Wollok: reconciliando didáctica e industria en un lenguaje educativo para POO

Carlos Lombardi (1,2) - Nicolás Passerini (1,3)
(1) Departamento de Ciencia y Tecnología, Universidad Nacional de Quilmes
(2) Instituto de Ingeniería, Universidad Nacional de Hurlingham
(3) Escuela de Ciencia y Tecnología, Universidad Nacional de San Martín
carlombardi@gmail.com - npasserini@gmail.com

Resumen

Los cursos iniciales sobre programación, y en particular los que introducen programación orientada a objetos (POO), resultan dificultosos para muchos estudiantes como lo indica la literatura. Entre las múltiples propuestas que se proponen para resolver esta problemática para cursos universitarios, la mayoria utiliza herramientas diseñadas para profesionales sin contemplar la introducción de herramientas educativas. Por otro lado, las publicaciones que definen lenguajes de programación educativos resultan aún más escasas.

Este trabajo presenta Wollok, un lenguaje de programación educativo para POO desarrollado por docentes de distintas universidades del conurbano bonaerense y de CABA en el marco de una propuesta pedagógica para la enseñanza inicial de la POO a comienzos del nivel terciario (o a fines del secundario), que pone el foco en la gradualidad para la introducción de los conceptos teóricos.
Wollok permite definir objetos en forma completa con una sintaxis minima sin necesidad de recurrir a los conceptos de clase o prototipado. Además, su sintaxis está pensada para simplificar la transición a lenguajes industriales como Java o JavaScript.

Los resultados obtenidos muestran que el uso de un lenguaje de programación educativo puede resultar ventajoso en cursos universitarios iniciales, sin representar un obstáculo para el uso posterior de lenguajes industriales.

Palabras clave: programación orientada a objetos, enseñanza de programación, lenguajes de programación educativos.

1. Introducción

Aprender a programar resulta, por lo general, una tarea difícil para muchos estudiantes. Este hecho ha sido ampliamente reconocido en la literatura [18,10,3,41,39]. Los repetidos reportes de bajas cifras de aprobación [12,5] constituyen una comprobación de estas dificultades.

En este escenario, el curso que introduce la Programación Orientada a Objetos (POO) en currículas que incluyen programación puede verse afectado por algunos obstáculos comunes a cursos iniciales, y también por otros que son específicos de la POO. Entre los primeros, mencionemos la elevada cantidad de conceptos que intervienen en la construcción de cualquier programa, incluso los más simples, en varios lenguajes o entornos de programación; o la falta de adecuación de herramientas pensadas para profesionales a las necesidades de programadores novatos. Entre los segundos, destacamos la confusión que provoca, en muchos estudiantes, la dualidad entre los conceptos de instancia y de clase [18,9,17]. Estos problemas, que aparecen por lo general al principio de un curso, fueron detectados por el grupo de docentes del que formamos parte a lo largo de nuestra experiencia dictando cursos introductorios sobre POO que siguen a un curso inicial sobre programación.
También hemos sido testigos de un problema relevante (aunque menos presente en la literatura) que se presenta *al final* de cursos iniciales sobre programación: cómo impulsar que los estudiantes aprovechen los conocimientos adquiridos en cursos más avanzados, y (especialmente) en su futuro desarrollo profesional.

Desde hace tiempo está presente en el ámbito de la enseñanza, el debate sobre qué lenguajes de programación resultan más adecuados para el ámbito educativo. En las últimas décadas han aparecido numerosas propuestas al respecto. Aunque la mayor parte apunta al ámbito de la educación pre-terciaria o para cursos iniciales de programación a nivel universitario, también existen varios trabajos que analizan distintas alternativas de lenguajes de programación para la enseñanza inicial de la POO.
En el ámbito universitario, esta temática incluye el debate sobre la conveniencia de utilizar en los cursos los mismos lenguajes que se utilizan en el ámbito profesional del desarrollo de software (a los que llamaremos *lenguajes industriales*), o bien usar lenguajes diseñados específicamente para la enseñanza (que denominaremos *lenguajes educativos*).
En particular, en varias universidades del AMBA han surgido, en los últimos años, propuestas de lenguajes educativos, tanto para un primer curso de programación como para la iniciación en la POO en un segundo curso de programación. Para la iniciación en programación se desarrolló *Gobstones*, que actualmente se utiliza en UNQ y UNAHur. Pensando específicamente en la POO se diseñó e implementó *Wollok*, de uso actualmente en UTN (Regs. Bs.As. y Delta), UNQ, UNSaM, UNDaV y UNAHur; y que ha contado con experiencias de uso en la educación secundaria. En ambos casos, el desarrollo del lenguaje, y también de un entorno de programación asociado, se corresponde con propuestas pedagógicas sobre cómo desarrollar los cursos que utilicen dichas herramientas.

El foco de este trabajo es la enseñanza de POO, y su objeto de estudio el lenguaje Wollok. Este lenguaje, sumado al entorno de programación que lo acompaña, ha resultado un vehículo adecuado para implementar una propuesta de curso que logra altos porcentajes de retención y aprobación, en donde los estudiantes son capaces de aplicar en la práctica los conceptos más relevantes de este paradigma. El lenguaje Wollok fue diseñado teniendo en cuenta la transición a lenguajes industriales, en particular Java y JavaScript. En las experiencias recogidas hasta el momento, los estudiantes no evidenciaron mayores problemas en cursos más avanzados en los que se usan lenguajes industriales.

Esquema En la Sec. 2 repasaremos varias alternativas que aparecen en la literatura, indicando sus fortalezas y debilidades. Luego describiremos el lenguaje Wollok (*cf.* Sec. 3), haciendo breves referencias a la propuesta pedagógica en la que se enmarca su concepción (*cf.* Sec. 3.4) y a la posterior transición hacia lenguajes industriales (*cf.* Sec. 3.5). La Sec. 4 resume los resultados obtenidos y en la Sec. 5 se esbozan algunas conclusiones y líneas de trabajo futuro.

2. Trabajo relacionado – lenguajes en enseñanza de programación

Existen muchas propuestas sobre qué lenguaje usar para introducir conceptos de programación a distintos públicos.
Entre ellas, han aparecido distintos lenguajes *educativos*, desde los iniciales LOGO [29] y Pascal [43], hasta los más recientes Karel [30], Oz [38] y otros. Destacamos el lenguaje educativo Gobstones [24], diseñado y desarrollado en Argentina, hoy se utiliza en UNQ, UNAHur, y al menos una institución de enseñanza secundaria. En trabajos y experiencias más recientes se han propuesto *lenguajes basados en bloques* (BBPL por sus iniciales en inglés). Varias propuestas en este sentido han tenido gran popularidad, en particular Scratch [25] y Alice [7]. Otras opciones incluyen a EToys [23] y a Kodu [36]. Aunque la mayor parte del trabajo sobre BBPL se refiere al uso pre-universitario (ver p.ej. [42]), también existen algunos reportes del uso de este tipo de lenguajes en cursos introductorios a nivel universitario [34].
Sin embargo, la literatura disponible indica que en el nivel universitario, es mayoritario el uso de *lenguajes industriales* en cursos iniciales sobre programación. En una amplia reseña

de 2007 [31], la gran mayoría de los trabajos y casos de ejemplo mencionados se refieren a lenguajes industriales, cuya adaptabilidad al uso educativo es tenida en cuenta raramente (a este respecto, se destacan Python y Scheme); los únicos lenguajes concebidos para su uso didáctico que se mencionan son Logo y Pascal. Esta tendencia se ve confirmada en varios artículos aparecidos recientemente [5,8,33,2]. La adaptación a las necesidades de cursos iniciales se aborda mayoritariamente mediante editores, entornos u otras herramientas de programación.

El escenario es similar respecto de la enseñanza inicial de la POO: la existencia de lenguajes educativos como Karel++ [4], Mama [16] y Blue [21], no impide el uso mayoritario de lenguajes industriales en los cursos universitarios, muchas veces en conjunto con entornos y otras herramientas de propósito educativo[1]. En efecto, una reseña de 2007 sobre herramientas para enseñanza de POO [11] menciona 15 herramientas y entornos que aplican a lenguajes industriales, y sólo 2 lenguajes. Otros trabajos acerca de experiencias [37], propuestas de organización de cursos [13,26,22], análisis de distintos aspectos de un curso inicial sobre POO [40,27], y otras herramientas [35], se basan en lenguajes industriales, mayoritariamente Java, y en algunos casos Python, C++, o Eiffel.
Entre los entornos de propósito educativo más mencionados en la literatura encontramos a BlueJ [22], Dr.Java [1] y Traffic [26].

La fortaleza más extensamente mencionada para el uso de lenguajes industriales es la intención de mantener acotada la divergencia entre la práctica universitaria de los estudiantes y su futuro desarrollo como profesionales, incluyendo la percepción de los mismos estudiantes al respecto [32].

1 Una variante es la definición de variantes de lenguajes industriales que permiten evitar, en las primeras etapas de un curso, la exposición a características que podrían resultar confusas para los estudiantes. Tal es el caso de ProfessorJ [14], Beanshell [28] y Ozono [15].

Del otro lado, su uso en ámbitos educativos ha recibido varias críticas. Una de las más señaladas es la gran cantidad de conceptos o detalles sintácticos necesarios para construir, y ejecutar, el programa más simple, p.ej. en Java [19,1].
Adicionalmente, la sintaxis de muchos lenguajes industriales presenta aspectos que, siendo útiles en el ámbito profesional, pueden resultar confusos en el uso educativo. Un ejemplo es el carácter opcional de `this` al enviar un mensaje en Java. Otro es la existencia de distintas formas de expresar un mismo concepto u operación de distintas formas, o variantes de una misma operación con diferencias sutiles como la distinción entre `++x` y `x++` p.ej. en C.
Finalmente, la sintaxis de estos lenguajes también incluye ambigüedades. En Java, podemos señalar la polisemia de la palabra clave `static`, y que para establecer una relación de subtipado debe usarse `implements` (si el supertipo es una interface) o `extends` (si el supertipo es una clase abstracta) [6].

Como ya mencionamos, estas críticas no impiden la tendencia al uso de lenguajes industriales. El caso de Michael Kölling es significativo a este respecto: luego de haber creado el lenguaje educativo Blue, decidió dejarlo de lado para basar el entorno BlueJ en Java. Su motivación fue el convencimiento que ese paso contribuiría a la mayor difusión de la herramienta [20], lo que refleja la tendencia mencionada.

3. El lenguaje Wollok

Wollok fue concebido en la Argentina, como un producto de propósito educativo que reúne un lenguaje de programación y un IDE. Comenzó a usarse en 2015. Actualmente se utiliza en UTN (Regs. Buenos Aires y Delta), UNSaM, UNQ, UNDaV y UNAHur, con un total aproximado de 25 cursos y 800 alumnos por año.
El objetivo del desarrollo de Wollok es contar con una herramienta de soporte adecuada para una estructura didáctica de un curso inicial

sobre POO, que viene siendo desarrollada por un grupo de docentes de distintas universidades del AMBA desde 2005. Esta propuesta surge a partir de haber observado en la experiencia docente, las dificultades de aprendizaje y de aprovechamiento de los conocimientos adquiridos que se describen en la Sección 1 junto con otras, en especial el poco aprovechamiento del *polimorfismo* en la construcción de programas.
A continuación presentamos algunos aspectos relevantes de la sintaxis de Wollok. Luego describiremos brevemente la propuesta pedagógica en cuyo marco se concibió el lenguaje, señalando cómo este contribuye a que dicha propuesta pueda ser implementada exitosamente.

3.1. Objetos autodefinidos

La sintaxis de Wollok admite *objetos autodefinidos*, esto es, declaraciones de objetos que incluyen todos sus atributos y métodos. Nos referiremos a ellos como objetos bien conocidos o WKO (por las siglas en inglés) porque son globalmente accesibles. El siguiente extracto de código define un objeto que representa un modelo básico de un ave:

```
object pepita {
  var energia = 0
  method energia() {
    return energia
  }
  method dobleDeLaEnergia() {
    return energia * 2
  }
  method volar(kms) {
    energia -= kms + 10
  }
  method comer(gramos) {
    energia += gramos * 4
  }
}
```

Esta definición es completamente funcional. El entorno Wollok incluye una consola del estilo REPL (Read-Evaluate-Print Loop) que permite interactuar con el objeto así definido.

```
>>> pepita.comer(20)
>>> pepita.energia()
80
>>> pepita.dobleDeLaEnergia()
160
```

Esta sintaxis permite la definición completa y funcional de un objeto, a partir de un mínimo de conceptos: objeto, variable, método, parámetro/argumento, valor de retorno. Para estudiantes que hicieron un curso básico de programación, el único concepto novedoso es el de objeto; todos los otros son familiares, si asimilamos método a procedimiento o función.
La sintaxis del envío de mensajes coincide con la utilizada en p.ej. Java, C++, C\#, JavaScript, Scala y Python:

```
objeto.mensaje(arg1, arg2, ...)
```

La sintaxis para la alternativa también es standard:

```
if (condicion) {
  sentencias
} else if (condicion) {
  sentencias
} else {
  sentencias
}
```

Estas decisiones contribuyen a simplificar la transición a lenguajes industriales. Un aspecto que contribuye notablemente a mantener sencilla la sintaxis de Wollok es la ausencia de anotaciones de tipo. Al mismo tiempo, Wollok incluye un sistema de *inferencia de tipos*, que permite la detección de algunos errores que suelen cometer los estudiantes, manteniendo una sintaxis mínima. De esta forma, se combina una introducción sencilla del *polimorfismo* que no requiere explicitar las relaciones de subtipado[2], con la detección de problemas en forma estática. Un ejemplo usado frecuentemente implica la introducción de un entrenador, que puede entrenar cualquier ave que se le indique. Una implementación inicial tiene esta forma:

2 El sistema de inferencia usa *tipos estructurales*.

```
object roque {
  var pupilo
  method pupilo() {
    return pupilo
  }
  method tuPupiloEs(ave) {
    ave = pupilo
  }
  method entrenar() {
    pupilo.volar(10)
    pupilo.comer(80)
    pupilo.volar(20)
  }
}
```

Típicamente, agregamos al modelo otros objetos que representan aves, que entienden los mensajes `comer` y `volar`. Por ejemplo:

```
object pipa {
  var totalComido = 0
  var totalVolado = 0
  method viajoMucho() {
    return totalVolado > 500
  }
  method volar(kms) {
    totalVolado += kms
  }
  method comer(gramos) {
    totalComido += gramos
  }
}
```

El comportamiento polimórfico de las aves definidas se puede mostrar usando el REPL:

```
>>> roque.tuPupiloEs(pepita)
>>> roque.entrenar()
>>> roque.tuPupiloEs(pipa)
>>> roque.entrenar()
```

donde se pueden intercalar consultas a `pepita` y a `pipa`, para mostrar cuál cambió luego de cada llamada a `roque.entrenar()`.

Wollok incluye operaciones con números con notación `objeto.mensaje`, p.ej.

```
>>> 4.min(9)
4
```

lo que permite señalar que todos los valores involucrados en un programa son objetos, cuya funcionalidad se accede mediante mensajes.
A partir de este señalamiento, se puede indicar que los atributos de un objeto son referencias a otros objetos. Así, el atributo `energia` en `pepita` es una referencia a un número, mientras que el atributo `pupilo` en `roque` es una referencia a un ave, que puede ser `pepita`, `pipa`, o cualquiera *que se defina a posteriori* mientras resulte polimórfica con las anteriores.

3.2. Colecciones

El lenguaje Wollok admite literales para listas, p.ej. `[1,85,394,31]`, y para conjuntos, p.ej. `#{1,85,394,31}`. Las listas y los conjuntos tienen interface de objetos; p.ej. el tamaño de la lista indicada puede obtenerse así:

```
>>> [1,85,394,31].size()
4
```

Las listas y los conjuntos son instancias de clases incluidas en la biblioteca básica Wollok, `List` y `Set` respectivamente. La inclusión de literales que permiten construir listas y conjuntos sin necesidad de una instanciación explícita, permite trabajar con colecciones antes de introducir la noción de clase.
Los literales de colecciones no están limitados a números, ni a otros literales o constantes; la siguiente es una expresión válida:

```
[pepita,roque.pupilo(),8.max(15)]
```

Wollok también incluye literales para *funciones de primera clase* (*i.e.* funciones que pueden ser manipuladas como valores), que son objetos que entienden el mensaje `apply`:

```
>>> { n => n * 2 }.apply(5)
10
```

Esta característica facilita la inclusión de una interface de estilo funcional para las colecciones, en las que los recorridos se resuelven con mensajes que toman una función como pa-

rámetro. P.ej., las colecciones de Wollok entienden `map` y `filter` con la semántica usual.

```
>>> [1,85,394,31].map({n => n*2})
[2,170,788,62]
>>> [1,85,394,31].filter(
      {n => n.between(10,100)})
[85,31]
```

Nuestra experiencia indica que los estudiantes se acostumbran rápidamente a esta forma de manipular colecciones, lo que minimiza en gran medida el uso de estructuras de repetición. A su vez, el uso de una interface declarativa permite un abordaje de más alto nivel de los problemas que involucran manipulación de colecciones: el foco puede estar puesto en la naturaleza de la manipulación deseada, en lugar de concentrarse en los detalles algorítmicos del recorrido.

3.3. Clases

La sintaxis de Wollok está diseñada de forma que una definición de WKO pueda convertirse en una clase mediante una modificación mínima. Por ejemplo el WKO `pepita` definido en la Sección 3.1 puede transformarse en una clase, simplemente reemplazando la expresión `object pepita` por `class Golondrina`, de la siguiente manera:

```
class Golondrina {...}
const pepita = new Golondrina()
```

La simplicidad en la transición se refuerza porque los objetos instanciados a partir de clases (en adelante CBOs) son totalmente compatibles con los WKO. No sólo conviven en el mis-mo programa sino es habitual que un WKO y un CBO puedan ser utilizados polimórficamente.
En Wollok no es necesario definir *constructores*. Para los casos en los que es necesario parametrizar la creación del nuevo objeto, la expresión de instanciación admite *parámetros nominales* que se asignan automáticamente a los atributos del nuevo objeto. Por ejemplo, para crear una nueva Golondrina con una energía inicial de 100, se puede usar la expresión:

```
new Golondrina(energia = 100)
```

Si bien este modelo no abarca la totalidad de las posibilidades que brindan los constructores *à la* Java o C\#, son muy raros los programas en un curso inicial de objetos que puedan requerir toda esa potencia. A cambio, el código resultante es mucho más breve, evitando el *boilerplate* habitual de la inicialización mediante constructores y simplificando el metamodelo del lenguaje a ser aprehendido.
El metamodelo completo incluye *herencia simple*, que se introduce mediante la palabra clave `inherits`. Tanto las clases como los WKOs pueden heredar de otra clase. Al heredar un WKO de una clase también es posible inicializar atributos en valores determinados. Por ejemplo:

```
class Golondrina inherits Ave {
  ...
}

object pepita inherits Golondrina
(energia = 100) {...}
```

3.4. Una estrategia pedagógica bien soportada por Wollok

Como se describió anteriormente, Wollok surgió a partir de la concepción de una estrategia pedagógica para un primer curso sobre POO, que permitiera evitar, o bien tratar correctamente, varias dificultades detectadas por un grupo de docentes de distintas universidades.
La estrategia se basa en la dosificación de los conceptos que se van introduciendo a lo largo de la cursada, definiéndose cuatro etapas incrementales. La primera etapa se basa en los *objetos auto-definidos* descriptos en la sección 3.1. Este tipo de definiciones resulta suficiente para delinear un metamodelo completo a partir de los conceptos de objeto, mensaje, método, atributo y referencia; que permite estudiar el polimorfismo y las relaciones dinámicas entre objetos sin necesidad de agregar otros conceptos básicos; en particular, no se mencionan en esta etapa clase, prototipado ni instanciación o clonación. En la segunda etapa se incorporan

colecciones en conjunto con *funciones de primera clase*. Esto permite trabajar con las interfaces funcionales de colecciones descriptas en la sección 3.2. Recién en la tercera etapa se introducen las ideas de *clase* e *instanciación*. Finalmente, la cuarta etapa introduce la *herencia de clases* y otros conceptos asociados, como la sobre-escritura de métodos y la idea de *template method*.

Varias características sintácticas del lenguaje Wollok lo convierten en un vehículo válido para esta estrategia. Las cuatro etapas se corresponden con subconjuntos crecientes del lenguaje. El sublenguaje correspondiente a cada etapa es completo y coherente, evitándose así la sensación de trabajar con un lenguaje "parcial" que será completado sólo más adelante. Asimismo, todos los elementos sintácticos introducidos en una etapa siguen siendo utilizados hasta el final del curso. Esta continuidad ayuda a que la transición entre etapas sea suave, lo que también está contemplado por la similitud sintáctica entre las definiciones de un WKO y una clase, descripta en la sección 3.3.

3.5. Transición a lenguajes industriales

El lenguaje Wollok fue pensado para minimizar los problemas que pudieran aparecer en la transición hacia lenguajes industriales; su estructura sintáctica es muy similar a lenguajes muy populares como Java o Javascript. Para brindar un ejemplo, transcribimos una versión minimal en Wollok, con sólo dos métodos, de la clase `Golondrina` introducida en la Sección 3.

```
class Golondrina {
  var energia = 0
  method energia() {
    return energia;
  }
  method volar(kms) {
    energia -= kms + 10;
  }
}
```

Para traducir esta definición al lenguaje Java, alcanza con agregar anotaciones de tipo y visibilidad (que en algunos casos reemplazan a las palabras clave `var` o `method`), junto con el punto y coma al final de cada sentencia; y diferenciar los nombres de atributos y métodos. Transcribimos la traducción de la clase `Golondrina` a Java, subrayando las palabras que se agregan, e indicando con un subrayado ondulado las que se modifican.

```
public class Golondrina {
  private int energia = 0
  public int getEnergia() {
    return energia;
  }
  public void volar(int kms) {
    energia -= kms + 10;
  }
}
```

Por otro lado, una transformación a Javascript requiere: la eliminación de las palabras clave `var`} y `method`; la diferenciación de los nombres de atributos y métodos al igual que en Java; el agregado de un constructor para inicializar las variables; y el de `this.` cada vez que se accede a un atributo.

```
class Golondrina {
  constructor() {
    this.energia = 0
  }
  getEnergia() {
    return this.energia;
  }
  volar(kms) {
    this.energia -= kms + 10;
  }
}
```

Los tres lenguajes comparten una misma estructura para definir una clase (salvo el constructor en JavaScript); las diferencias se limitan a variantes en la definición de cada elemento.

4. Resultados

De acuerdo a la experiencia acumulada hasta el momento, el lenguaje Wollok, en conjunto con el IDE asociado, han resultado adecuados para el propósito con el que fueron creados: acompañar la estrategia pedagógica descripta en la sección 3.4.
En efecto, los objetos autodefinidos permiten postergar la introducción de clases hasta bastante avanzado el curso, la ausencia de anotaciones de tipo simplifca el tratamiento del polimorfismo, los literales de listas y conjuntos permiten una introducción sencilla a las colecciones, y las funciones de primera clase permiten trabajar con la interface funcional de colecciones como lo muestra el uso de `map` y `filter` en la sección 3. En conjunto, estas características permiten una introducción gradual de los conceptos de la POO, en donde en particular el polimorfismo se introduce muy tempranamente, y se puede trabajar con referencias y el diagrama de objetos antes de introducir instanciación.
Destacamos que ninguno de los entornos o lenguajes descriptos en la sección 2 permiten objetos autodefinidos. A la vez en la mayor parte de estas propuestas, basadas en Java o C++, el manejo de colecciones resulta significativamente más complejo que el que presenta Wollok.

El *rendimiento académico* de los cursos que implementan la propuesta pedagógica soportada por Wollok es un indicador de la pertinencia de la misma. Al respecto, se reunió información sobre cursos iniciales sobre POO en UNSaM, UNQ y UNAHur. La materia pertenece al segundo cuatrimestre en UNQ y UNAHur, y al tercero en UNSaM, en todos los casos de carreras de informática. Los datos relevados corresponden a 14 cursos con un total de 327 estudiantes inscriptos[3], de 2015 a 2018. La cantidad de estudiantes que aprobaron es 247, o sea el 75,5%. Si consideramos sólo los estudiantes que rindieron al menos una de las dos instancias de evaluación, el universo se reduce a 296 estudiantes, con una tasa de aprobación del 83,4%.
En las universidades incluidas en el estudio de la tasa de aprobación, no se cuenta con información confiable sobre cursos anteriores de las mismas materias, o bien no hubo tales cursos; de ahí que no puedan establecerse comparaciones. En otras universidades donde la propuesta que involucra a Wollok es utilizada, no se cuenta con datos sólidos. Sólo podemos reportar estimaciones informales indicadas por profesores, que estiman una tasa de aprobación de entre 70% y 90%, contra un rango de 30% a 40% antes de aplicarse este enfoque[4].

Para medir la *percepción de los estudiantes* respecto del lenguaje, realizamos una encuesta voluntaria en cursos que utilizan Wollok en varias universidades a fines de 2017, obteniendo 134 respuestas, y a fines de 2018 con un cuestionario extendido; esta vez con 104 respuestas. La encuesta consistió mayormente en preguntas Likert de cinco opciones: dos positivas ("Absolutamente" y "Bastante"), dos negativas ("Un poco" y "Para nada") y alguna variante de "No sé" o "No tengo opinión".
Un análisis de las respuestas obtenidas indica que los estudiantes valoran a Wollok como una herramienta que no presenta dificultades *per se*, y útil para el desarrollo del curso. Respecto de la afirmación "El lenguaje resultó más fácil de lo que esperaba" se obtuvo un 81% de respuestas positivas, mientras que "El lenguaje ayuda a comprender los conceptos que se enseñan en el curso" obtuvo un 93%. Destacamos que la valoración que hacen los estudiantes *respecto de su desarrollo futuro* también es positiva, a pesar de que Wollok no es un lenguaje industrial: la afirmación "Haber aprendido el lenguaje me va a ayudar en mi práctica profesional" obtuvo un 65\% de respuestas positivas.

3 excluimos a quienes abandonaron el curso muy tempranamente o debido a razones personales

4 Las bajas tasas de aprobación son comunes en cursos iniciales sobre programación: p.ej., [5] menciona para una materia inicial de programación con 377 inscriptos, una tasa de aprobación de 20%, o 32% tomando sólo los estudiantes que rindieron el examen final.

5. Conclusiones y trabajo futuro

En este trabajo presentamos un lenguaje de programación educativo con varias características que facilitan una estrategia en la que los conceptos principales de la POO se introducen en forma gradual, habilitando una experimentación más ágil y una mayor comprensión por parte del alumnado. Esta experiencia indica que contar con un lenguaje de programación que se corresponda con cómo se concibe una determinada secuencia didáctica, puede resultar relevante para el éxito de la misma.
Al mismo tiempo, la sintaxis de Wollok ha sido pensada para facilitar la transición hacia lenguajes industriales, en particular Java (o JavaScript). Los resultados obtenidos sugieren que el uso de un lenguaje educativo no resulta perjudicial, ni en la percepción de los estudiantes sobre la utilidad del lenguaje aprendido, ni respecto del rendimiento en cursos posteriores basados en lenguajes industriales.
Como trabajo futuro, mencionamos varias acciones que propenden a la extensión del uso de la propuesta pedagógica a otras instituciones, como la publicación de actividades de clase y otros materiales, y el desarrollo de una versión Web que haga innecesaria la instalación.

Referencias

[1] E. Allen, R. Cartwright, and B. Stoler. Drjava: A lightweight pedagogic environment for java. In *ACM SIGCSE Bulletin*, volume 34, pages 137–141. ACM, 2002.

[2] A. Bart, A. Sarver, M. Friend, and L. Cox II. Pythonsneks: An open-source, instructionally-designed introductory curriculum with action-design research. In *SIGCSE '19*, pages 307–313, New York, NY, USA, 2019. ACM.

[3] J. Bennedsen and M. Caspersen. Teaching object-oriented programming – towards teaching a syste- matic programming process. In *Eighth Workshopon Pedagogies and Tools for the Teaching and Lear- ning of Object Oriented Concepts - ECOOP 2004*.

[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, USA, 1st edition, 1996.

[5] M. Blesa, A. Duch, J. Gabarró, J. Petit, and M. Serna. Continuous assessment in the evolution of a cs1 course: The pass rate/workload ratio. In *Computer Supported Education*, pages 313–332, Cham, 2016. Springer.

[6] P. Burton. Kinds of language, kinds of learning. *Sigplan Notices*, 33:53–61, 1998.

[7] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.

[8] P. Denny, A. Luxton-Reilly, M. Craig and A. Petersen. Improving complex task performance using a sequence of simple practice tasks. In *ITiCSE 2018*, pp 4–9, New York, USA, ACM.

[9] A. Eckerdal and M. Thuné. Novice java program- mers' conceptions of "object" and "class", and va- riation theory. In *ITiCSE '05*, pages 89–93, New York, USA, 2005. ACM.

[10] V. Efopoulos, V. Dagdilelis, G. Evangelidis and M. Satratzemi. Wipe: A programming environment for novices. *SIGCSE Bull.*, 37(3):113–117, 2005.

[11] S. Georgantaki and S. Retalis. Using educational tools for teaching object oriented design and programming. *Journal of Information Technology Impact*, 7:111–130, 01 2007.

[12] M. Giannakos, I. Pappas, L. Jaccheri, and D. Sampson. Understanding student retention in computer science education: The role of environment, gains, barriers and usefulness. *Education and Information Technologies*, 22(5):2365–2382, 2017.

[13] M. H. Goldwasser and D. Letscher. Teaching an object-oriented cs1 - with python. In *ITiCSE '08*, pages 42–46, New York, NY, USA, 2008. ACM.

[14] K. Gray and M. Flatt. Professorj: a gradual intro- duction to java through language levels. In *Compa- nion of 18th OOPSLA*, pages 170–177. ACM, 2003.

[15] C. Griggio, G. Leiva, G. Polito, G. Decuzzi, and N. Passerini. A programming environment supporting a prototype-based introduction to OOP. In *IWST '11*, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[16] A. Harrison Pierce. Mama-an educational 3d programming language.

[17] S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. In *SIGCSE '97*, pages 131–134, New York, USA, 1997. ACM.

[18] 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.

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

[20] M. Kölling. Lessons from the design of three educational programming environments: Blue, bluej and greenfoot. *International Journal of People- Oriented Programming (IJPOP)*, 4:5–32, 07 2016.

[21] M. Kölling and J. Rosenberg. Blue - a language for teaching object-oriented programming. volume 28, pages 190–194, 03 1996.

[22] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

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

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

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

[26] 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, 2003.

[27] C. S. Miller and A. Settle. Some trouble with transparency: An analysis of student errors with object- oriented python. In *ICER '16*, pages 133–141, New York, NY, USA, 2016. ACM.

[28] P. Niemeyer and D. Leuck. Beanshell—lightweight scripting for java, 2003.

[29] S. Papert. Teaching children thinking. *Innovations in Education and Training Intl.*, 5, 09 1972.

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

[31] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *ITiCSE-WGR '07*, pages 204–223, New York, NY, USA, 2007. ACM.

[32] A. Pears, S. Seidman, L. Malmi, L.Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *ACM sigcse bulletin*, volume 39, pp 204–223. ACM, 2007.

[33] V. Phan and E. Hicks. Code4brownies: An active learning solution for teaching programming and problem solving in the classroom. In *ITiCSE 2018*, pp 153–158, New York, USA. ACM.

[34] T. W. Price and T. Barnes. Comparing textual and block interfaces in a novice programming environment. In *ICER 2015*, pages 91–99. ACM, 2015.

[35] D. Radoševic, T. Orehovacki, and A. Lovrencic. Verificator: Educational tool for learning programming. *Informatics in Education*, 8(2):261–280, 2009.

[36] M. Research. Kodu.

[37] A. W. Schmolitzky and T. Göttel. Guess my object: An 'objects first' game on objects' behavior and implementation with bluej. In *ITiCSE'14*, pp 219–224, New York, USA,ACM.

[38] G. Smolka. The oz programming model. In *JELIA*, volume 1126 of *Lecture Notes in Computer Scien- ce*, page 251. Springer, 1996.

[39] J. Sánchez-García, M. Urías-Ruiz, and B. Gutiérrez-Herrera. Análisis de los problemas de aprendizaje de la programación orientada a objetos. *Ra Ximhai*, 11(4):148–175, 2015.

[40] E. Tempero, P. Denny, A. Luxton-Reilly, and P. Ralph. Objects count so count objects! In *ICER '18*, pages 187–195, New York, NY, USA, 2018. ACM.

[41] M. P. Uysal. The effects of objects-first and objects-late methods on achievements of oop learners.

[42] D. Weintrop and U. Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*, pages 199–208. ACM, 2015.

[43] N. Wirth. The programming language pascal. *Acta Inf.*, 1:35–63, 1971.