has some form of support for group work [13]. Also we want to include more *automatic refactorings*.

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] Computing curricula 2001. *J. Educ. Resour. Comput.*, 1(3es), Sept. 2001.

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

[3] D. M. Arnow and G. Weiss. *Introduction to programming using java: an object-oriented approach*. Addison Wesley, 1998.

[4] 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.

[5] 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.

[6] A. P. Black, S. Ducasse, O. Nierstrasz, and D. Pollet. *Pharo by Example (Version 2010-02-01)*. Square Bracket Associates, 2010.

[7] K. B. Bruce, A. Danyluk, and T. Murtagh. A library to support a graphics-based object-first approach to CS 1. In *ACM SIGCSE Bulletin*, volume 33, pages 6–10. ACM, 2001.

[8] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. *SIGCSE Bull.*, 35(1):191–195, Jan. 2003.

[9] 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.

[10] E. Gamma and K. Beck. *Contributing to Eclipse*. Addison Wesley, 2003.

[11] 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 International Workshop on Smalltalk Technologies*, IWST '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[12] 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.

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

[14] 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.

[15] 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.

[16] Lombardi, Carlos and Passerini, Nicolás. Alumnos, docentes y recorridos en una materia de programación informática. UNQ – Bernal, Buenos Aires, Argentina, Oct. 2008.

[17] P. E. M. López, E. A. Bonelli, F. A. Sawady, U. M. de Terramar, and U. K. Le Guin. El nombre verdadero de la programación. Aug. 2012.

[18] D. J. Malan and H. H. Leitner. Scratch for budding computer scientists. *SIGCSE Bull.*, 39(1):223–227, Mar. 2007.

[19] 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.

[20] I. Milne and G. Rowe. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information Technologies*, 7(1):55–66, Mar. 2002.

[21] S. H. Moritz, F. Wei, S. M. Parvez, and G. D. Blank. From objects-first to design-first with multimedia and intelligent tutoring. *SIGCSE Bull.*, 37(3):99–103, June 2005.

[22] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report 64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004.

[23] R. E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1981.

[24] M. Research. Kodu.

[25] E. Roberts and A. Picard. Designing a java graphics library for CS 1. In *ACM SIGCSE Bulletin*, volume 30, pages 213–218. ACM, 1998.

[26] J. Sajaniemi and C. Hu. Teaching programming: Going beyond "objects first".

[27] Spigariol, Lucas and Passerini, Nicolás. Enseñando a programar en la orientación a objetos. UTN FRC, Córdoba, Argentina, Nov. 2013.

[28] T. Vilner, E. Zur, and J. Gal-Ezer. Fundamental concepts of cs1: Procedural vs. object oriented paradigm - a case study. *SIGCSE Bull.*, 39(3):171–175, June 2007.

A. Implementation

Nico ►*Qué podemos decir de esto*◄

B. Images

Imágenes y otros detalles de wollok que no entran en las 6/7 páginas del artículo

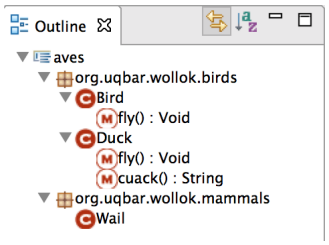

Figure 4. Outline View: muestra un resumen del contenido del archivo

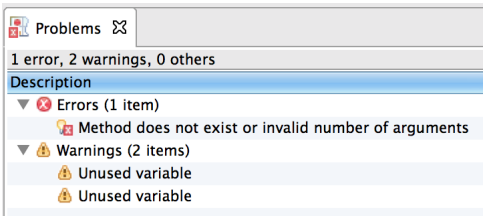

Figure 5. Vista de Problemas: errores y warnings

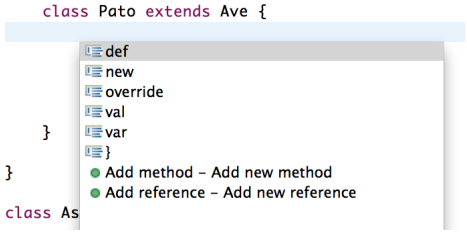

Figure 6. Code Assist: templates para crear código rápidamente

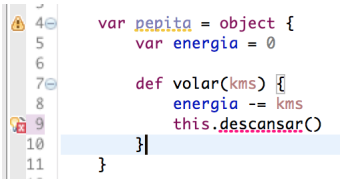

Figure 7. Checkeo de método inexistente en this