UNIVERSITAT POLITÈCNICA DE VALÈNCIA

IN-RED 2016
Congreso Nacional de Innovación Educativa y de Docencia en Red

Congreso In-Red 2016
UPV, 7 y 8 de julio de 2016
Doi: http://dx.doi.org/10.4995/INRED2016.2016.4326

# Corrección Semi-Automática de Programas Java*

**David Insa y Josep Silva**

Universitat Politècnica de València

***Abstract***

*La evaluación es una parte fundamental de la enseñanza. Permite tanto a los estudiantes como a los profesores medir el grado de éxito obtenido en el proceso de aprendizaje. De hecho, la evaluación no debería usarse solo para puntuar, sino que debe ser parte integral de la educación, y proporcionar a los alumnos retroalimentación inmediata. Obviamente, la evaluación debe ser objetiva y justa, y tratar a todos los alumnos por igual. Esto ocurre, p.e., en exámenes de respuesta múltiple, pero, desafortunadamente, no está asegurado en la evaluación de código programado por alumnos, puesto que su corrección es todavía un proceso manual y propenso a errores e interpretaciones. En este artículo proponemos un sistema semi-automático de evaluación de código que utiliza técnicas de caja negra (basadas en* output-comparison*) y técnicas de caja blanca (que observan las propiedades internas del código). El método propuesto incluye nuevas ideas y técnicas que le permiten evaluar incluso código que no compila.*

***Keywords****: Evaluación, Java.*

## 1 Introducción

La evaluación es una parte fundamental de la educación, ya que determina si el alumno ha cumplido los objetivos propuestos. Cuanto mejor funcione la evaluación, mejor retroalimentación reciben tanto el alumno como el profesor. Por lo tanto, esta proporciona un medio para guiar al alumno a su vez que proporciona al alumno y al profesor información esencial sobre el proceso de aprendizaje.

*Proyecto financiado por EU (FEDER) y el Ministerio de Economía y Competitividad (TIN2013-44742-C4-1-R), por la *Generalitat Valenciana* (PROMETEO-II/2015/013), y por la Universitat Politècnica de València (PIME B16). Los autores agradecen el respaldo de la acción COST IC1405.

En el área de la educación psicológica, se ha demostrado que la mayoría de alumnos dirigen sus estudios hacia lo que se evalúa en la asignatura y cómo esto afecta a su nota final (véase, p.e., Biggs y Tang 2007, [Capítulo 9]). Como consecuencia, realizar una evaluación continua durante el curso puede ayudar a dirigir y mejorar el proceso de aprendizaje. Sin embargo, asegurar manualmente una evaluación de calidad para incluso una pequeña clase requiere de un gran esfuerzo. Cuanto más grande sea la clase, más debe limitarse la cantidad de trabajo de evaluación o distribuirse de alguna otra forma.

Esta es la razón por la cual se han dedicado muchos esfuerzos para proporcionar herramientas y técnicas para la evaluación automática (EA). De hecho, esto ha sido un área de interés durante mucho tiempo [Ihantola y col. 2010], y en la actualidad el profesor puede utilizar diversas formas para definir tests, políticas de reenvío de ejercicios, temas de seguridad, etc.

La EA ha resultado bastante útil en los exámenes tipo test. De hecho, la puntuación automática de los exámenes tipo test es de gran interés en asignaturas universitarias que tienen un gran número de alumnos. Por esta razón, ha sido sistemáticamente implantada en casi todas las universidades. Casi todo el trabajo relacionado con la EA se ha enfocado en esta área, mejorando principalmente el proceso de escaneo, permitiendo el reconocimiento de las anotaciones proporcionadas por los alumnos (p.e., permitiendo modificar la respuesta marcando el círculo que indica que se ha cometido un error e indicando a su lado cuál es la respuesta correcta), o permitiendo el reconocimiento automático de un conjunto limitado de respuestas escritas a mano, minimizando de esta forma la intervención humana.

Otra área de especial interés es la corrección de programas escritos por los alumnos, puesto que la evaluación manual de código fuente es una tarea que consume mucho tiempo y es bastante propensa a errores. Incluso si los criterios de evaluación son exhaustivos, y los profesores tienen casos de test listos para comprobar el código fuente, encontrar errores puede ser muy difícil, especialmente para proyectos de gran envergadura, puesto que el profesor tiene que inspeccionar manualmente los diferentes módulos y ficheros. Además, si un alumno entregase un ejercicio cuyo código no compile, entonces los casos de test no pueden utilizarse para ayudar a evaluar el ejercicio. Existen diversas otras razones por las cuales la evaluación automática de código no se ha generalizado en las asignaturas universitarias. Una razón importante es que la mayoría de sistemas existentes se basan en *output comparison* contra un `gold standard`.[1] Si la salida es la esperada, el código es correcto. Si no, el código se reporta como erróneo, incluso si solo hubiera un error tipográfico. Además, nunca se profundiza en por qué es erróneo. En general, las herramientas de evaluación tratan el código como si fuera una caja negra, y solo evalúan aquellos comportamientos observables desde el exterior.

[1] Debido a su uso extendido, usamos estos anglicismos. Output comparison son aquellas técnicas de testeo basadas en la comparación de la salida, es decir, de caja negra. Gold standard hace referencia a una solución perfecta a un problema. Esta solución se usa para evaluar otras aproximaciones a la solución.

La mayoría del trabajo realizado se ha centrado en programas restringidos, donde para construir un programa el alumno tiene que elegir entre distintas (pero finitas) opciones mostradas a través de una IGU, o en *output comparison*. La mayoría de sistemas avanzados de EA compilan el código del alumno, lo ejecutan, y comparan el resultado con la respuesta correcta. En estos casos, la generación aleatoria de casos de test puede utilizarse para reducir el número de falsos positivos.

Desafortunadamente, comprobar la salida del código del alumno, incluso si comparamos todas las posibles salidas, normalmente no es suficiente para asegurar una evaluación de calidad. Esto queda de manifiesto en el Ejemplo 1.1.

**Ejemplo 1.1** *Un ejercicio de Java dice lo siguiente:* Impleméntese la clase *Alumno*. Los alumnos deben tener tres atributos `DNI`, `carrera`, y `curso` que identifican al alumno y determinan en qué carrera y en qué año lectivo se encuentra, además de un atributo `nota` que puede ser accedido mediante el método `getNota()`. También deben implementar el método `esExcelente()` el cual devuelve si el alumno tiene un expediente excelente. *Se puede asumir que este enunciado forma parte de un modelo más amplio para representar la estructura de una universidad.*

*Solución 1*

```
class Alumno {

    String DNI, carrera, curso;
    double nota;

    Alumno(String DNI, String carrera,
        String curso, double nota) {
        this.DNI = DNI;
        this.carrera = carrera;
        this.curso = curso;
        this.nota = nota;
    }
    double getNota() {
        return this.nota;
    }
    boolean esExcelente() {
        return this.getNota() >= 9;
    }
}
```

*Solución 2*

```
class Alumno extends Persona
implements Calificaciones {
    String carrera, curso;
    double nota;

    Alumno(String DNI, String carrera,
        String curso, double nota) {
        super(DNI);
        this.carrera = carrera;
        this.curso = curso;
        this.nota = nota;
    }
    double getNota() {
        return this.nota;
    }
    public boolean esExcelente() {
        return this.getNota() >= 9;
    }
}
```

*A pesar de que ambas soluciones pueden ser aceptables para un alumno de primer curso, para un alumno de segundo curso la primera solución es inaceptable. La solución 2 es más mantenible a la vez que reutiliza código a través de la herencia. La solución 1 en su lugar no utiliza la herencia, y por lo tanto puede ser erróneo cuando un alumno debe comportarse como una persona. Sin embargo, ambos alumnos tienen los mismos atributos y métodos, y por lo tanto para la mayoría de tests siempre producirán la misma salida. Además, la solución 2 implementa una interfaz, por lo que el resto de objetos del sistema saben a priori que cualquier alumno implementa un método para saber si su expediente es excelente o no. En este ejemplo, un profesor no estaría interesado en la salida, sino en las propiedades especificas de la clase* Alumno *(p.e., extiende de* Persona*, utiliza la herencia a través de* `super()`*, e implementa la interfaz* Calificaciones*).*

Otra restricción importante del *output comparison* es que devuelven el código entero como correcto o incorrecto. No es posible asignar una nota intermedia. La realidad es sin embargo muy diferente: los profesores normalmente dividen el ejercicio en pequeñas piezas puntuables. Por ejemplo, usar la herencia podría tener una nota de 3 sobre 10, y utilizar interfaces una nota de 1 sobre 10, por lo tanto la solución 1 debería tener una nota de 6 sobre 10.

Además, el *output comparison* es muy sensible a pequeños errores. Por ejemplo, si el alumno accidentalmente escribiese `this.getNta` en lugar de `this.getNota`, entonces cualquier método basado en *output comparison* asignaría un 0 al ejercicio. No obstante, la mayoría de profesores considerarían este error tipográfico como insignificante y le pondrían al ejercicio un, p.e., 9 sobre 10. Por supuesto, los métodos basados en *output comparison* no pueden manejar programas que no compilen (p.e., debido a un punto y coma no escrito).

### 1.1 Contribuciones

En este trabajo introducimos un nuevo método de evaluación de código que va más allá del *output comparison*, y puede evaluar cualquier código (es decir, el alumno puede utilizar todo el lenguaje de programación sin ningún tipo de restricción). En particular, nuestra técnica puede evaluar automáticamente el código no solo a partir de la salida final, sino también comprobando que el propio código fuente cumple aquellas propiedades definidas por el profesor (p.e., una jerarquía particular de clases, implementación de interfaces, existencia de atributos concretos, etc.), permitiendo así puntuar el código incluso si este estuviera parcialmente correcto. Todo ello ha sido implementado en una herramienta semi-automática de corrección.

Es importante destacar que calificamos a nuestra herramienta de *semi-automática* a pesar de ser tan automática como lo son el resto de herramientas disponibles que aseguran que lo son. Queremos remarcar que no todos los ejercicios de programación pueden ser automáticamente evaluados. En general, la EA es un problema indecidible, y por lo tanto no puede existir un sistema de evaluación completamente automático. Por ejemplo, un programa que no termina no puede producir ninguna salida, y en consecuencia, debe pararse manualmente. En este programa, no es posible saber si la ejecución realmente no iba a terminar o si por el contrario el código era simplemente ineficiente, ya que determinar la terminación es ya de por sí un problema indecidible. Otros programas simplemente no compilan, son sintácticamente incorrectos, y por lo tanto no se les puede realizar ningún análisis dinámico. Por todos estos problemas, la intervención humana (el profesor) es necesaria para corregir un programa o para asignarle una nota.

Nuestro sistema, llamado ASys, también realiza *output comparison*. Automáticamente compara la salida del ejercicio con la solución esperada. También utiliza generación de casos de test para comparar la salida obtenida al introducir diferentes entradas. Pero, como nueva y principal característica, también valida aquellas propiedades del código que se deseen definir. Tan pronto como la intervención humana sea necesaria,

el sistema muestra al profesor la información disponible que describe el problema encontrado, junto al ejercicio del alumno y la solución al problema.

La principal contribución de nuestro trabajo consiste en una herramienta semi-automática de evaluación para exámenes y ejercicios en Java basado en (i) *output comparison* via generación de casos de test, y (ii) verificación de propiedades. Además, otra característica importante de nuestra herramienta es que no ha sido diseñada como una herramienta para una asignatura de Java especifica y para un único profesor, sino como una herramienta que puede ser ampliamente utilizada, configurada y aumentada para diseñar cualquier ejercicio o examen Java y para evaluar automáticamente baterías de ejercicios.

El resto del trabajo se estructura como sigue. El trabajo relacionado se presenta en la Sección 2. La Sección 3 muestra el modelo de evaluación que utiliza nuestra herramienta. En la Sección 4 se explica nuestra herramienta y las fases que la componen. Finalmente, las conclusiones se exponen en la Sección 5.

## 2 Trabajo relacionado

Pears y col. 2005 clasifican las herramientas que soportan al proceso de enseñanza en cuatro grupos:

- herramientas de visualización (p.e., Animal [Rosling y Freisleben 2002], Jeliot [Moreno y col. 2004], y Tango [Stasko 1990])
- herramientas de EA (p.e., TRAKLA2 [Korhonen, Malmi y Silvasti 2003], WEB-CAT [Edwards y Pérez-Quiñones 2008], y BOSS [Joy, Griffiths y Boyatt 2005])
- herramientas de soporte a la programación (p.e., BlueJ [Barnes y Kolling 2006])
- microworlds (p.e., Karel [Pattis 1994], y Alice [Cooper, Dann y Pausch 2000])

Centrándonos en las herramientas de EA, el principal área de interés ha sido puntuar automáticamente exámenes tipo test. Esto es de especial interés en asignaturas universitarias con un gran número de alumnos. El problema se ha solucionado y la solución ha sido ampliamente implantada en la mayoría de universidades. Algunos sistemas remarcables de este tipo son [Supic y col. 2014; Hendriks 2012].

Un segundo tipo de herramientas de EA se basa en *ejercicios visuales de simulación de algoritmos* [Laakso y col. 2004]. Estas herramientas utilizan IGU avanzadas que controlan todas las posibles (pero finitas) acciones del estudiante respecto a un problema en particular (p.e., ordenar un array). En los ejercicios visuales de simulación de algoritmos, el estudiante manipula la representación visual de la estructura de datos subyacente en el que se aplica el algoritmo. El estudiante manipula estas estructuras de datos reales mediante operaciones a través de la IGU (p.e., un estudiante puede simular los pasos de Quicksort utilizando la IGU, y el sistema evalúa si cada movimiento sigue el algoritmo real de Quicksort) [Laakso y col. 2004].

En este trabajo nos centramos en el tercer tipo de herramientas de EA, el cuál ha sido menos investigado: EA de código fuente. En esta área, prácticamente todas las aproximaciones son de caja negra, es decir, basados en *output comparison* (véase [Prados y col. 2005; Denny y col. 2011] y los cuatro artículos de revisión [Ala-Mutka 2005; Abd Rahman y Nordin 2007; Liang y col. 2009; Ihantola y col. 2010]). La idea general es compilar el código de los alumnos, ejecutarlo con un conjunto predeterminado de entradas, y comparar su salida con el resultado esperado. Los sistemas más avanzados [Tung, Lin y Lin 2013; Kitaya e Inoue 2014; Beierle, Kulas y Widera 2003] también utilizan generación de casos de test y algún tipo de modelo (normalmente el algoritmo correcto) para comparar los resultados. Algunos de estos sistemas son independientes del lenguaje. Esto es posible debido a que reciben el compilador como parámetro de entrada, y se limitan a comparar los resultados.

Desgraciadamente, existen muy pocas aproximaciones de caja blanca, p.e., que realicen análisis de código para corregir ejercicios. Dos excepciones son [Naudé, Greyling y Vogts 2010] y [Wang y col. 2007]. En ambas aproximaciones la idea consiste en medir la similitud del código del alumno con un repositorio de soluciones conocidas. La similitud se calcula con un grafo que representa el código.

Se puede encontrar en [Ihantola y col. 2010] una comparación de las herramientas de EA. Otras revisiones previas son [Ala-Mutka 2005; Liang y col. 2009; Abd Rahman y Nordin 2007].

Nuestra aproximación también utiliza *output comparison* con generación automática de casos de tests. Pero, al contrario que muchos trabajos relacionados, no utilizamos un repositorio de soluciones conocidas, un modelo, o un grafo. Además, nuestra generación de casos de test no es aleatoria, se basa en un análisis de *path conditions* para maximizar el *branch coverage*.

Otra diferencia importante de nuestra aproximación es el módulo de verificación de propiedades. No somos conscientes de ninguna otra técnica que utilice un DSL para especificar propiedades que posteriormente puedan ser automáticamente validadas, se utilicen para corregir un ejercicio, y asignarles una nota. Finalmente, otra característica que es completamente novedosa es la posibilidad de puntuar un ejercicio que no compila.

## 3 Un modelo de evaluación semi-automático

De la misma forma que en el contexto de la construcción de compiladores se distingue entre tiempo de compilación y de ejecución, nosotros también distinguimos entre dos momentos del proceso de evaluación: *tiempo de construcción del ejercicio* y *tiempo de evaluación*. En la Figura 1 se muestra en qué fase se sitúan los componentes de nuestro sistema.

El sistema está compuesto principalmente de dos componentes independientes:

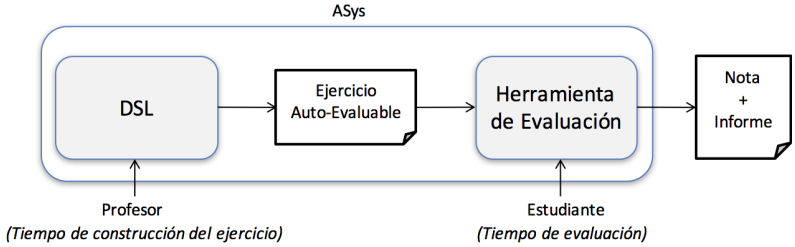


**Fig. 1: Esquema de evaluación semi-automática.**

1. Un DSL para la especificación de ejercicios auto-evaluables. Basado en este lenguaje, hemos desarrollado una IGU que permite a los profesores crear y generar ejercicios auto-evaluables, de tal modo que la herramienta automáticamente genera el código Java que corrige el ejercicio.

2. Una herramienta de evaluación automática que permite la EA de programas escritos en Java. El profesor simplemente carga un ejercicio auto-evaluable, carga la solución proporcionada por el alumno, y el sistema puntúa el ejercicio y genera un informe con todos los errores encontrados. Esta herramienta también puede ser utilizada por los alumnos. Estos pueden automáticamente conocer su nota inmediatamente después de un examen, pero es aún más interesante el hecho de que la herramienta también permite al alumno solucionar ejercicios, convirtiéndose de este modo en un recurso de aprendizaje de gran valor.

Obsérvese que la salida de la primera fase (construcción del ejercicio) es un ejercicio auto-evaluable, el cual es la entrada de la herramienta de evaluación. Un ejercicio auto-evaluable contiene varios componentes:

**Enunciado del ejercicio.** Puede ser proporcionado en cualquier formato (normalmente es un PDF).

**Recursos.** Es una colección (posiblemente vacía) de clases Java con el modelo que el alumno debe completar. Puede contener cualquier recurso necesario para resolver el ejercicio (e.g., ficheros de texto, imágenes, etc.).

**La solución del profesor.** No es más que una colección de clases Java que utilizando los recursos disponibles solucionan el ejercicio.

**Casos de test.** Ya han sido automáticamente generados a partir de la solución del profesor intentando cubrir todas las ramas de decisión. El profesor puede añadir más casos de test si así lo desease (p.e., para testear alguna situación especifica).

**Una plantilla de evaluación.** Es un meta-programa Java capaz de comprobar las propiedades que debe cumplir el código del alumno y asignarles una nota.

A continuación describimos nuestra herramienta de evaluación que recibe como entrada un ejercicio auto-evaluable.

## 4 ASys: Un sistema semi-automático de evaluación

Nuestro sistema de evaluación, junto a manuales, código fuente y ejemplos, está disponible en:

http://www.dsic.upv.es/~jsilva/ASys/

Describimos aquí su funcionalidad y arquitectura, y redirigimos al lector interesado a la página web del proyecto para consultar los detalles de implementación.

Una vez se ha creado un ejercicio auto-evaluable con el DSL, la herramienta de evaluación se utiliza para cargar este ejercicio (véase Figura 1). Hay dos perfiles de usuario disponibles para esta herramienta: profesor y alumno.

**Herramienta de evaluación usada por los profesores** En este caso el objetivo es el de asistir al profesor para puntuar de manera rápida, precisa y homogénea los ejercicios o exámenes. Por lo tanto, el sistema está preparado para evaluar un conjunto de ejercicios. Durante la evaluación, los nombres de los estudiantes se ocultan para asegurar la objetividad. Solo aquellos errores que no pueden ser automáticamente corregidos se muestran al profesor, quien decide qué nota poner. Si no se detecta ningún error, entonces la evaluación es completamente automática.

**Herramienta de evaluación usada por los alumnos** En este caso el objetivo es didáctico. Concretamente, enseñar a los alumnos cómo corregir sus propios errores. Por lo tanto, una vez el sistema muestra al estudiante el error, estos pueden intentar corregirlo, pedir un consejo, o ver la solución del profesor.

La entrada al sistema es (i) un ejercicio auto-evaluable creado con el DSL, y (ii) la ruta del directorio donde se encuentran los ejercicios a ser evaluados. La salida del sistema es, para cada ejercicio, un informe que especifica la nota final junto a una lista de los problemas encontrados. Esta salida es útil para tanto el profesor como el alumno, y por lo tanto, cada informe se duplica y presenta de dos formas diferentes: una para el profesor con información útil para automatizar la puntuación y publicación de notas, y otra para el estudiante con retroalimentación explicando los problemas encontrados. Concretamente, el informe se compone de información acerca de los errores de compilación (compilación), propiedades no satisfechas (análisis), y errores de ejecución (testeo).

Toda esta información proviene de tres fases de evaluación:

1. *Compilación*: Para identificar errores de compilación.
2. *Análisis*: Para comprobar si ciertas propiedades se cumplen.
3. *Testeo*: Para identificar errores de ejecución.

• Fase 1 (compilación). Utiliza el compilador estándar de Java para compilar el código, por lo que informa de los mismos errores que haría el compilador estándar. El

sistema compila el código fuente del alumno normalmente, y muestra al usuario la clase exacta que produce el error, resaltando la linea donde este se encuentra.

• Fase 2 (análisis). Analiza el código fuente para verificar propiedades. Esta es nuestra principal contribución. Para implementar la fase de análisis, anteriormente implementamos un DSL que genera ejercicios auto-evaluables. Debido a que el DSL es compatible con Java, los ejercicios auto-evaluables que se generen se pueden ejecutar directamente por una herramienta de evaluación que automáticamente verifique las propiedades del ejercicio.

• Fase 3 (testeo). En esta fase se testea el código del alumno. El profesor puede proporcionar casos de test hechos manualmente para evaluar el código del alumno. Sin embargo, nuestro sistema no se limita a un conjunto predefinido de casos de test, sino que implementa un módulo de generación de casos de test que automáticamente genera un número arbitrario de casos de test. Para ello, el sistema recibe la solución del ejercicio (en Java), e iterativamente genera los casos de test. Después, estos tests se ejecutan contra la solución del alumno, y, de este modo, la salida que este ofrece se compara con la salida proporcionada por la solución del profesor.

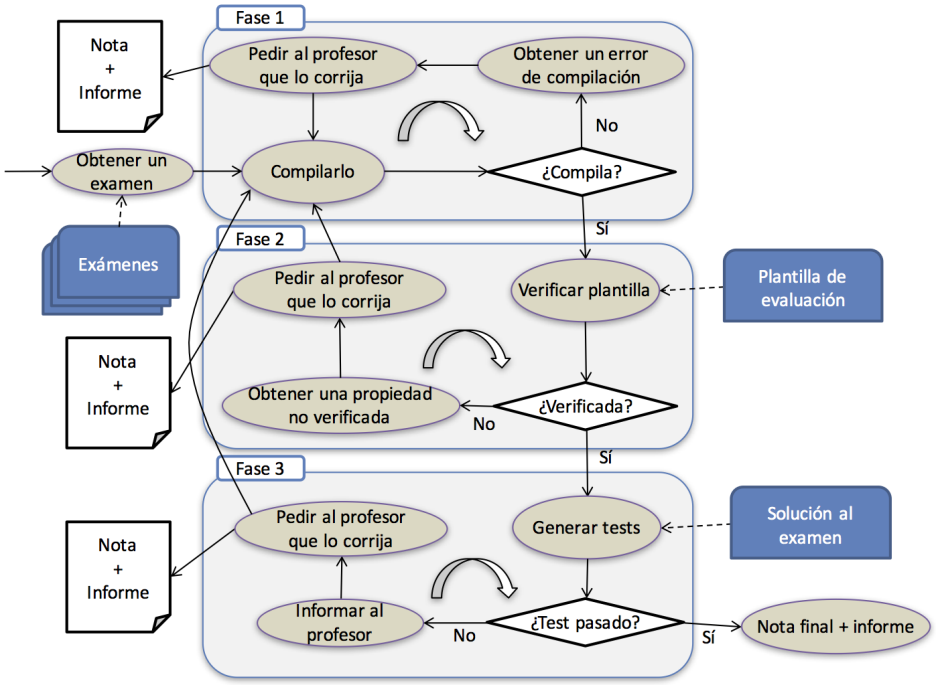


**Fig. 2: Diagrama de flujo de datos de ASys**

Con el fin de entender cómo se construye el triple informe (errores de compilación-análisis-testeo), se puede observar el diagrama de flujo de datos (DFD) del sistema mostrado en la Figura 2. Las cajas oscuras etiquetadas con `Exámenes`, `Solución al examen`, y `Plantilla de evaluación` son las entradas proporcionadas por el usuario. Siguiendo los caminos en el DFD, uno puede fácilmente darse cuenta de las fases que componen la evaluación de los ejercicios `Fase 1`, `Fase 2`, y `Fase 3`.

Uno podría llegar a pensar que estas tres fases se ejecutan secuencialmente. Es decir, el ejercicio se compila una vez, después se analiza el código generado, y, finalmente, se testea para encontrar errores de ejecución. Sin embargo, nuestro sistema trabaja de manera diferente.

Repetimos las tres fases cada vez que el profesor modifica el código fuente; y esto ocurre cada vez que se detecta un error (ya sea un error de compilación, una propiedad no satisfecha, o un caso de test fallido). Por lo tanto, el esquema general del DFD es: `se encuentra un error` → `mostrárselo al profesor y permitirle hacer cambios` → `recompilar el ejercicio` → `comprobar *todas* las propiedades` → `comprobar *todos* los tests`. Esto se repite mientras se sigan detectando errores. Este esquema:

- le permite al profesor corregir (y puntuar) cualquier parte del código donde haya un error. Puede corregir un simple error, más de un error, o incluso el ejercicio completo. Después, ASys comprobará nuevamente que todo funciona correctamente.
- impide al profesor cometer errores. Por ejemplo, una vez que el profesor ha corregido la propiedad $A$, es posible que al corregir la propiedad $B$ se introduzca un error que haga que la propiedad $A$ falle de nuevo, esto produciría que una propiedad dependiente de $A$ que está perfectamente implementada (p.e., $C$) también falle. Pero esto no es ningún problema puesto que ASys recompilará y verificará todas las propiedades de nuevo.

La Figura 3 muestra una impresión de pantalla en la cual se ha ejecutado ASys en el modo semi-automático y este ha informado de un error: *Alumno debe heredar de Persona*. En la ventana izquierda se puede ver (en distintas pestañas) el código sobre el que trabaja el profesor. Este código se corresponde inicialmente con la solución del alumno, y se va corrigiendo hasta que finalmente el ejercicio cumpla con todos los requisitos. En el caso de que los errores no puedan ser corregidos automáticamente y el profesor tenga que intervenir, este dispone del código del alumno y la solución al ejercicio en la ventana de la derecha. Una vez corregido, en la parte inferior, el profesor puede asignar notas y comentarios a los problemas encontrados, los cuales serán incluidos en el informe final que se le enviará al alumno. Con el botón *Añadir*, es posible añadir varias (independientes) notas y comentarios a la misma propiedad (p.e., debido a que se han encontrado varios problemas). Nótese que *Comentario* es un listbox. Este contiene todos los comentarios y notas previas asignadas a esta propiedad (hecho normalmente por el mismo u otro profesor en el examen de otro alumno). Esto es muy útil a la hora de corregir varios ejercicios, puesto que una vez se detecta un error y se le ha asignado una nota y comentario, esta información puede reutilizarse

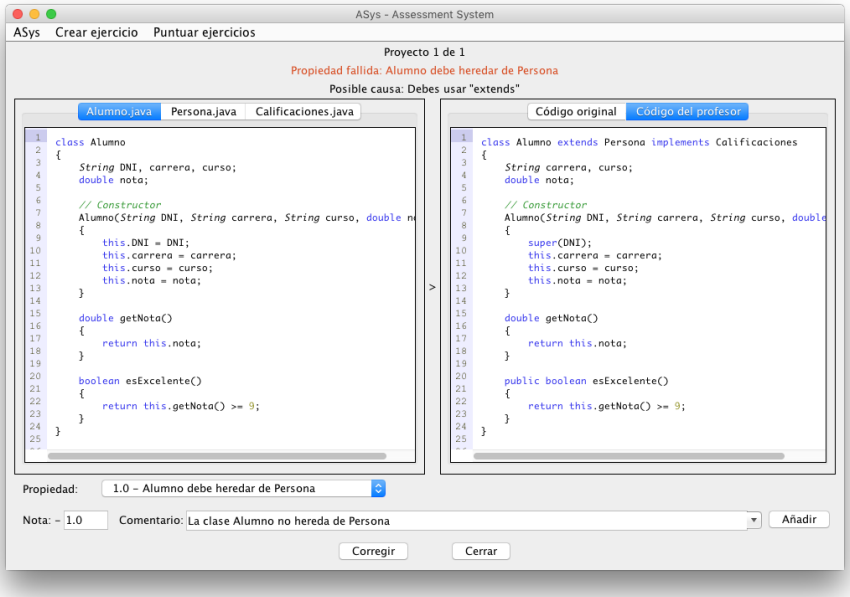


**Fig. 3: Impresión de pantalla de ASys**

por otros profesores para evaluar el resto de ejercicios. Esto acelera significativamente el proceso de evaluación, puesto que tras unos cuantos ejercicios los errores cometidos por los alumnos se repiten una y otra vez; y el profesor solo tiene que hacer clic y seleccionar el comentario y nota correspondiente a ese error. Si, por el contrario, se utiliza el modo automático, entonces ASys solo asigna notas a las propiedades que han fallado pero no las corrige, produciendo un proceso de corrección totalmente automático.

## 5 Conclusiones

Hemos presentado una nueva herramienta para la auto-evaluación de código Java. Esta herramienta introduce una nueva fase de análisis que puede verificar propiedades y asignarles notas. Además, puede ser utilizado tanto en modo didáctico (por los alumnos) como para corregir ejercicios (por los profesores). A su vez, dispone de un modo semi-automático y otro automático que permite al diseñador del ejercicio auto-evaluable especificar si se requiere intervención del profesor en el proceso de evaluación de los ejercicios.

Como ya fue explicado en Ihantola y col. 2010, muy pocos sistemas de EA son código abierto, y menos todavía disponible (gratuitamente). En muchos trabajos se dice que se ha desarrollado un prototipo pero no fuimos capaces de encontrar la herramienta. En otros casos se dice que el sistema era de código abierto pero se necesitaba contactar

con los autores por adelantado. Por todos estos inconvenientes hemos hecho nuestro sistema de código abierto y está públicamente disponible, de tal modo que cualquier otro investigador puede reutilizarlo o unir fuerzas para ayudar a desarrollarlo.

## Referencias

Abd Rahman, Khirulnizam y Md Jan Nordin (2007). "A Review on the Static Analysis Approach in the Automated Programming Assessment Systems". En: *National Conference on Programming 07*.

Ala-Mutka, Kirsti M. (2005). "A survey of automated assessment approaches for programming assignments". En: *Computer Science Education* 15.2, págs. 83-102.

Barnes, David J. y Michael Kolling (2006). *Objects first with Java - A Practical Introduction using BlueJ*.

Beierle, Christoph, Marija Kulas y Manfred Widera (2003). "Automatic Analysis of Programming Assignments". En: *Proceedings of the 1. E-Learning Fachtagung Informatik (DeLFI'03)*. Verlang, págs. 144-153.

Biggs, John y Catherine Tang (2007). *Teaching for Quality Learning at University : What the Student Does (3rd Edition)*. Society for Research Into Higher Education. Open University Press.

Cooper, Stephen, Wanda Dann y Randy Pausch (2000). "Alice: A 3-D Tool for Introductory Programming Concepts". En: *Journal of Computing Sciences in Colleges* 15.5, págs. 107-116. ISSN: 1937-4771.

Denny, Paul y col. (2011). "CodeWrite: Supporting student-driven practice of java". En: *Proceedings of the 42nd ACM technical symposium on Computer science education*. New York, NY, USA: ACM, págs. 471-476. ISBN: 978-1-4503-0500-6.

Edwards, Stephen H. y Manuel Pérez-Quiñones (2008). "Web-CAT: Automatically grading programming assignments". En: *ACM SIGCSE Bulletin* 40.3, págs. 328-328.

Hendriks, Remco (2012). "Automatic Exam Correction". Tesis de mtría. Universiteit Van Amsterdam.

Ihantola, Petri y col. (2010). "Review of recent systems for automatic assessment of programming assignments". En: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. New York, NY, USA: ACM, págs. 86-93. ISBN: 978-1-4503-0520-4.

Joy, Mike, Nathan Griffiths y Russell Boyatt (2005). "The BOSS Online Submission and Assessment System". En: *ACM Journal on Educational Resources in Computing* 5.3. ISSN: 1531-4278.

Kitaya, Hiroki y Ushio Inoue (2014). "An Online Automated Scoring System for Java Programming Assignments". En: *International Journal of Information and Education Technology* 6.4, págs. 275-279.

Korhonen, Ari, Lauri Malmi y Panu Silvasti (2003). "TRAKLA2: A framework for automatically assessed visual algorithm simulation exercises". En: *Proceedings of the Third Annual Baltic Conference on Computer Science Education*, págs. 48-56.

Laakso, Mikko-Jussi y col. (2004). "Automatic assessment of exercises for algorithms and data structures - a case study with TRAKLA2". En: *Proceedings of Kolin Kolistelut/Koli Calling - Fourth Finnish/Baltic Sea Conference on Computer Science Education*, págs. 28-36.

Liang, Yingli y col. (2009). "The recent development of automated programming assessment". En: *International Conference on Computational Intelligence and Software Engineering (CiSE 2009)*. IEEE, págs. 1-5. ISBN: 978-1-4244-4507-3.

Moreno, Andrés y col. (2004). "Visualizing programs with Jeliot 3". En: *Proceedings of the Working Conference on Advanced Visual Interfaces*. AVI '04. New York, NY, USA: ACM, págs. 373-376. ISBN: 1-58113-867-9.

Naudé, Kevin A., Jean H. Greyling y Dieter Vogts (2010). "Marking student programs using graph similarity". En: *Computers & Education* 54.2, págs. 545-561.

Pattis, Richard E. (1994). *Karel the robot: A gentle introduction to the art of programming*. ISBN: 978-0-471-59725-4.

Pears, Arnold y col. (2005). "Constructing a core literature for computing education research". En: *ACM SIGCSE Bulletin* 37.4, págs. 152-161.

Prados, Ferran y col. (2005). "Automatic generation and correction of technical exercises". En: *International Conference on Engineering and Computer Education (ICECE 2005)*.

Rosling, Guido y Bernd Freisleben (2002). "ANIMAL: A system for supporting multiple roles in algorithm animation". En: *Journal of Visual Languages and Computing* 13.2, págs. 341-354.

Stasko, John T. (1990). "TANGO: A framework and system for algorithm animation". En: *IEEE Computer* 23.9, págs. 27-39.

Supic, Marko y col. (2014). "Automatic recognition of handwritten corrections for multiple-choice exam answer sheets". En: *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, págs. 1136-1141. ISBN: 978-953-233-081-6.

Tung, Sho-Huan, Tsung-Te Lin y Yen-Hung Lin (2013). "An Exercise Management System for Teaching Programming". En: *Journal of Software* 8.7, págs. 1718-1725.

Wang, Tiantian y col. (2007). "Semantic similarity-based grading of student programs". En: *Information and Software Technology* 49.2, págs. 99-107.