Wollok – Relearning How To Teach Object-Oriented Programming

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

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

Abstract

In this context... We consider this problem P... P is a problem because... We propose this solution... Our solution solves P in such and such way.

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++ [4] Nico ►*¿otros?*◄ . This approach has been used even outside the OO world [7, 14, 17]. A step further is to provide a whole programming environment specifically designed to aid novice programmers such as Squeak [9], Traffic [16] and BlueJ [3].

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 [2, 5]. Other languages are focused on teaching to children or teenagers, such as Scratch [15] and Etoys [11]. Finally, there are many approaches which emphasize the importance of *visualization tools* to simplify the understanding of the underlying programming model [6, 18].

Previous works from our team [8, 12, 13, 19] 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 programming 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

One of the causes of the difficulties to learn 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.

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 [**?**]. 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 enviroments in introductory courses [10]. 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 informatic carreer which fail their introductory programming courses, are often likely to give up their studies Nico ►*cite needed*◄ . 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. [12]

Ozono hereda muchos de los problemas de los entornos Smalltalk

Smalltalk, however, lacks other important facilities: no visualisation tools for class relations are available. The main problem with this lies in the Smalltalk language itself: since it is not statically typed, it is not possible to extract usage relations from its source code. No indication exists before runtime as to the call relationships between classes. Inheritance relationships as shown in the browser do not present the relationships of one application but rather the whole Smalltalk environment and so the browser is not used as an application modelling tool. Smalltalk blurs the distinction between the environment and the application under development. Reports about the use of Smalltalk systems for teaching also point to another problem: its size. While the language itself (in terms of the number of constructs) is small, the class library and tools are large and often confusing. Several authors reported difficulties with the students ability to cope with the environment [7, 8], especially that experimentation and self directed learning was not working well because students were overwhelmed by the system. They also found that the functionality of the browser should be limited, since its power and flexibility caused more problems than it solved. [10].

Lo concreto que hicimos es tener un lenguaje con - clases y objetos integrados - un IDE profesional con syntax highlighting, refactors, autocompletion y la vedette (?) inferencia de tipos (en progreso). - bueno, muchas mejoras a nivel lenguaje, como literales para coleciones, imports - una forma fácil de construir tests.

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. Proposed Solution

Wollok is a complete new tool based on the same fundamental ideas that were present in Ozono and LOOP. In particular:

- **Incremental concepts introduction**: references, objects, messages, polymorphism, classes, inheritance.
- **General purpose**: meaning not tied to any specific domain (e.g.: robots)

Besides this we also tried to address some other concerns detected while using Ozono for several years. Here's a list of main cathegories or lines of work in which Wollok extends the previous work

- **Profundizar y pulir el highlighting the conceptos primarios y la estratificacion de conceptos**. (ej: literales de objetos, literales de colecciones. Objetos no como un elemento de la IDE -Ozono: nueva referencia global-, sino como un elemnto del lenguaje. Evita referencias globales.)
- **Introducción de nuevos elementos concretos que explicitan conceptos ya existentes** (ej: 1- var/val, 2- la idea de hacer un effect system power que detecte efecto de lado, y asi poner checkeos para resolver el problema de si un método es una 'orden' o una 'pregunta', 3- program/libreria/test, 4-override).
- **Unificar las fases del aprendizaje** (ej: objetos+clases: un solo lenguaje, misma herramientas, poder reutilizar y hacer convivir)
- **Proveer un entorno inteligente que**: por un lado, estructure en forma más estricta/explícita la experiencia; y que, por el otro lado, permita una gran asistencia al estudiante/desarrollador (esto tiene muchos elementos: 1- desde content assist, 2-syntax coloring, 3- resaltado de errores (sintaxis y tipado) 4-navegación de código, 5-busqueda de referencias, 6-diagramas automáticos de clases, 7-hasta llegar un sistema de tipos que permita la detección temprana de errores, 8-reducir errores frustrantes: se cancela la edicion por tener 1 solo editor de metodo por ves (poder visualizar más que un sólo método simul), evitar errores de imagenes)
- **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.)

3.1 Entorno Inteligente de Trabajo

3.1.1 Visualización y Navegación

// todo:

- syntax highlight
- outline
- hovering
- vista de problems
- navigate: goto (F3, click), flechita para ir al método que sobrescribe.
- find references

3.1.2 Asistencia

//todo:

- content assist
- quick fixes
- code templates (nuevo)

3.1.3 Checkeos y validaciones

Wollok provee numerosos checkeos y validaciones estáticas, a fin de que el alumno pueda encontrar los problemas en su código de manera temprana. Incluso a medida que va escribiendo su código. Esto contribuye a que el alumno mantenga control completo de su código, y evita la sensación de estar perdido.

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

- **De sintaxis**: dados por el parser y lexer automáticamente.
- **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 parámetros.
  - *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.
- **De tipos**: verifican compatibilidad de referencias en base a sus tipos. Por ejemplo ante envío de mensajes, o asignaciones de variables. Basado en el sistema de tipos.

4. Discussion

This work builds on the experience of eight years using our approach in 4 different universities in Argentina[1].

5. Related Works

LOOP is presented as a visual environment to teach OOP using a reduced set of language constructions and a prototype approach to create objects. It presents the main concepts of object, message and reference in a specialized tool with a visual representation of the object environment. Several visual tools to teach programming already exists, like ObjectKarel[2], Scratch[14] and Etoys[4]. ObjectKarel presents a visual tool based on the abstraction of robots to teach OOP, using a map where the robots-the objects- move when messages are sent to them. LOOP does not center on a specific abstraction like a robot: it allows the student to create any other abstraction. Scratch and Etoys, are aimed to teach the basics of programming to children, using visual objects and scripts to play tween objects. This kind of diagrams could be inferred from the evaluation of any piece of code, even the execution of tests. Another subject of research is a "debugger" for the tool [1]. We think that a live and powerful debugger a ' la Smalltalk is a rich tool for the understanding of the whole environment behaviour. After a message is sent, a debugger view can be used like a video player, with play, forward and backward buttons to navigate the message stack and see how the state changes after each message send in the object diagram. Finally, there are some improvements to be made to the user interface, such as shortcuts, code completion, improved menus or internationalization. Currently the tool is only available in spanish, we want to make it configurable to add more languages as neces- sary.

In response to interest in an objects-first approach, several texts and software tools have been published/developed that promote this strategy (such as [1, 12]). Four recent software tools are worthy of mention as using an objects-first approach: BlueJ [9], Java Power Tools [11], Karel J. Robot [2], and various graphics libraries. Interestingly, all these tools have a strong visual/graphical component; to help the novice "see" what an object actually is – to develop good intuitions about objects/object-oriented programming. BlueJ [9] provides an integrated environment in which the user generally starts with a previously defined set of classes. The project structure is presented graphically, in UML-like fashion. The user can create objects and invoke methods on those objects to illustrate their behavior. Java Power Tools (JPT) [11] provides a comprehensive, interactive GUI, consisting of several classes with which the student will work. Students interact with the GUI, and learn about the behaviors of the GUI classes through this interaction. Karel J. Robot [2] uses a microworld with a robot to help students learn about objects. As in Karel [10], Robots are added to a 2-D grid. Methods may be invoked on the robots to move and turn them, and to have the robots handle beepers. Bruce et al. [3] and Roberts [13] use graphics libraries in an object-first approach. Here, there is some sort of canvas onto which objects (e.g. 2-D shapes) are drawn. These objects may have methods invoked on them and they react accordingly. In the remainder of this paper, we present a new tactic and software support for an objects-first strategy. The software support for this new approach is a 3D animation tool. 3D animation assists in providing stronger object visualization and a flexible, meaningful context for helping students to "see" object-oriented concepts. (A more detailed comparison of the above tools with our approach is provided in a later section.) [6]

6. Conclusion

Y a futuro le agregaríamos: - Integración con la UI - Integración fácil con SCM

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. Ideally, we also want to teach our students about the techniques needed for teamwork. To do this, it is essential that the environment has some form of support for group work. [10]

- Más refactors y mejora del sistema de inferencia. - Una versión web / versión liviana con el objetivo de poder ejecutarse en las netbooks que tienen los chicos de secundaria.

Probar la nueva herramienta en entornos educativos.

Eso pensando el lenguaje/herramienta, si pienso a nivel docencia se me ocurre que lo que tenemos que hacer es integrarnos con otras entidades, como Sadosky u otras universidades. Haciendo foco en que el punto no es la herramienta sino que tenemos que repensar cómo enseñamos a programar.

Acknowledgements

References

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

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

[3] J. Bennedsen and C. Schulte. BlueJ visual debugger for learning the execution of object-oriented programs? *Trans.*

[1] 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.

Comput. Educ., 10(2):8:1–8:22, June 2010.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

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

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

A. Images

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

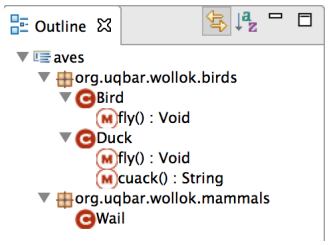

Figure 2. Outline View: muestra un resumen del contenido del archivo

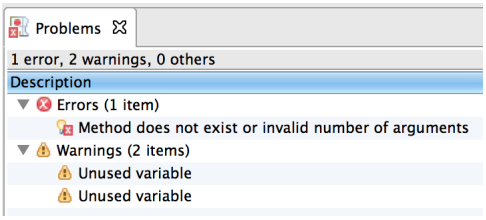

Figure 3. Vista de Problemas: errores y warnings

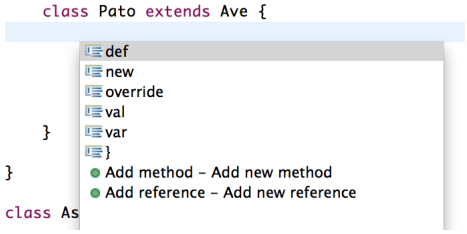

Figure 4. Code Assist: templates para crear código rápidamente

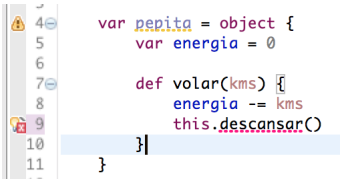

Figure 5. Checkeo de método inexistente en this