UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de un sistema de análisis y evaluación de código Java

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Hernández Rocha, Airam

Tutor/a: Silva Galiana, Josep Francesc

CURSO ACADÉMICO: 2021/2022

A mis padres y a mi familia,

que pese a la enorme distancia

que han traído estos años,

nunca han dudado de mí.

Muchísimas gracias.

Resumen

ASys es una plataforma educativa destinada al aprendizaje de programación en Java y Haskell que ha sido desarrollada en la Universitat Politècnica de València. Los objetivos de este sistema son proveer a los alumnos de una plataforma de realización de ejercicios que pueden ser corregidos automáticamente, y que además de rapidez, les proporcione información útil para su desarrollo en la materia. El sistema también permite reducir significativamente la carga de trabajo de corrección de los profesores.

Por tanto, los alumnos podrán mejorar sus habilidades y conocimientos de programación mediante ejercicios que les ofrecen retroalimentación instantánea y automática, con información acerca de sus errores, independencia total de un profesor. Todo ello sin necesidad de tener que enseñar la solución correcta o propiedades genéricas, sino que cada corrección estará hecha sobre el propio código del alumno, lo cual mejorará su comprensión.

Como hemos mencionado, para los profesores también supone una reducción enorme en carga de trabajo, ya que no será necesario que empleen horas corrigiendo este tipo de ejercicios, pudiendo invertir mejor su tiempo en crear otros nuevos, que fomenten de mejor forma la habilidad de resolución de problemas del alumnado y sus conocimientos específicos de Java.

ASys ha sido implementado utilizando Vue.js para el Front-End y Java para el servidor (Spring Boot) y el módulo de corrección automática, en donde se centrará en mayor medida el presente trabajo.

Este proyecto aborda el desarrollo de una funcionalidad de corrección automática de código Java, la cual modificará los fragmentos incorrectos del código del alumno, complementando así a la información sobre los errores que se proporciona actualmente.

En particular, con el desarrollo de este trabajo, ASys será capaz de añadir herencias no presentes o erróneas, sustituir tanto el tipo como el nombre de un atributo o método por el correcto, modificar los parámetros de un método o constructor para que se ajuste a lo especificado por el profesor, añadir *imports* necesarios, transformar una clase a una interfaz o viceversa, entre otras correcciones a la solución enviada. Todo esto se hará de manera automática, sin intervención del profesor ni del alumno.

Palabras clave: ASys, Java, corrección automática, Spring Boot, Vue.js, ejercicios, programación.

Abstract

ASys is an educational platform for learning Java and Haskell programming, developed at the Universitat Politècnica de València. Its main objectives are to provide students with an automatic exercise correction platform that provides fast as well as useful information for their correct development in the field, along with reducing the teacher's correction workload.

Therefore, the students will be able to improve their skills and programming knowledge through exercises that offer instant and automatic feedback, with information regarding their errors and total independence from a teacher. All of this without having to resort to showing the original solution or generic properties. Instead, every correction will be made in the student's code, which will improve their comprehension of what is correct and why.

As we have previously mentioned, it will result in an enormous reduction of workload for the teachers, because it will not be necessary for them to spend hours correcting these types of exercises, leaving time for them to invest in creating new ones that better foster the problem-solving skills of the students, along with their knowledge of Java.

ASys is written in Vue.js for the Front-End and Java for the server (Spring Boot) and correction module, where most of this work will be focusing on.

This project addresses the development of an automatic Java correction functionality, which will complement the current information that is offered to the student when they make a mistake, adding a corrected version of their own code.

Specifically, when this project is completed, ASys will be able to add missing or correct wrong `extends` and `implements`, substitute an incorrect attribute or method type or name, modify a method's or constructor's parameters to adjust it to the one defined by the teacher, add missing imports, transform a class to an interface and vice versa, among other corrections to the student's solution.

Keywords: Vue.js, Spring Boot, Java, automatic correction, ASys, exercises, programming

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Índice de contenidos

Índice de Figuras

Índice de Tablas

Escola Tècnica Superior d'Enginyeria Informàtica

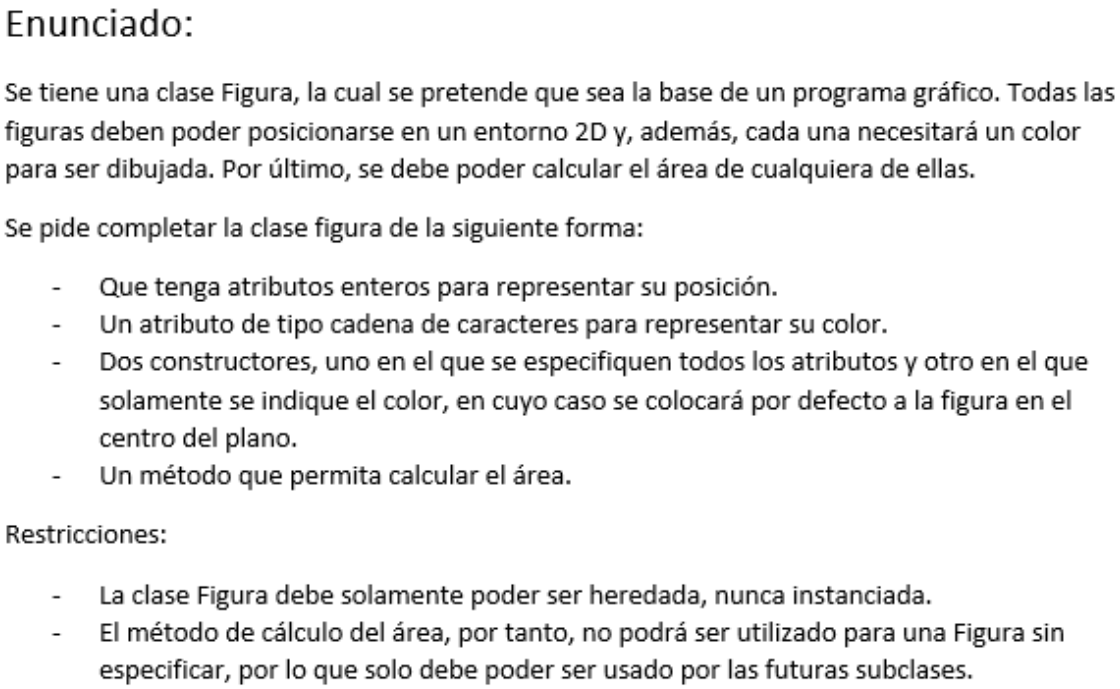
Universitat Politècnica de València

Introducción

1.1 Motivación

Este trabajo aborda un problema ambicioso y enfocado a la meta-programación. Dentro del marco del análisis de software y de desarrollo de herramientas para programadores podemos encontrar la herramienta ASys.

ASys es una plataforma educativa orientada a la enseñanza de programación Java y Haskell, en la que los alumnos resuelven ejercicios que mejorarán su conocimiento y habilidades en este ámbito. En muchas plataformas similares, los ejercicios son corregidos solamente en base al comportamiento y rendimiento del código, pero en el caso de ASys, se pretende proponer ejercicios mucho más complejos, en los que además del funcionamiento también se evalúe la forma en la que se escribe el código. Pongamos como ejemplo el ejercicio con el siguiente enunciado:



Enunciado:

Se tiene una clase Figura, la cual se pretende que sea la base de un programa gráfico. Todas las figuras deben poder posicionarse en un entorno 2D y, además, cada una necesitará un color para ser dibujada. Por último, se debe poder calcular el área de cualquiera de ellas.

Se pide completar la clase figura de la siguiente forma:

- Que tenga atributos enteros para representar su posición.
- Un atributo de tipo cadena de caracteres para representar su color.
- Dos constructores, uno en el que se especifiquen todos los atributos y otro en el que solamente se indique el color, en cuyo caso se colocará por defecto a la figura en el centro del plano.
- Un método que permita calcular el área.

Restricciones:

- La clase Figura debe solamente poder ser heredada, nunca instanciada.
- El método de cálculo del área, por tanto, no podrá ser utilizado para una Figura sin especificar, por lo que solo debe poder ser usado por las futuras subclases.

Figura 1 Enunciado del ejercicio de ejemplo

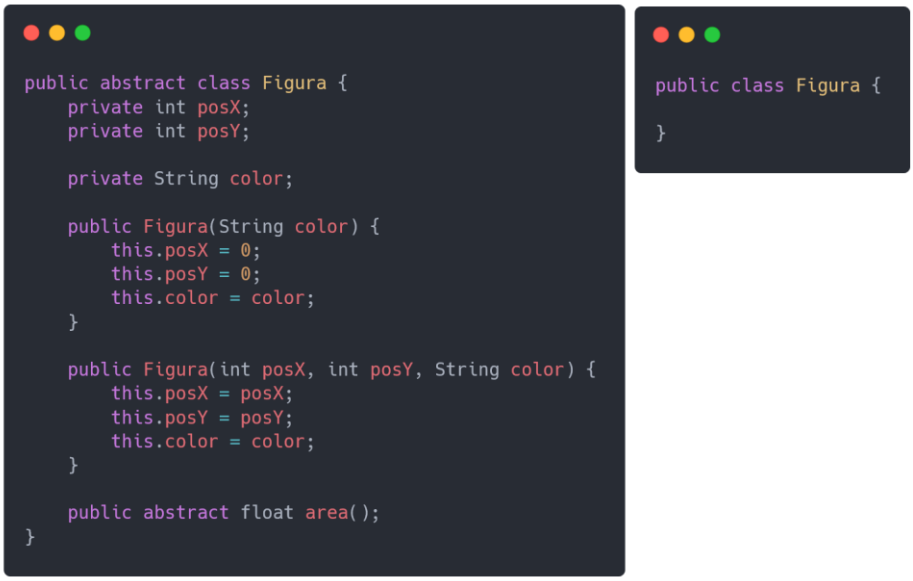
Como podemos observar, se trata de un ejercicio más complejo de corregir de forma automatizada que otro como podría ser calcular el número "x" de Fibonacci o la función factorial de un número de forma recursiva, ya que no nos basta con una batería de pruebas para comprobar la corrección de todos los requisitos especificados. Para conseguir corregir este tipo de ejercicios sin tener que depender de la corrección individual de cada uno por un profesor, se necesita de la corrección automática.

Todos los programas pueden ser estructurados en forma de árbol sintáctico, lo cual es una técnica utilizada en compiladores en la fase de análisis sintáctico (Lee, 2018) y otros programas de análisis de código. La corrección automática sigue un principio parecido: consiste en, partiendo de un código a corregir, el código de la solución y las propiedades que se van a evaluar, descomponer ambos programas en forma de árbol sintáctico para así analizar y modificar su estructura.

En el caso de Java, es posible realizar este proceso descomposición de forma sencilla a través de la librería JavaParser (Danny van Bruggen, 2021). Gracias a esta, es posible aliviar la enorme carga que supondría implementar esta funcionalidad desde cero y, sobre todo, abstraer la modificación de código de forma muy cómoda e intuitiva para el programador.

Para el profesor (o usuario final) solo es necesario añadir dos cosas cuando se crea un ejercicio: el código correcto, sobre el cual especificará en ASys las propiedades que desea que sean corregidas, y un código inicial que se proveerá al alumno para empezar a trabajar. Podemos

ver un ejemplo de cómo se verían estos para el enunciado anterior, a la izquierda el código correcto y a la derecha el punto de partida del alumno:



```java
public abstract class Figura {
    private int posX;
    private int posY;

    private String color;

    public Figura(String color) {
        this.posX = 0;
        this.posY = 0;
        this.color = color;
    }

    public Figura(int posX, int posY, String color) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
    }

    public abstract float area();
}
```

```java
public class Figura {

}
```

Figura 2 Código de ejemplo correcto y recursos del alumno

ASys tiene un módulo de corrección automática de código con una funcionalidad parcial. En la actualidad, dicho módulo realiza el proceso descrito para analizar y detectar errores de programación, pero no permite corregirlos. Es el profesor (o el alumno) quien debe realizar de manera manual la corrección del código erróneo utilizando la retroalimentación de ASys. Un módulo de corrección automática completo debería ser capaz de detectar y arreglar tanto errores estructurales como de comportamiento.

Este proyecto aborda el problema de la corrección automática de código Java, en el apartado de corrección estructural, lo cual supone un enorme reto dado el fuerte componente de meta-programación que ello supone. Se pretende implementar la funcionalidad necesaria en el actual módulo de corrección de código Java del sistema ASys para que este sea capaz de modificar y corregir el código del alumno.

1.2 Objetivos

Este trabajo de fin de grado tiene por objetivo que el sistema ASys sea capaz de transformar el código erróneo proporcionado por el alumno en uno que cumpla la especificación del ejercicio, para que así no solo se le informe de sus errores mediante su descripción, sino que esta se acompañe con un nuevo código corregido, haciendo la experiencia mucho más productiva y similar a lo que sería un ejercicio corregido a mano por un profesor.

Por tanto, el objetivo principal de este TFG es:

(O1) La implementación para el sistema ASys de un módulo de corrección automática de propiedades como herencia, declaraciones de paquetes, atributos, cabeceras y parámetros de métodos y clases, entre otros.

Este objetivo implica la consecución de un subobjetivo que, dada su complejidad, conviene destacar:

(O2) La implementación de técnicas para la resolución de problemas de compilación generados por los cambios del código a raíz de la corrección.

Este segundo objetivo puede ilustrarse fácilmente con un ejemplo:

Supongamos el ejercicio presentado en el apartado anterior, en el que se pretende que el alumno cree una clase abstracta Figura con atributos que representen la posición y color de esta. En el primer apartado se especifica que los atributos de posición deben de ser de tipo entero, pero imaginemos que el alumno, fruto de la prisa por terminar o de un simple despiste, ignora esa especificación y termina utilizando como tipo un valor real, enviando como solución el siguiente código:

```java
public abstract class Figura {
    private double posX;
    private double posY;

    private String color;

    public Figura(String color) {
        this.posX = 0.0;
        this.posY = 0.0;
        this.color = color;
    }

    public Figura(double posX, double posY, String color) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
    }

    public abstract float area();
}
```

Figura 3 Código erróneo enviado por el alumno

En este caso el único error presente sería ese tipo equivocado. Sin embargo, debemos fijarnos que, debido al tercer requisito del enunciado, el cual pide que se escriba un constructor en el que los atributos de posición se inicialicen con valor cero, a ambos se les asigna el valor cero con punto decimal. Esto provocará que en el momento que ASys sustituya el tipo `double` por un `int` presente en la solución, el código sea incorrecto y no compile.

1.3 Impacto esperado

Tras el desarrollo de este trabajo, se espera que ASys de un salto cualitativo como plataforma de enseñanza de programación tanto de cara a los alumnos que la utilicen como de cara a los profesores, que percibirán una reducción significativa del trabajo de corrección.

Se espera que, tras la implementación de esta nueva funcionalidad, el rendimiento y entendimiento de los alumnos sobre los problemas planteados a través del sistema, junto con sus posibles soluciones, se vea incrementado significativamente; además de que se vea reducida la frustración de estos cuando encuentren que han competido errores en los ejercicios.

Por parte de los profesores, gracias a este trabajo se cree que aumentará su confianza a la hora de utilizar la plataforma para la evaluación de sus alumnos, pudiendo dedicar gran parte del tiempo que se dedicaba a la corrección manual, al desarrollo de ejercicios y recursos educativos de mayor calidad.

1.4 Metodología

La metodología aplicada a este trabajo ha sido la metodología de desarrollo en espiral. Esta metodología consiste en desarrollar el producto mediante iteraciones de todo el proceso, desde la planificación hasta la evaluación, pasando por el desarrollo. En cada una de las iteraciones o bien se planifican y añaden nuevas funcionalidades o se replantean las existentes y se corrigen los problemas del producto, todo ello en función de los descubrimientos de la etapa anterior.

Antes de iniciar la espiral se ha realizado una fase de formación, con el objetivo de adquirir los conocimientos necesarios acerca de Spring (VMware, 2022) y Vue.js (Vue.js Team, 2022) para poder entender la estructura y lógica del proyecto.

La espiral se ha organizado con una primera fase de preparación y planificación. Para comprender mejor el funcionamiento de los procesos de creación y corrección de ejercicios, se realizó una exploración del código actual, su versión anterior de escritorio y de la documentación. Tras esto, establecimos una planificación con los máximos de fechas en las que se deberían haber implementado y documentado las diferentes funcionalidades. Esta planificación puede resumirse en el siguiente diagrama de Gantt:

Figura 4 Diagrama de Gantt de la planificación

Tras realizar un extenso análisis del problema y un diseño de la solución propuesta, se abordó la fase de implementación, con entregas y correcciones cada aproximadamente dos semanas. En paralelo a la misma, se fue documentando todo el proceso y se fueron redactando cada uno de los apartados de la memoria. La implementación ha seguido un desarrollo incremental, en el que hemos ido introduciendo de una en una las correcciones de diferentes propiedades del código (herencia, cabeceras de clase, atributos, métodos…).

En la fase de validación y testeo tuvieron lugar las pruebas, a partir de un conjunto extenso de baterías de casos de prueba para probar de manera sistemática el software desarrollado. Finalmente, se realizaron varias iteraciones de revisiones y retoques finales en la memoria.

En las primeras iteraciones, como por el momento se habían introducido pocos tipos de correcciones, la implementación se limitó a completar el método corrector presente en ASys. No obstante, esto no resultaría óptimo a largo plazo, por lo que a lo largo de las siguientes fases se terminó creando un paquete completo con toda la lógica de corrección.

Tras esto, hubo una iteración centrada exclusivamente en el manejo de los errores de compilación y, por último, otra centrada en mejorar la precisión del módulo de detección de errores y la integración con la interfaz de usuario.

1.5 Estructura

La memoria se ha estructurado siguiendo los apartados que se describen a continuación.

Tras la introducción, se encuentra la especificación de requisitos; en la que se detallará el funcionamiento del sistema de corrección, junto con una estimación del coste de trabajo que se deberá llevar a cabo.

Seguidamente, se detallará el problema y las librerías que serán necesarias para la implementación de la solución. A continuación, en la fase de diseño, se presentará dicha solución, su estructura y cómo esta contribuye a las decisiones tomadas para afrontar las diferentes situaciones que se generen a causa de la corrección.

Posteriormente, se pasará a explicar la implementación realizada, centrándonos en sus principales características y las dificultades que han surgido a lo largo del proceso de desarrollo, junto con ejemplos del funcionamiento final.

En el próximo apartado se expone cómo se puede desplegar el sistema ASys, enseñando los pasos a seguir y el software necesario para su funcionamiento. A continuación, se mostrarán todas las pruebas que se han realizado para verificar que cumple con los requisitos previamente especificados.

Por último, concluiremos con un análisis sobre la precisión de las estimaciones de esfuerzo que se plantearon al inicio del proyecto y acerca de si se han cumplido los objetivos establecidos. Para terminar, se propondrán ideas para el futuro del sistema ASys y mejoras que podrían llevarse a cabo en el apartado de corrección.

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

[redacted] Especificación de Requisitos Software

2.1 Introducción

2.1.1 Propósito

El propósito de esta ERS es detallar todos los requisitos funcionales y no funcionales que deberán implementarse en el sistema ASys para conseguir la corrección automática del código del alumno. Va dirigida a alumnos, profesores y desarrolladores de ASys.

2.1.2 Ámbito del módulo de corrección de ASys

El módulo de corrección de Asys es el encargado de corregir el código del alumno. Puede verse como una función que recibe un código fuente y devuelve otro código fuente. Por tanto, implementa una transformación de programas. Este módulo se encargará de que, cada vez que un alumno envíe el resultado de un ejercicio Java a través de ASys, si se le encuentran errores intentará corregirlos en la medida de lo posible. Se podrán corregir cabeceras de clases, métodos, `imports`, atributos, declaraciones de paquetes y demás propiedades que puedan ser verificadas mediante el código de la solución ofrecido por el profesor y la especificación que haya hecho al crear el ejercicio.

No se corregirán, sin embargo, los casos en los que el código sea erróneo por el comportamiento de este, es decir, en los que los cuerpos de funciones y métodos provoquen que los casos de prueba den resultados diferentes a los de la solución del profesor.

Gracias a este sistema, los alumnos tendrán una mayor cantidad de información disponible y ejemplos personalizados en base a sus errores en tiempo real, además del alivio de carga y enormes posibilidades que esto ofrece al profesorado.

Los objetivos del sistema son poder llegar a corregir la mayor parte de los errores que sea posible corregir, dentro de lo mencionado anteriormente, tratando de modificar el código todo lo necesario si surgieran errores de compilación para devolver una respuesta satisfactoria en el mayor número de casos.

2.1.3 Definiciones, acrónimos y abreviaturas

Dentro del ámbito de esta especificación de requisitos, se van a utilizar los siguientes términos:

- **Meta programación:** Programación en la que un programa es capaz de modificarse a sí mismo o a otros programas, es decir, código capaz de escribir o modificar código.
- **ERS**: Especificación de Requisitos Software.
- **Front-End**: Es la parte de un sistema con la que interactúa un usuario. Contiene la interfaz gráfica del sistema.
- **Back-End:** Parte lógica del sistema en la que se procesan las peticiones y se provee de la información necesaria al Front-End.
- **Spring Boot:** Framework para desarrollar APIs web en Java.

2.1.4 Referencias

[1] "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, pp. 1–40, Oct. 1998, doi: 10.1109/IEEESTD.1998.88286.

[2] D.Insa, S.Pérez, J.Silva, S.Tamarit, "Semiautomatic generation and assessment of Java exercises in engineering education," Oct. 3, 2020. https://onlinelibrary.wiley.com/doi/10.1002/cae.22356

[3] M. Gregorio, "Gamificación de un Sistema de Ejercicios Auto-Corregibles de Programación en Java," Universitat Politècnica de València. Valencia, 2021

[4] A. Maya Gomis, "Diseño y Desarrollo del sistema ASys con Spring y Vue.js," Valencia, 2019.

[5] J. Ramallón Martínez, "Desarrollo de un IDE para el Desarrollo y la Corrección Automatizada de Ejercicios de Programación," Valencia, 2020.

2.1.5 Visión general del documento

En esta especificación de requisitos se describirá el funcionamiento del sistema de corrección automática de código a implementar en ASys. Para empezar, se describirá el lugar que ocupará

este módulo en el sistema general y cómo interactuará con el mismo. Después se detallarán las funciones que debe implementar, divididas en características y requisitos.

Seguidamente, se tratarán las características de los usuarios, describiendo los perfiles de aquellos que se verán afectados por la implementación de esta funcionalidad, las restricciones que se deben cumplir durante el desarrollo, los requisitos que se podrían implementar en una futura versión y, para terminar, los atributos de calidad del sistema que deben mantenerse en ASys una vez finalizado el trabajo actual.

2.2 Descripción general

En esta sección vamos a explorar la arquitectura y funcionamiento del sistema ASys una vez el módulo de corrección se encuentre integrado en este.

2.2.1 Perspectiva del producto

El producto que se desarrollará forma parte del subsistema de corrección de ejercicios de ASys. Cuando un alumno envíe una solución a un ejercicio a través del Front-End web, si el módulo actual detecta que existen errores, se encargará de utilizar la solución proporcionada por el profesor para modificar el código del alumno y generar una versión correcta del mismo utilizando meta programación. Una vez hecho esto, se le enviará al usuario.

Para proporcionar la funcionalidad descrita, el módulo de corrección será llamado a través de la plantilla generada por el ejercicio. Cuando el detector de errores encuentre un fallo en una propiedad, se llamará al corrector de esta. Se utilizarán tanto los árboles de nodos de la solución como del ejercicio enviados. Para ilustrar la interacción de los módulos del sistema en el proceso de corrección, consultar el siguiente diagrama:

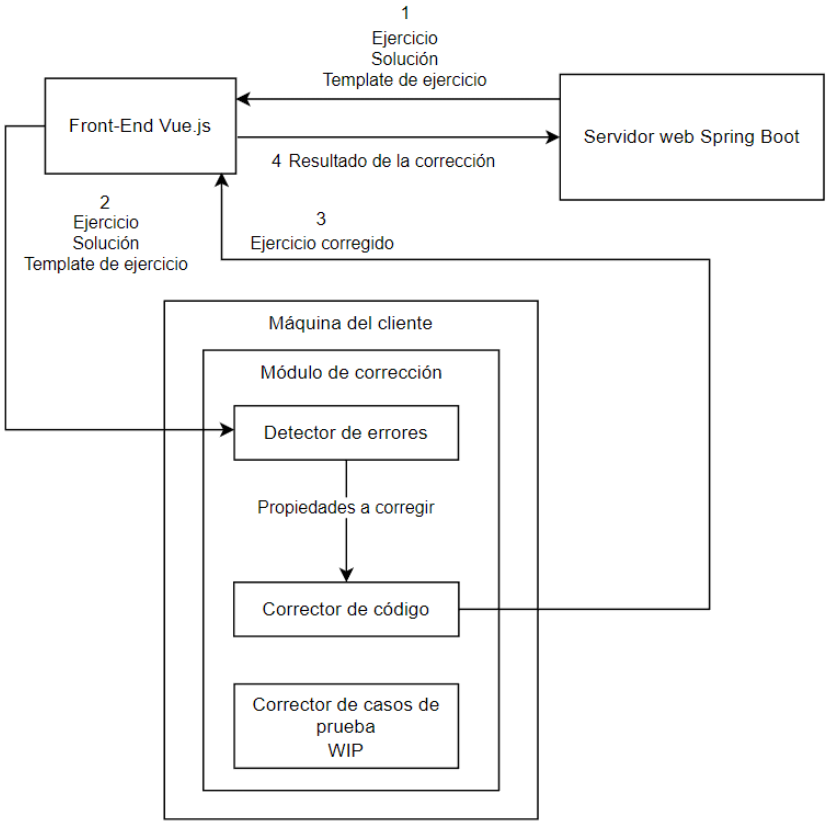


Figura 5. Diagrama de la interacción del proceso de corrección de ejercicios una vez completado el sistema especificado.

En el diagrama se muestra, además, que el módulo de corrección no se ejecuta en la máquina servidora, sino que deberá ejecutarse en la del alumno. Gracias a esto, es posible aliviar enormemente la carga de corrección del sistema, haciendo que si, por ejemplo, hay 60 alumnos intentando corregir sus ejercicios, el rendimiento de ASys no se vea afectado.

Como podemos comprobar, el módulo corrector solamente interactúa con el servidor web, sirviendo para aliviar la carga de este cuando tiene que manejar peticiones del resto de funciones del sistema y al mismo tiempo corregir ejercicios. Para mostrar mejor los límites del módulo de corrección se adjunta también un diagrama de contexto de este:

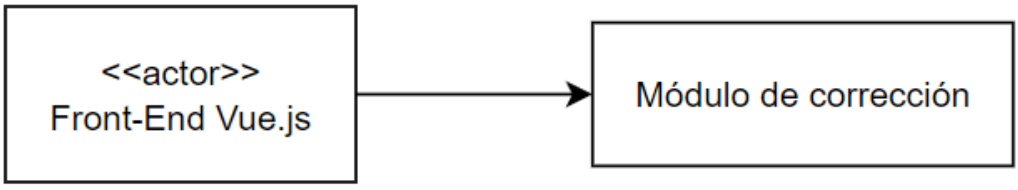


Figura 6 Diagrama de contexto del módulo corrector

2.2.2 Funciones del producto

Cuando se haya completado el desarrollo, se deben haber completado las siguientes funcionalidades:

- Gestión de errores de herencia.
    - **Añadido de herencia necesaria.** Dada una clase en la solución del alumno, se deben incluir las clases heredadas que no estén presentes en la misma. Se incluyen tanto herencias de clase `extends`, como implementaciones de interfaces con `implements`.
    - **Sustitución de herencias erróneas.** Si existe herencia, pero no es la correcta, se debe sustituir por la adecuada.
    - **Eliminado de atributos incorrectos por la herencia.** Si un atributo existe en la subclase del alumno, no existe en la subclase solución y sí en la superclase solución, se eliminará.
    - **Gestión de polimorfismo paramétrico.** Al añadir un parámetro mediante herencia, existirá la opción de que ASys infiera el tipo incorrecto y lo sustituya por el parámetro añadido.
    - **Transformación clase-interfaz e interfaz-clase.** Si una clase debía ser una interfaz, se eliminarán sus atributos y el cuerpo de los métodos. En el caso de transformación interfaz-clase, se sustituirá todo el cuerpo (atributos y métodos) por el de la clase solución.
- Corrección de atributos
    - Añadido de atributos no presentes.
    - **Corrección de atributos con propiedades erróneas.** Si el atributo no tiene el tipo o visibilidad correctos, se debe corregir.
    - **Corrección de nombre de atributos.** Si un atributo con un nombre no presente en la solución cumple todas las propiedades del correcto, y su nombre es una propiedad evaluable, este se sustituirá en todo el archivo.
- Corrección de `imports`
    - Añadido de `imports` necesarios que no existan.
- **Corrección de propiedades erróneas de clases.** Se deberán poder corregir en caso de que se requiera, tanto el nombre como la visibilidad de las clases.
- Corrección de métodos y constructores

    - **Corrección de propiedades erróneas.** Se deben poder corregir tanto el nombre como la visibilidad o si son estáticos o no, si se requiere en la solución. El nombre deberá sustituirse en todas las llamadas a lo largo del archivo.
    - **Corrección de parámetros de entrada.** Se debe poder corregir el número, tipo, nombre y orden de los parámetros de la función, según se especifique.
    - **Corrección de cantidad.** Si se requiere por el profesor, se deberá ajustar el número de métodos a la cantidad estipulada, usando como referencia para eliminar o añadir, la clase solución.
- Prevención de errores de compilación tras la corrección
    - **Gestión de errores de compilación por herencia.** Los errores causados por esta corrección deben ser tenidos en cuenta y arreglados en los siguientes casos:
        - Métodos que deben ser definidos en la subclase.
        - Atributos necesarios en el constructor.
        - Llamada necesaria a "`super()`" en el cuerpo del constructor.
- Minimización de generación de errores de compilación
    - **Ordenación de propiedades según capacidad de generación de errores.** Se deben ordenar las propiedades antes de solucionarlos para minimizar, o como mínimo posponer, la generación de errores de compilación.
- Gestión de errores de compilación
    - **Reversión de propiedades causantes de errores.** El mayor número de correcciones de propiedades posible deben de ser reversibles.

2.2.3 Características de los usuarios

Como usuarios del sistema a desarrollar podemos identificar varios tipos:

- **Alumno.** El alumno es el usuario que enviará sus respuestas a los ejercicios y quien, por tanto, recibirá el resultado de las correcciones. Su nivel de experiencia en la programación variará desde prácticamente nula hasta ser expertos.
- **Profesor.** Se encarga de revisar aquellas correcciones que se hayan marcado como necesarias. Sobra mencionar que su nivel educacional y técnico será muy alto.
- **Creador de ejercicios.** Los ejercicios pueden ser creados por cualquier usuario de ASys, tanto profesor como alumno. Se encarga de proporcionar los recursos y de definir qué

propiedades deben ser comprobadas, puntuadas y corregidas. Este tipo de usuario debería tener un alto nivel técnico y experiencia en la programación.

2.2.4 Restricciones

Las restricciones impuestas para el desarrollo del sistema son:

- La corrección debe estar integrada en el módulo actual de corrección.
- Debe estar escrito en Java.
- Se deben usar las librerías de reflexión presentes en el módulo de corrección.
- Las llamadas a los métodos de corrección deben realizarse mediante la plantilla generada cuando se crea el ejercicio.
- La respuesta al Front-End se debe mantener con el servidor web Spring como intermediario.
- El sistema debe poder seguirse ejecutando en cualquier sistema operativo.
- El código Java del alumno y del profesor deben estar escritos en una versión de Java que pueda ser soportada por el módulo de JavaParser que haya integrado en el sistema.

2.2.5 Requisitos futuros

Tras haber completado este trabajo, en el futuro, el producto podría evolucionar satisfactoriamente mediante la implementación de los siguientes requisitos:

- **Configuración de seguridad en las correcciones.** Se debe añadir una opción que solamente permitiera realizar correcciones que en ningún caso vayan a causar errores de compilación.
- **Configuración de correcciones de métodos.** Se debe poder elegir qué aproximación se debe tomar en caso de que un método del profesor no se encuentre en el código del alumno, teniendo por opciones: sustituir un método percibido como incorrecto (aproximación actual) y añadirlo como método extra.

2.2.6 Atributos del Sistema

Los requisitos de calidad que se deben cumplir en el sistema tras la implementación de este módulo deben ser:

- **Fiabilidad:** Se debe informar y tratar los fallos adecuadamente. En cuanto al sistema de corrección en específico, no se deben devolver correcciones erróneas. Es decir, el sistema de corrección debe ser *correcto* por definición, pero no necesariamente *completo*. Por tanto, siempre que haya un error y lo corrija, el código corregido estará bien (propiedad

de corrección). Pero es posible que haya errores que no corrija (propiedad de incompletitud).

- **Mantenibilidad**: El código debe mantenerse con la menor complejidad y mayor legibilidad posible; además de que debe haber una gran calidad en la documentación para mantener el sistema mantenible.
- **Eficiencia**: Las peticiones deben tener un tiempo medio de respuesta de aproximadamente 1 segundo, y en ningún caso exceder la cifra de 5 segundos.
- **Disponibilidad**: El sistema debe mantenerse disponible.
- **Seguridad**: Las peticiones deben estar autentificadas y realizarse mediante el protocolo HTTPS (Google, 2022).

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Análisis del Problema

La arquitectura de ASys está compuesta por cuatro servidores con diferentes funciones y el cliente web. En primer lugar, tenemos al servidor de base de datos para la persistencia, en segundo lugar, el servidor de archivos para guardar los ejercicios, fotos de perfil, ficheros de premios y todo aquello que no es posible o eficiente almacenar en la base de datos; el propio servidor web con la lógica de negocio, el módulo de corrección (en el cual se centra este trabajo) y, por último, la interfaz web.

En la siguiente figura, podemos observar la arquitectura de forma gráfica y cómo se relacionan los componentes entre ellos, además de mención de algunas de las tecnologías utilizadas en cada uno.

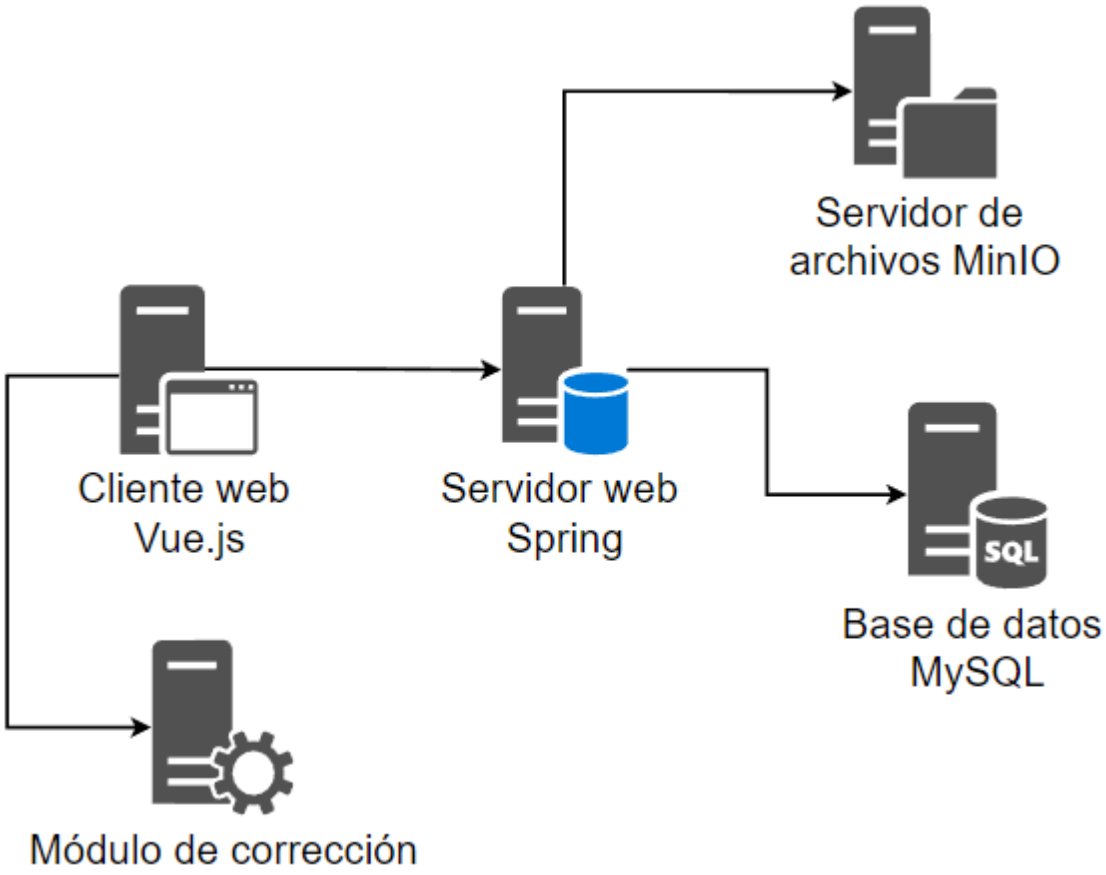


Figura 7 Representación de la arquitectura de ASys

A lo largo de la presente sección trataremos con detalle las tecnologías y *frameworks* utilizados en cada uno de los componentes, además de las librerías que juegan una parte fundamental en el sistema.

3.1 Tecnologías en el cliente

El cliente está desarrollado con el *framework* de código abierto Vue.js, por lo que utiliza tanto HTML (Mozilla Foundation, 2021), CSS (Mozilla Foundation, 2021) y JavaScript (Mozilla Foundation, 2022).

3.1.1 Vue.js

Vue.js es un *framework* que, al contrario de otras propuestas como *React* (Facebook, 2021) con su JSX (Facebook, 2022) o Angular (Google, 2022) con TypeScript (Microsoft, 2022), se basa en HTML, CSS y JavaScript estándar para construir interfaces web basadas en eventos y escritas de forma declarativa.

Lo que sí tiene en común con los *frameworks* previamente mencionados es su propuesta de desarrollo de las interfaces mediante componentes reutilizables. En Vue, un componente solamente necesita de un archivo que se divide en tres secciones delimitadas por etiquetas HTML:

- **Script**: Dentro de esta etiqueta colocaremos toda la lógica del componente en JavaScript.
- **Style**: Esta sección contendrá la hoja de estilo CSS para el estilo del componente.
- **Template**: Aquí colocaremos todo el código HTML que definirá la vista del componente.

ASys, sin embargo, no utiliza componentes creados desde cero; sino que aprovecha una librería de componentes llamada Vue Material (Moura, 2020), que provee de elementos básicos estilizados siguiendo las directrices de Material Design (Simoes, 2021) de Google para acelerar el desarrollo y conseguir mejores resultados.

3.1.2 HTML y CSS

HTML (*HyperText Markup Language*) es un lenguaje de maquetación basado en etiquetas predefinidas con la forma "`<etiqueta></etiqueta>`" o "`<etiqueta/>`", que define la estructura y elementos de una página web, siendo el elemento básico de construcción de interfaces de usuario en este entorno.

No obstante, para conseguir una experiencia de usuario óptima y satisfactoria, no basta con HTML, sino que se necesita algo más para mejorar visualmente las aplicaciones y páginas. Para ello existe CSS (*Cascading Style Sheets*). Se utiliza para definir hojas de estilo que modificarán la presentación de documentos definidos en un lenguaje de maquetación, en este caso HTML.

En ASys, como ya hemos mencionado, se utilizan como parte de Vue.js.

3.2 Tecnologías del servidor

El servidor de ASys está implementado en Java (Oracle, 2021), acompañado de Spring, un *framework* diseñado para la creación de aplicaciones web o microservicios fácilmente desplegables utilizando contenedores Docker (Docker, Inc., 2022). Para la capa de persistencia, se utiliza una base de datos relacional (Google, 2022) MySQL (Oracle, 2022), que interactúa con la capa de servicios a través de la *Java Persistence API* (JPA) (Oracle, 2013) y cuyos datos se definen mediante la herramienta Hibernate (Hibernate, 2022). Para el servidor de almacenamiento de ficheros se ha optado por MinIO (MinIO, Inc, 2022).

3.2.1 Spring

Spring es un *framework* de código abierto fundamental en el desarrollo Java en la actualidad. La gran cantidad de funcionalidad que provee hace que el desarrollo se acelere exponencialmente, ahorrando a los programadores escribir código repetitivo que no influye en su lógica de negocio.

Utiliza un contenedor de inversión de control (IoC) para instanciar y conectar entre sí los objetos en la aplicación, los cuales se obtienen mediante el uso de inyección de dependencias (Crusoveanu, 2022).

Normalmente, una aplicación Spring tendrá para cada entidad del sistema una clase "controlador", que será la puerta de entrada a través de HTTP. Podemos indicar cuál queremos que sea esta clase mediante la anotación "`@RestController`" o "`@Controller`", que suele ir acompañada de "`@RequestMapping('ruta/de/entrada')`", que a su vez define la URL necesaria en la petición para acceder a los métodos del controlador. Dentro de estas funciones, podemos definir cuál atenderá a qué método HTTP utilizado con anotaciones como "`@GetMapping`" o "`@PostMapping`", o también, tener URLs personalizadas para métodos específicos.

Cada controlador tendrá como atributo a un servicio que proveerá la lógica de negocio y que será instanciado mediante inyección de dependencias, ya sea con la anotación propietaria de Spring "`@Autowired`" o con la estándar desde Java EE 6 "`@Inject`".

La clase servicio se definirá marcándola con "`@Service`". En la mayoría de los casos obtendrá los datos desde la capa de persistencia utilizando el mismo mecanismo de inyección de dependencias, pero esta vez con un repositorio o DAO, que se definirá con "`@Repository`". Spring provee de las operaciones CRUD por defecto para cualquier entidad, pero podremos extender la clase "`Repository<Entity, ID>`" (donde "`Entity`" es la clase del objeto almacenado e "`ID`" es el tipo de su clave primaria) para definir repositorios con operaciones más específicas.

3.2.2 JPA + Hibernate + MySQL

JPA (Java Persistence API) es un *framework* de la plataforma Java EE que utiliza el lenguaje JPQL (Java Persistence Query Language)[27] para abstraer las comunicaciones con una base de datos relacional.

Por su parte, Hibernate es una herramienta de mapeo objeto relacional, la cual es capaz de definir tablas SQL mediante *mappings* definidos con un fichero XML, o más recientemente, anotaciones JPA como "`@Entity`", "`@Table(name = "TABLE_NAME")`" o "`@Id`".

El sistema de gestión de bases de datos relacional que se ha elegido ha sido MySQL, desarrollado por Oracle Corporation.

3.2.3 MinIO

MinIO es un servidor de almacenamiento de archivos de código abierto diseñado para ser desplegado en servicios *cloud* mediante Docker y Kubernetes (Google, 2022). En ASys se utiliza para guardar los archivos de los ejercicios, premios y avatares.

3.3 Tecnologías del módulo de corrección

El módulo de corrección está también desarrollado en Java, pero no utiliza las mismas tecnologías que podemos ver en el servidor. La comunicación con el cliente se implementa con el *framework* Restlet (Talend, 2021), mientras que para la corrección se utiliza la librería JavaParser.

3.3.1 Restlet framework

Restlet es un *framework* de código abierto para el desarrollo de servicios REST bastante ligero, pero que provee extensiones para cubrir cualquier funcionalidad deseada. Con solamente el *core* del *framework* tendremos soporte para funcionalidad tanto de servidor como de cliente de cualquier API Web. Busca ser compatible con cualquier capa de presentación o persistencia.

Restlet funciona de la siguiente forma:

- Creamos un objeto de la clase `Component`.
- Al objeto `Component` le añadimos un objeto Server", que contendrá el protocolo a utilizar y puerto, tras esto, llamamos al método "`start()`".
- Añadimos al servidor por defecto del `Component` (que será el que acabamos de añadir) una clase que herede de la clase "`restlet.Application`".
    - La clase en su interior tendrá un método "`createInboundRoot()`" que definirá las direcciones en las que atenderá peticiones y las clases encargadas de cada una.
- Las clases encargadas de atender peticiones heredarán de la clase `ServerResource` y cada método irá acompañado de la anotación del método HTTP que le corresponde, como por ejemplo "`@Put`" o "`@Get`". Estos devolverán un objeto `Representation` que será la respuesta que se enviará al cliente.

3.3.2 JavaParser

Como ya hemos mencionado, JavaParser es una librería que es capaz de generar un árbol de sintaxis abstracta (AST) a partir de código Java de versión 12 o inferior. En ASys, se utiliza para generar los árboles tanto de la solución del profesor como del ejercicio del alumno y, mediante este, hacer las comprobaciones y correcciones necesarias.

Para comprender mejor el funcionamiento de la librería, la representación en árboles sintácticos y el proceso de corrección, vamos a utilizar un ejemplo. Utilizaremos una versión reducida del código del ejercicio de la clase Figura que ya hemos visto con anterioridad, pongamos que se trata de un código parcial que envía el alumno.

```java
public abstract class Figura {
    private String color;

    public abstract float area();
}
```

Figura 8 Fragmento del código del ejercicio de Figura

Cuando JavaParser procesa este programa, lo que genera es el ya mencionado AST que representa su estructura, que podemos representar gráficamente como lo siguiente:

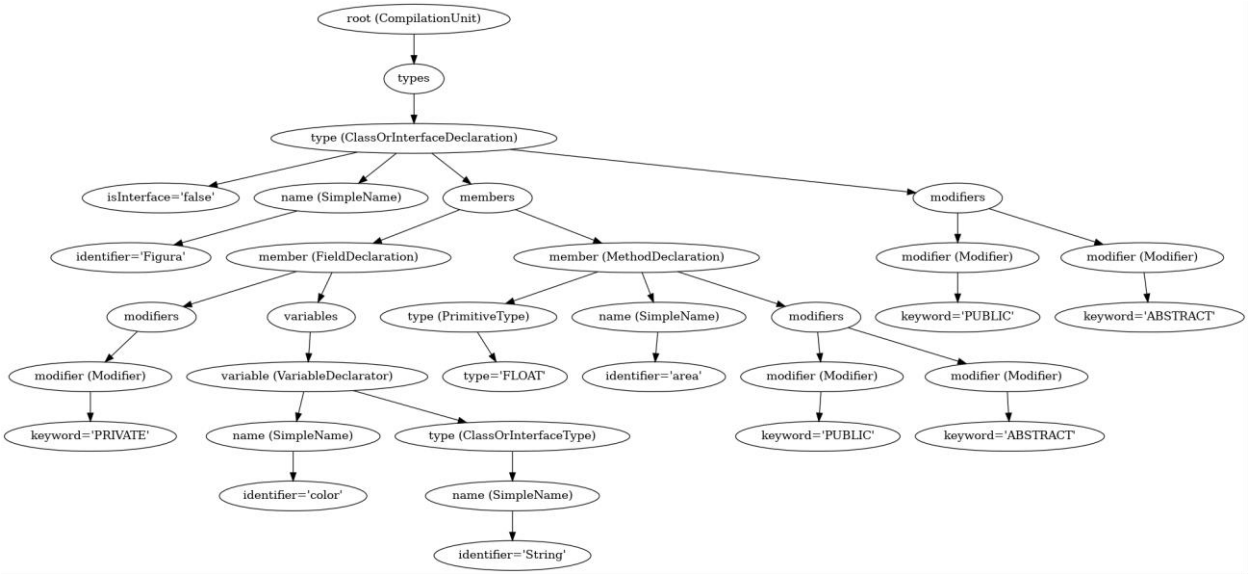


Figura 9 AST del fragmento de código del ejercicio Figura

Como podemos observar, solamente con un código tan reducido ya obtenemos un árbol de anchura y profundidad considerables. Dentro encontramos a la clase principal bajo el nodo *types*, con sus características representadas por sus hijos, entre las que se encuentran el nodo que nos indica si es una interfaz, sus modificadores (`abstract` y `public`) y un nodo padre que contiene las representaciones de sus miembros: un atributo y una declaración de método abstracto.

El módulo de detección de errores explora tanto el árbol generado por el código del alumno como el del código del profesor, intentando encontrar los nodos faltantes, sobrantes o que deben ser sustituidos. El trabajo del módulo de corrección que se desarrolla en este trabajo es, por tanto, realizar esas tareas de eliminación, sustitución y añadido en el árbol del alumno, acompañadas por supuesto de otras acciones necesarias específicas de cada corrección, de la indicación del error al alumno y del manejo de errores surgidos.

Por tanto, se trata de la librería más importante de cara al proyecto actual, dado que para corregir y modificar el código no se modifica este en su forma textual, sino que como se ha explicado, se sustituyen, eliminan o añaden los nodos necesarios en el código del alumno.

3.4 Aspectos clave del problema

Para conseguir un sistema de corrección de código que funcione correctamente, no es solo necesario modificar el código, sino que hay que tener en cuenta las múltiples consecuencias y problemas que pueden surgir de dicha modificación.

En primer lugar, en algunos casos se pueden causar errores de compilación tras corregir, ya sea porque se ha eliminado un método que era llamado en el cuerpo de otro o incluso desde un archivo diferente.

Volvamos de nuevo al ejemplo del ejercicio Figura, pero esta vez extenderemos el enunciado de la siguiente forma:

> Una vez se ha completado la clase Figura, se debe crear una clase FiguresGroup, que representa un grupo de figuras dibujadas en el plano. Además de contener todas las figuras necesarias, se debe poder calcular el área que estas ocupan en total.
>
> Se pide completar la clase FiguresGroup de la siguiente forma:
>
> - Un atributo colección de tipo ArrayList para contener a las figuras.
> - Un constructor por defecto que inicie la colección de figuras a una lista vacía.
> - Un método que calcule el área ocupada por las figuras.

Figura 10 Continuación del enunciado del ejercicio Figura, añadiendo FiguresGroup

El profesor, por ende, debe escribir esa clase para que sirva de solución al ejercicio, por lo que envía este código:

```java
import java.util.ArrayList;

public class FiguresGroup {
    public ArrayList<Figura> figuras;

    public FiguresGroup() {
        this.figuras = new ArrayList<>();
    }

    public float totalArea() {
        return (float) figuras.stream()
                .mapToDouble(Figura::area)
                .sum();
    }
}
```

Figura 11 Código correcto FiguresGroup

Aquí podemos observar que se necesita hacer llamadas al método "`area()`" de la clase Figura dentro de la función "`totalArea()`", por lo que será sensible al nombre de este método. Ahora pongamos que el alumno ha enviado como solución a las clases Figura y FiguresGroup los siguientes fragmentos:

```java
public abstract class Figura {
    private int posX;
    private int posY;

    private String color;

    /*
     * CONSTRUCTORES
     */

    public abstract float calculaArea();
}
```

Figura 12 Clase figura con el nombre del método "area" incorrecto

```java
import java.util.ArrayList;

public class FiguresGroup {
    public ArrayList<Figura> figuras;

    public FiguresGroup() {
        this.figuras = new ArrayList<>();
    }

    public float totalArea() {
        float area = 0.0f;
        for (int i = 0; i < figuras.size(); i++) {
            area += figuras.get(i).calculaArea();
        }
        return area;
    }
}
```

Figura 13 Clase FiguresGroup con método "area" con nombre incorrecto

En la clase Figura del alumno, el nombre del método para calcular el área es diferente al especificado por el profesor, por lo que en el caso de que el nombre de la función resulte evaluable, este será sustituido en la corrección por "`area()`", generando el siguiente código:

```java
public abstract class Figura {
    private double posX;
    private double posY;

    private String color;

    /*
        CONSTRUCTORES
    */

    //ASys: Se ha sustituido calculaArea por area
    public abstract float area();
}
```

Figura 14 Código de la clase Figura con el nombre del método "area" corregido

El problema viene en el momento en el que intentamos volver a compilar. Es en este punto en el que nos damos cuenta de que por culpa de que en el código de la función "`totalArea()`" del alumno se llama a "`calculaArea`" en lugar de "`area`", tenemos un código erróneo pese a que la transformación del código es correcta.

Con este tipo de corrección, nuestra única opción es deshacerla para no enviar un código erróneo, puesto que explorar todos los cuerpos de todos los archivos en busca de aquellos lugares en donde se llame un método en específico para sustituirlo resulta, como mínimo, muy poco eficiente y escalable.

Otro tipo de error que también puede ocurrir se da en los casos en los que se ha cambiado el tipo de una variable que a su vez era inicializada con un tipo incompatible en alguna parte del código (una variable siendo inicializada con una cadena de caracteres y corrigiendo su tipo a entero), como pudimos ver en un ejemplo anterior con el cambio entre un número real a un entero o más claramente en la siguiente figura, la corrección de un atributo de cadena de caracteres a un número entero:

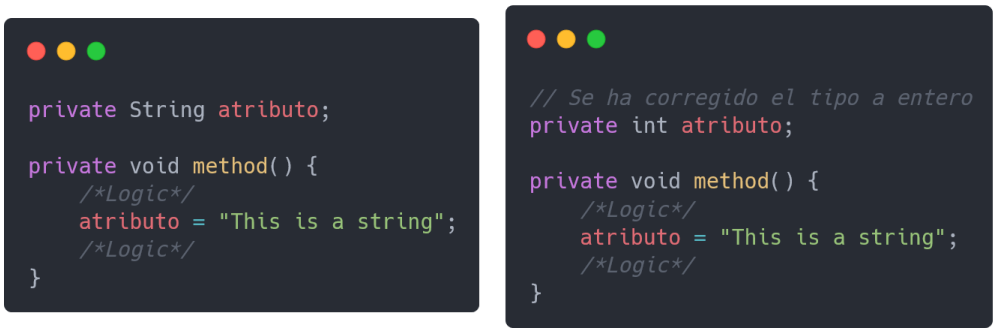


Figura 15 Inicialización que impide la corrección del tipo

Sin embargo, no todos los tipos de propiedades generan inherentemente el mismo número de errores de compilación al modificarse. No resulta igual de peligroso cambiar el nombre o tipo de una variable como lo puede ser añadir un nuevo método o modificar la visibilidad de un atributo. También puede haber propiedades que no compilen justo después de ser corregidas, pero que, gracias a cambios generados por otras propiedades posteriores, el código vuelva a compilar.

Podemos ilustrar esta situación utilizando de nuevo el ejercicio de la clase Figura. Imaginemos que el alumno, de nuevo, se ha equivocado con los tipos de los atributos posicionales, escribiéndolos como números reales en lugar de enteros y, por tanto, también lo ha hecho con los parámetros del constructor, enviando este código:

```java
public abstract class Figura {
    private float posX;
    private float posY;

    private String color;

    public Figura(String color) {
        this.posX = 0.0f;
        this.posY = 0.0f;
        this.color = color;
    }

    public Figura(float posX, float posY, String color) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
    }

    public abstract float area();
}
```

Figura 16 Código erróneo del alumno con números de punto flotante en lugar de enteros

Si resulta que se corrige antes una de las propiedades que representan el tipo de cualquiera de los atributos “posX” o “posY”, esto provocará que, como el atributo del constructor sigue siendo un número de punto flotante, pero ahora un atributo será entero, el código sea incorrecto y no compile, como se muestra en la siguiente figura:

```java
public abstract class Figura {
    //ASys: Se ha sustituido float por int
    private int posX;
    private float posY;

    private String color;

    public Figura(String color) {
        this.posX = 0.0f;
        this.posY = 0.0f;
        this.color = color;
    }

    public Figura(float posX, float posY, String color) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
    }

    public abstract float area();
}
```

Figura 17 Resultado parcial de la corrección que ha provocado que el código no compile

En los siguientes pasos de la corrección, se irán sustituyendo los tipos de los parámetros y del atributo restante, por lo que el resultado final será un código correcto sin problemas de compilación, pero no podemos ignorar el hecho de que en los pasos intermedios ha habido correcciones que, sin ser incorrectas, han provocado que el programa deje de compilar, para consecuentemente volver solucionarse en los siguientes pasos.

Teniendo en cuenta estas cuestiones, el sistema en ningún caso debe generar una corrección con problemas de compilación como resultado final, pero tampoco resulta lógico no devolver ninguna solución si es posible alcanzar un resultado intermedio corrigiendo el mayor número de propiedades posible. Por consiguiente, la solución será deshacer aquellas correcciones que provoquen errores de compilación.

3.5 Estado previo del sistema

Antes de comenzar el desarrollo, se encontraban desarrollados el módulo de detección de errores y una versión muy básica de un módulo de corrección.

3.5.1 Funcionamiento del módulo de detección de errores

El módulo de detección de errores es el encargado de, utilizando las propiedades a corregir definidas por el profesor, proveer al módulo de corrección de la información necesaria para poder realizar la corrección.

Esto se hace mediante una estructura de datos `Map`, que almacena como clave el identificador de cada propiedad y como valor el resultado de la identificación de errores en forma de un objeto llamado `PropertyTree`. Esta clase contiene mucha información, pero de cara a la corrección, resultan relevantes los siguientes campos:

- Los códigos en forma de árbol del alumno y el profesor.
- Dos atributos de tipo `Object`, llamados `project` y `solution`, que representan el nodo o nodos incorrectos del alumno y su contraparte del profesor, respectivamente.
- El tipo de propiedad a tratar, como por ejemplo herencia, tipos o nombre.

A partir de este punto nos referiremos a `project` y `solution` con sus traducciones al castellano para agilizar la lectura de la explicación.

Los atributos proyecto y solución, al tratarse de objetos del tipo `Object`, pueden contener valores de diferentes tipos dependiendo de la propiedad a corregir o de la información disponible. En general, para todas las correcciones, se puede dar uno de los siguientes casos:

- Ambos son objetos cuya clase hereda de la clase `Node`, lo que significa que hay algo incorrecto en el código del alumno que debe ser sustituido.

- O bien proyecto es nulo y solución hereda de la clase `Node` o viceversa. Esto ocurre cuando hay algo en el código del profesor que no está presente en la del alumno, o bien algo en la clase del alumno que debe ser eliminado.
- Ambos son objetos son instancias de la clase `NodeList`. Esto representa aquellos casos en los que hay una serie de nodos que el profesor ha especificado que deben estar juntos en la solución, como una lista de `imports`, una serie de modificadores en específico, etc.
- Valores booleanos en ambos. Esto quiere decir que hay una propiedad que no coincide entre los códigos del profesor y del alumno, ya sea que un método que debería ser de tipo `void` no lo es, o que una clase debería ser en su lugar, una interfaz.

En resumen, el trabajo del módulo de detección de errores es, además de evaluar el código del alumno, obtener para cada propiedad unos valores en sus atributos de proyecto y solución que tengan sentido y sean útiles para que el módulo de corrección pueda arreglar el programa.

Durante el desarrollo de este proyecto también se han desarrollado paralelamente mejoras en esta parte del sistema, por lo que tanto su precisión como la información que proveía han ido mejorando.

Antes de este trabajo, este no enviaba valores nulos cuando un elemento faltaba, sino que siempre intentaba sustituir un nodo del alumno por otro del profesor en caso de error, entre otros problemas que se han ido solucionando.

3.5.2 Funcionamiento de la versión previa del módulo de corrección

Toda la funcionalidad de corrección se encontraba en un solo método que devolvía un valor booleano indicando si se había podido corregir la propiedad o no. No se tenía en cuenta el tipo de la propiedad a corregir (si era un modificador, herencia, tipo o nombre), sino que solamente actuaba en función del objeto que se utilizaba para representar a las soluciones del alumno y profesor (un nodo simple o una lista de nodos).

Esta aproximación si bien era capaz de corregir algunas situaciones, no era apta para la inmensa mayoría de casos posibles ni tampoco se planteaba una solución a las consecuencias de modificar el código incorrectamente.

Podemos exponer muchos casos de correcciones que no se podían llevar a cabo con el estado previo del sistema, así que volvamos al ejercicio de la Figura para mostrar uno simple y otro con un nivel de complejidad mayor.

Para el primer caso, actualicemos el enunciado del ejercicio con una cláusula más, que pedirá al alumno añadir métodos extra en la clase `FiguresGroup` para tener implementar la

funcionalidad de añadir y eliminar cualquier figura del grupo, quedando de esta forma el enunciado:

> Una vez se ha completado la clase Figura, se debe crear una clase FiguresGroup, que representa un grupo de figuras dibujadas en el plano. Además de contener todas las figuras necesarias, se debe poder calcular el área que estas ocupan en total. Se debe poder añadir y eliminar cualquier figura.
>
> Se pide completar la clase FiguresGroup de la siguiente forma:
>
> - Un atributo colección de tipo ArrayList para contener a las figuras.
> - Un constructor por defecto que inicie la colección de figuras a una lista vacía.
> - Un método que calcule el área ocupada por las figuras.
> - Un método que elimine una figura de la lista y no devuelva nada.
> - Un método que añada una figura a la lista y no devuelva nada.

Figura 18 Nuevo enunciado de FiguresGroup con los métodos añadir y eliminar

El código del profesor también tendrá que ser actualizado con estos nuevos métodos, quedando de la siguiente forma:

```java
import java.util.ArrayList;

public class FiguresGroup {
    public ArrayList<Figura> figuras;

    public FiguresGroup() {
        this.figuras = new ArrayList<>();
    }

    public float totalArea() {
        return (float) figuras.stream()
                .mapToDouble(Figura::area)
                .sum();
    }

    public void addFigura(Figura figura) {
        figuras.add(figura);
    }

    public void removeFigura(Figura figura) {
        figuras.remove(figura);
    }
}
```

Figura 19 Código de FiguresGroup del profesor con los métodos "addFigura" y "removeFigura"

Como se indica en el enunciado, ambos métodos deben ser `void`, es decir, no deben devolver nada. Sin embargo, existe la posibilidad de que, dado que ambos métodos "`add()`" y

"`remove()`" devuelven un valor booleano, un alumno no lea bien el enunciado y decida definir estos métodos de esta forma:

```java
public boolean addFigura(Figura figura) {
    boolean added = figuras.add(figura);
    return added;
}

public boolean removeFigura(Figura figura) {
    boolean removed = figuras.remove(figura);
    return removed;
}
```

Figura 20 Métodos "addFigura" y "removeFigura" con tipo erróneo

En este caso, el detector de errores informaría de que hay dos propiedades de tipo *VoidType* con valor de proyecto a `false` y solución a `true`. Sin embargo, en el estado previo del sistema no había ninguna lógica implementada para manejar y corregir este tipo de situaciones, al igual que otras del mismo tipo como que una clase sea erróneamente una interfaz o viceversa.

Para otro tipo de corrección que no era posible llevar a cabo en la versión anterior, debemos de expandir el enunciado del ejercicio de figuras; esta vez de la siguiente forma:

> En siguiente lugar, se pide implementar una clase rectángulo, para así poder dibujar esta figura en el plano. Como no basta con su posición y color para ser dibujado, se debe añadir la información necesaria sobre sus lados y, además, es necesario poder calcular su área. Debe ser posible construir un rectángulo utilizando solamente su color y lados o especificar también su posición.
>
> Se pide por tanto completar la clase Rectángulo de la siguiente forma:
>
> - Debe heredar de la clase Figura.
> - Debe implementar el método área.
> - Debe tener dos atributos de tipo punto flotante para representar ambos lados.
> - Dos constructores, uno en el que solo se especifiquen los lados y el color, y otro en el que se introduzcan todos sus datos.

Figura 21 Enunciado del ejercicio de Figuras incluyendo la nueva clase Rectángulo

Y como no puede ser de otro modo, la solución del profesor se actualizará con la versión correcta de la nueva clase Rectángulo, que será así:

```java
public class Rectangulo extends Figura {
    private float base;
    private float altura;

    public Rectangulo(String color, float base, float altura) {
        super(color);
        this.base = base;
        this.altura = altura;
    }

    public Rectangulo(int posX, int posY, String color, float base, float altura) {
        super(posX, posY, color);
        this.base = base;
        this.altura = altura;
    }

    @Override
    public float area() {
        return base * altura;
    }
}
```

Figura 22 Versión del profesor de la clase Rectángulo

En esta ocasión, el alumno va a escribir una versión correcta de la clase Rectángulo, excepto por el pequeño detalle de que no va a heredar de la clase Figura. En su lugar, se va a implementar toda la información dentro de la propia clase Rectángulo, quedando una respuesta de la siguiente forma:

```java
public class Rectangulo {
    private float base;
    private float altura;
    private int posX;
    private int posY;
    private String color;

    public Rectangulo(String color, float base, float altura) {
        this.posX = 0;
        this.posY = 0;
        this.color = color;
        this.base = base;
        this.altura = altura;
    }

    public Rectangulo(int posX, int posY, String color, float base, float altura) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
        this.base = base;
        this.altura = altura;
    }

    public float area() {
        return base * altura;
    }
}
```

Figura 23 Clase Rectángulo sin herencia

En el estado previo del sistema, es posible solucionar la falta de herencia de la clase, ya que su declaración se representa con una lista de nodos, para la que sí que estaba implementado su sustitución al completo.

No obstante, no solo quedaría una clase incorrecta, con atributos que provocan ocultación sin que esta se pida en el enunciado, sino que como Figura no dispone de un constructor por defecto sin parámetros y en esta versión del corrector no se añade una llamada a "`super`" en el cuerpo de los constructores, el ejercicio ni siquiera compila. Cabe destacar que este error de compilación sí resulta relevante y no puede ser solucionado con ninguna corrección posterior, puesto que los cuerpos de los métodos y constructores no se analizan en el ámbito de la corrección automática estructural.

Como conclusión, se tenía un método demasiado genérico, capaz de realizar cambios muy simples en el código pero que no tenía en cuenta ni las particularidades de cada corrección, ni tampoco las posibles consecuencias de estas. La solución a este problema será explicada en el siguiente capitulo.

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Diseño de la solución

En este apartado vamos a explorar el diseño de la solución a desarrollar para el problema descrito, su arquitectura y funcionamiento. Para poder atajar los problemas presentados en el apartado anterior que no eran tenidos en cuenta por la versión previa del sistema, se ha diseñado el siguiente proceso de corrección:

4.1 Antes de la corrección

Antes de comenzar a corregir *per se*, todas las propiedades son ordenadas en función de su tipo, es decir, "Nombre", "Tipo", "Modificador", "Herencia", etc. El principio seguido en la ordenación se basa en colocar las propiedades más propensas a causar errores de compilación en último lugar. El orden de corrección finalmente ha sido:

Orden	Propiedad
1	Imports
2	Implementación de interfaces
3	Métodos que deberían ser *void* y no lo son
4	Herencia
5	Cambios de clase a interfaz y viceversa
6	Declaraciones de paquete
7	Número de constructores
8	Declaración al completo de constructores
9	Parámetros de tipo (Polimorfismo paramétrico)
10	Parámetros de métodos o constructores
11	Declaración de la clase al completo
12	Modificadores de atributos, métodos o clase (visibilidad o *abstract*, *final*, etc.)
13	Declaración al completo de métodos
14	Tipos
15	Nombres
16	Número de métodos

Tabla 1 Orden de corrección de propiedades

La razón principal para corregir en último lugar las propiedades más “peligrosas”, es para que en los casos en los que se termine la corrección y el resultado no compile, tener muchas más probabilidades de que la propiedad que ha provocado este error es de las últimas realizadas y de esta forma perder el mínimo trabajo posible, realizando el menor número de compilaciones.

Pongamos como ejemplo la corrección del siguiente código del ejercicio de las figuras, en la que el alumno envía esta solución:

```java
public class Figura {
    public float posX;
    public float posY;

    protected String color;

    public Figura(String color) {
        this.posX = 0.0f;
        this.posY = 0.0f;
        this.color = color;
    }

    public float area() {
        return posX * posY;
    }
}
```

Figura 24 Código incorrecto de la clase Figura para ilustrar la utilidad del orden de corrección

Si no tuviéramos en cuenta ningún orden a la hora de corregir, podría darse el caso de que al final de la ejecución se diera un resultado similar al siguiente:

```java
//7: Cambiada clase a abstracta
public abstract class Figura {
    //1: Cambiado de public a private
    //3: Cambiado de float a int
    private int posX;
    //2: Cambiado de public a private
    //6: Cambiado de float a int
    private int posY;

    protected String color;

    public Figura(String color) {
        this.posX = 0.0f;
        this.posY = 0.0f;
        this.color = color;
    }

    //5: Añadido constructor faltante
    public Figura(int posX, int posY, String color) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
    }

    //4: Cambiado método a abstracto
    public abstract float area();
}
```

Figura 25 Posible resultado de la corrección sin tener en cuenta el orden

Los números presentes en los comentarios del código de la figura anterior representan el orden en el que se ha corregido dicha propiedad.

Si tenemos en cuenta que no se han ordenado ni establecido ningún tipo de restricciones sobre cuándo se realiza cada tipo de corrección, resulta muy probable que lleguemos a conseguir frecuentemente casos con estas características, en el cual si nos fijamos podemos notar que tanto la propiedad número tres como la seis son causantes de errores de compilación.

Esto significa, que a partir de la propiedad número tres, todo el resto se registrarán como correcciones que no han compilado, por lo que ocurrirá lo siguiente:

1. Se encontrará que la propiedad tres es la raíz de los errores de compilación y se deshará.
2. Aún con esta corrección deshecha, el código sigue sin compilar, por lo que se continúa.
3. Se deshace la propiedad cuatro y no cambia nada.
4. Se deshace la propiedad cinco y no cambia nada.
5. Se deshace la propiedad seis y por fin se consigue un código que compile.

Por lo tanto, no haber establecido un orden ha provocado que, de siete propiedades a corregir, se mantengan en el resultado tres. Por otro lado, si hubiéramos aplicado el orden propuesto, el resultado de la corrección se vería de una forma similar a esta, con posibles variaciones entre las propiedades que comparten tipo:

```java
//2: Cambiada clase a abstracta
public abstract class Figura {
    //3: Cambiado de public a private
    //7: Cambiado de float a int
    private int posX;
    //4: Cambiado de public a private
    //6: Cambiado de float a int
    private int posY;

    protected String color;

    public Figura(String color) {
        this.posX = 0.0f;
        this.posY = 0.0f;
        this.color = color;
    }

    //1: Añadido constructor faltante
    public Figura(int posX, int posY, String color) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
    }

    //5: Cambiado método a abstracto
    public abstract float area();
}
```

Figura 26 Resultado de la corrección aplicando orden de propiedades

Como se puede ver, se ha corregido en primer lugar el constructor faltante. En segundo lugar, los modificadores no presentes o incorrectos y, por último, los cambios de tipo que generan errores. Gracias a haber ordenado con anterioridad, de las siete propiedades en total, solamente se han tenido que deshacer dos, y en ambas resulta sencillo indicar el error del alumno mediante la adición de un comentario en esa línea.

4.2 Durante la corrección

Después de corregir cada propiedad, el código es compilado y se almacena el resultado generado, tanto si compila de forma correcta como si no. En cualquiera de los casos, la corrección continúa hasta que se han procesado todas las propiedades.

4.3 Después de la corrección

Una vez todas las propiedades se han corregido y los resultados de las compilaciones han sido almacenadas, pueden darse dos casos:

1. El código compila perfectamente
2. Existen errores de compilación

Como se ha expuesto en el ejemplo de la fase de ordenación, cuando se ha finalizado el proceso de corrección y el código compila satisfactoriamente, se da por finalizado y se envía el resultado al alumno.

En el caso contrario, debemos atajar los errores que hayan surgido. Empezamos por buscar cuál es la propiedad raíz de los errores o, en otras palabras, aquella a partir de la cual el código no ha compilado. No sirve buscar la primera que dio un error de compilación, ya que otra corrección sucesiva puede haberlo arreglado. La búsqueda debe por tanto empezar por el final de la lista de resultados. Estos casos se pueden representar de forma visual con este diagrama:

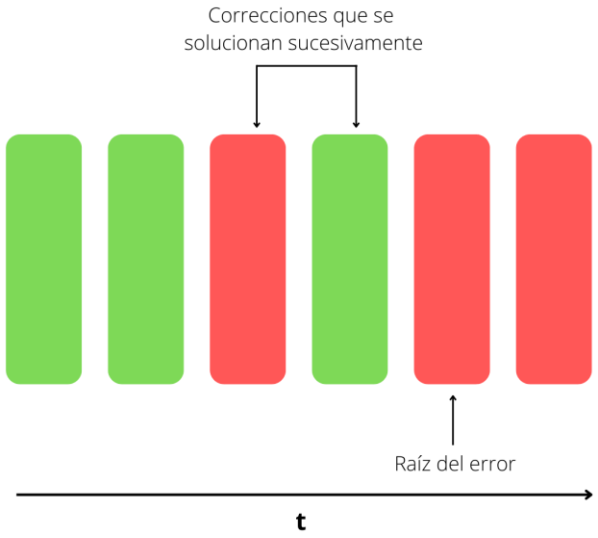


Figura 27 Representación de la búsqueda de la raíz del error

Cada rectángulo representa una propiedad corregida, en color verde aquellas en las que el proyecto ha seguido compilando tras su corrección, y en rojo, al contrario. Se encuentran ordenadas de forma cronológica de izquierda a derecha.

En él podemos ver que, pese a que la tercera propiedad ha provocado un error de compilación, no se trata de la raíz del error, puesto que una corrección sucesiva lo ha arreglado. Por tanto, para encontrar la verdadera raíz, deberemos buscar desde el final de la lista hasta encontrar aquella a partir de la cual el código ha dejado de compilar, en este caso, la quinta.

Una vez encontrada la raíz del error, los atributos “solución” y “proyecto” se invierten y se corrige el código de nuevo. Por ejemplo, si el error se generaba en una propiedad que tenía como solución `int` y como proyecto `float` (por lo que convertía un elemento de tipo `float` en uno

de tipo `int`), para deshacer la corrección se invierten los valores y se realiza el mismo proceso de corrección.

Utilicemos el ejemplo de corrección errónea presente en el apartado de Análisis del Problema. En este caso, se ha realizado una corrección que provoca un error de compilación (transformar de `float` a `int`) por culpa del contenido del constructor. Por consiguiente, el proceso de corrección se podría representar de la siguiente manera:

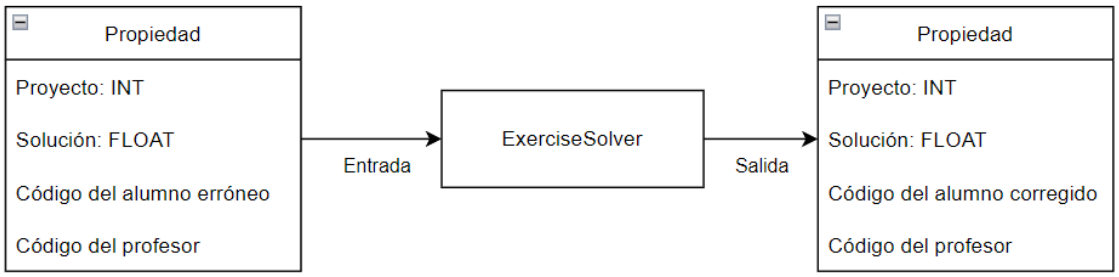


Figura 28 Diagrama del proceso de corrección

Para deshacer la corrección, el manejador de errores realizará el siguiente proceso:

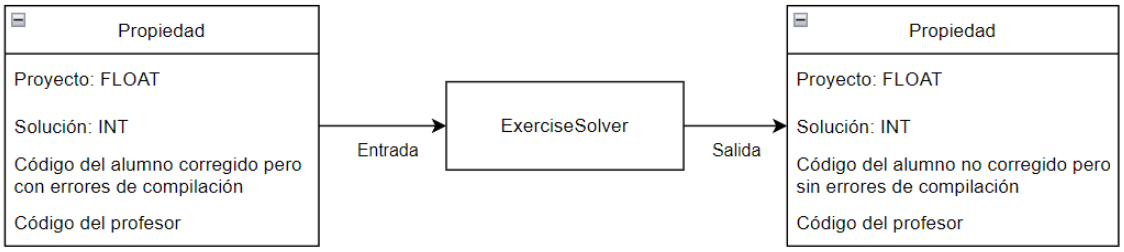


Figura 29 Diagrama del proceso de corrección inverso (deshacer correcciones)

Podemos observar que el proceso para deshacer errores es, esencialmente, el mismo que para corregir, con la diferencia de que el manejador de errores se habrá encargado de invertir las propiedades “proyecto” y “solución”, consiguiendo el efecto contrario.

Seguidamente, volvemos a compilar y repetimos el proceso anterior hasta que, o bien tengamos un código que compile a modo de solución intermedia, o sea imposible revertir la corrección, en cuyo caso devolveremos que no se ha podido corregir el ejercicio.

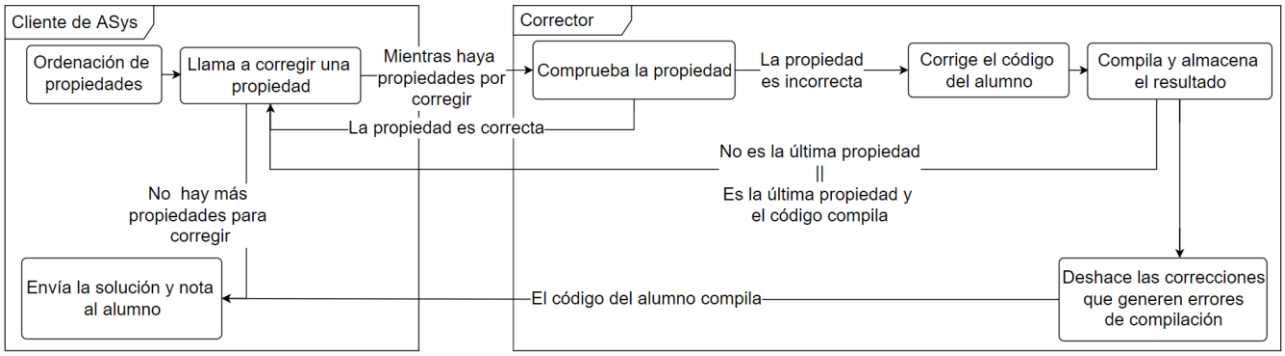


Figura 30 Diagrama del nuevo proceso de corrección

Podemos imaginar un caso en el que no se pueda corregir el ejercicio utilizando el ejemplo previo de esta sección. Si no existiera el proceso de ordenación y alguna de las propiedades que debían sustituir el tipo de los atributos se hubiera corregido en último lugar, terminaríamos devolviendo que el ejercicio no se ha podido corregir puesto que hemos deshecho todas las propiedades.

En cualquier caso, la fase de ordenación existe para minimizar lo máximo posible el número de propiedades deshechas, las compilaciones necesarias y, por supuesto, el número de casos en los que un ejercicio resulta incorregible.

Desarrollo de la solución

En esta sección se va a profundizar en el proceso de implementación y desarrollo de este proyecto, tanto de la corrección en sí misma, como de otros cambios que se han llevado a cabo en el sistema en general.

5.1 Desarrollo del módulo de corrección

La corrección de código, como ya hemos visto, se divide en tres fases: la ordenación de propiedades, la corrección propiamente dicha y el manejo de los errores surgidos. En este apartado vamos a profundizar en la implementación realizada para llevar a cabo cada una de estas fases.

5.1.1 Fase de ordenación previa

En lo que respecta al desarrollo de la funcionalidad de corrección de código, en el módulo cliente del corrector se ha implementado la ordenación de las propiedades utilizando una clase llamada `PropertyErrorComparator` que implementa la interfaz `Comparator<Property>`, lo cual nos permite utilizar el método "`sort()`" de `List` en cualquier lista de propiedades simplemente pasando una instancia como parámetro. Para obtener el orden deseado se utiliza un `Map` de clave `PropertyEnum` y su peso en forma de entero como valor. Como se puede observar en el método "`compare`", aquellos con mayor peso quedarán antes en la lista, corrigiéndose los primeros.

```java
public class PropertyErrorComparator implements Comparator<Property> {

    private final Map<PropertyEnum, Integer> errorWeights = Stream.of(
            new Object[][] {
                    {PropertyEnum.MethodSize, 0},
                    {PropertyEnum.SimpleName, 1},
                    {PropertyEnum.PrimitiveType, 2},
                    {PropertyEnum.ClassOrInterfaceType, 3},
                    {PropertyEnum.MethodDeclaration, 4},
                    {PropertyEnum.Modifier, 5},
                    {PropertyEnum.ClassOrInterfaceDeclaration, 6},
                    {PropertyEnum.Parameters, 7},
                    {PropertyEnum.TypeParameter, 8},
                    {PropertyEnum.ConstructorDeclaration, 9},
                    {PropertyEnum.ConstructorSize, 10},
                    {PropertyEnum.Name, 11},
                    {PropertyEnum.PackageDeclaration, 12},
                    {PropertyEnum.IsInterface, 13},
                    {PropertyEnum.ExtendedTypes, 15},
                    {PropertyEnum.VoidType, 15},
                    {PropertyEnum.ImplementedTypes, 16},
                    {PropertyEnum.ImportDeclaration, 17},
            }
    ).collect(Collectors.toMap(data -> (PropertyEnum)data[0], data -> (Integer) data[1]));

    @Override
    public int compare(Property propertyType, Property t1) {
        return errorWeights.getOrDefault(t1.getPropertyType(), 30) -
          errorWeights.getOrDefault(propertyType.getPropertyType(), 30);
    }
}
```

Figura 31 Clase comparadora para ordenar propiedades

5.1.2 Fase de corrección

Toda la funcionalidad correspondiente a la corrección se encuentra implementada en el módulo *Codeassess*. Se ha encapsulado toda la lógica desarrollada en un paquete *solver*, dentro del cual encontramos las clases que se encargan de modificar el código, de comprobar y revertir errores, una clase que representa a la propiedad ya resuelta, excepciones de propiedades e incluso ejercicios incorregibles, además de clases de ayuda para comparar métodos y constructores en el paquete *signature*. En la figura siguiente se muestra la estructura final de las clases implementadas:

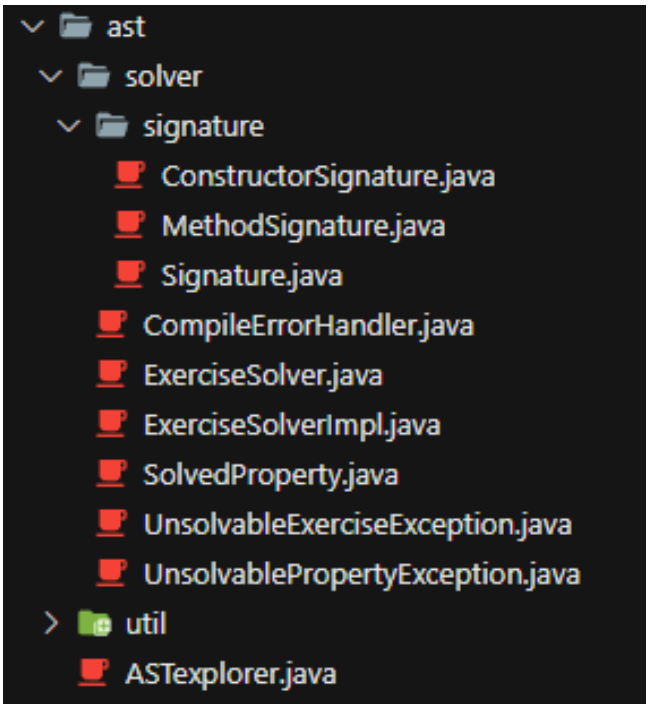


Figura 32 Clases creadas para la corrección

El funcionamiento previo al desarrollo estaba compuesto solamente de un método "`solve`" en la clase `ASTexplorer`, al que se le pasaba el índice de la propiedad y devolvía `true` si había podido corregir o `false` en caso contrario. Esta dinámica se ha mantenido, con la diferencia que ahora ese método "`solve`" en lugar de contener toda la lógica, se limita a crear un objeto de tipo `ExerciseSolver` para manejar la corrección, a llamar al controlador de errores definido en la clase `CompileErrorHandler` y a capturar las excepciones de propiedad o ejercicio irresolubles, en cuyo caso se devuelve `false`, quedando como resultado los métodos visibles en la siguiente figura:

```java
/**
 * == SOLVE I == Soluciona la i-property del directorio proyecto del alumno
 * @return boolean
 */
public boolean solve(int propIndex) {
    PropertyTree propertyTree = result.get(propIndex);

    //This property is not in the results
    if (propertyTree == null) {
        return false;
    }

    //There is either no need for change or it has already been corrected
    if (Objects.equals(propertyTree.solution, propertyTree.project)) {
        propertiesToSolve--;
        return true;
    }

    ExerciseSolver exerciseSolver = new ExerciseSolverImpl(propertyTree, result.values());
    try {
        SolvedProperty solvedProperty = new SolvedProperty(propIndex, propertyTree);
        PropertyTree solvedTree = exerciseSolver.getSolvedPropertyTree();
        result.put(propIndex, solvedTree);
        project = solvedTree.getpCU();
        try {
            handleCompileErrors(solvedProperty);
        } catch (UnsolvableExerciseException e) {
            project = originalProject;
            addUnsolvableComment();
        }
        return true;
    } catch (UnsolvablePropertyException e) {
        propertiesToSolve--;
        return false;
    }
}

private void handleCompileErrors(SolvedProperty solvedProperty) throws UnsolvableExerciseException {
    CompileErrorHandler errorHandler = new CompileErrorHandler(project, solveFile, result);
    project = errorHandler.handleErrors(compileErrorProperties, propertiesToSolve, solvedProperty);
}

private void addUnsolvableComment() {
    project.getPrimaryType().ifPresent(typeDeclaration -> {
        typeDeclaration.setLineComment("Could not correct this exercise");
    });
}
```

Figura 33 Código de ASTexplorer.java que controla la corrección

Antes de comenzar con la corrección en sí misma, se comprueba que la propiedad no sea nula para controlar errores y que la solución del alumno y del profesor no sean iguales. ¿Por qué se hace esto último? Porque se pueden dar casos donde la corrección de una propiedad afecte a otra, haciendo que se corrija antes de tiempo.

Por ejemplo, supongamos que el alumno envía de nuevo una solución errónea a la clase Rectángulo del ejercicio de las figuras, entre cuyos errores se encuentran que no se ha añadido la herencia necesaria a la clase `Figura` y también que, por despiste, ha hecho que el atributo "`posX`" sea de visibilidad pública:

```java
public class Rectangulo {
    private float base;
    private float altura;
    public int posX;
    private int posY;
    private String color;

    public Rectangulo(String color, float base, float altura) {
        this.posX = 0;
        this.posY = 0;
        this.color = color;
        this.base = base;
        this.altura = altura;
    }

    public Rectangulo(int posX, int posY, String color, float base, float altura) {
        this.posX = posX;
        this.posY = posY;
        this.color = color;
        this.base = base;
        this.altura = altura;
    }

    public float area() {
        return base * altura;
    }
}
```

Figura 34 Clase Rectángulo con errores de herencia y modificadores

Cuando se corrija esta clase, en primer lugar, se añadirá la herencia a `Figura`, lo que provocará que los atributos "`posX`", "`posY`" y "`color`" sean eliminados de `Rectangulo`, dado que los tres están presentes en `Figura` y en el código del profesor no se utiliza ocultación.

A pesar de ello, en los resultados seguirá habiendo una propiedad que intentará corregir el modificador público en el atributo "`posX`", pero no será necesario, ya que se ha corregido indirectamente por otra la propiedad de herencia.

Una vez se ha corregido la propiedad, se sustituye la versión incorrecta en el `Map` de resultados. De esta forma, cuando esta se requiera nuevamente para deshacer errores, se encontrará presente en su última versión.

Se debe trabajar con todo el `Map` ya que a veces es necesario poder consultar propiedades diferentes además de la que se está corrigiendo, puesto que puede ser clave para poder obtener la información necesaria para arreglar el código como, por ejemplo, cuando añades una herencia faltante a una clase y se deben eliminar los atributos presentes en la superclase, la cual es inaccesible solamente con esa propiedad.

A continuación, vamos a explorar la clase con la funcionalidad clave de este trabajo, `ExerciseSolverImpl`, que implementa la interfaz `ExerciseSolver`. En este caso, la interfaz contiene solamente un método "`getSolvedPropertyTree()`", que devuelve el objeto con la propiedad corregida. En caso contrario, lanza la excepción `UnsolvablePropertyException`. Toda la lógica y funcionalidad se encuentra aislada en `ExerciseSolverImpl`, para que, de esta forma, corregir sea tan simple como crear una instancia y llamar únicamente al método de la interfaz, el cual podemos ver en la siguiente figura:

```java
package codeassess.javassessment.ast.solver;

import codeassess.javassessment.ast.PropertyTree;

public interface ExerciseSolver {
    PropertyTree getSolvedPropertyTree() throws UnsolvablePropertyException;
}
```

Figura 35 Interfaz ExerciseSolver.java

Por su parte, la implementación de este método en la clase `ExerciseSolverImpl` resulta bastante intuitiva. Se llama al método privado "`solve()`", que es quien realmente realiza la corrección. Si este nos devuelve `true`, entonces tenemos una propiedad corregida y la devolvemos. En caso contrario, lanzamos manualmente la excepción `UnsolvablePropertyException`:

```java
@Override
public PropertyTree getSolvedPropertyTree() throws UnsolvablePropertyException {
    if (solve()) {
        return propertyTree;
    } else {
        throw new UnsolvablePropertyException(propertyTree);
    }
}

private boolean solve() {
    if (solution instanceof NodeList) {
        if (project instanceof NodeList) {
            return replacePropertyList();
        }
        if (project == null) {
            return false;
        }
    }

    switch (propertyTree.getType()) {
        case SimpleName:
        case PrimitiveType:
        case ClassOrInterfaceType:
        case Modifier:
            return handleSingleNodeCorrection();
        case MethodSize:
            return solveMethodSize();
        case ConstructorSize:
            return solveConstructorSize();
        case ImportDeclaration:
            return addMissingImports();
        case Name:
            if (solution instanceof ImportDeclaration) {
                return addMissingImports();
            }
            break;
        case MethodDeclaration:
        case ConstructorDeclaration:
            return solveCallableDeclaration();
        case IsInterface:
            return solveClassOrInterfaceError();
        case Parameters:
            return solveParameters();
        case VoidType:
            return makeMethodVoid();
    }
    return false;
}
```

Figura 36 Método de corrección en ExerciseSolverImpl.java

El método "`solve()`" sirve como una puerta de entrada para la corrección. Hay dos criterios para decidir qué tipo de corrección se debe aplicar: el objeto que represente a las soluciones del alumno y profesor, y el tipo de propiedad a resolver, que como pudimos ver en el funcionamiento del módulo de detección de errores, resulta de la información más importante de cada propiedad.

A continuación, vamos a indagar en cada propiedad y cómo se lleva a cabo su corrección. La Tabla 2 muestra un resumen de todas las correcciones del módulo corrector. Cada una de ellas es explicada en los siguientes apartados.

Correcciones
Corrección de nombres de atributos y métodos
Corrección de tipos
Corrección de modificadores
Corrección de número de métodos y constructores
Corrección de *imports*
Corrección de declaraciones de métodos y constructores
Correción de clase a interfaz y viceversa
Correcciones de métodos relacionadas con el tipo *void*
Corrección de herencia e implementación de interfaces
Corrección de otras propiedades en formato de lista

Tabla 2 Correcciones realizadas por el módulo corrector

5.1.2.1 Corrección de nombres de atributos y métodos

En la corrección de nombres pueden darse tres casos:

1. Tanto el atributo proyecto como la solución están presentes.
2. Solamente el atributo proyecto está presente.
3. Solamente el atributo solución está presente.

La corrección del primer caso es la más simple, se sustituye el nodo incorrecto por la solución y seguidamente se explora el árbol de la clase para reemplazar el resto de las ocurrencias, evitando así errores de compilación. Esto aplica tanto a las apariciones de los atributos como a las llamadas a los métodos. Se trata por tanto del caso ideal en este tipo de corrección.

En el segundo caso se ha optado por no actuar en caso de que el detector de errores envíe una propiedad con estas características, ya que implicaría eliminar un atributo o método del código, el cual no sabemos si ha sido utilizado por el alumno como herramienta auxiliar para resolver el problema en el cuerpo de algún método.

Por último, en el caso de que solo la solución esté presente, puesto que el atributo proyecto es nulo, nos impide obtener información de a dónde debemos añadir este elemento en el código del alumno. En este caso, lo que debemos hacer es lo siguiente:

1. Obtener el nodo padre del nodo solución en el código del profesor
2. Si el nodo padre es una declaración de método, añadimos la función al código del alumno.
3. Si el nodo padre es un atributo, añadimos el atributo.

En la mayoría de los casos habríamos terminado, pero se puede dar una situación más con el proyecto nulo: que el nodo padre sea un parámetro. En este caso, como no tenemos acceso al método o constructor del código del alumno al que le falta dicho parámetro, debemos intentar buscarlo.

Para buscarlo, intentamos obtener el constructor o método del alumno cuya firma (número y tipo de argumentos en caso de los constructores y lo mismo más el nombre en el caso de los métodos) coincida con el método del profesor, al que sí tenemos acceso, menos el parámetro faltante. Por ejemplo, con este método de búsqueda, el constructor de la izquierda (profesor) y el de la derecha (alumno), serían iguales.

```java
public class Figura {

  public Figura(int posX, int posY, String color) {
    this.posX = posX;
    this.posY = posY;
    this.color = color;
  }

}
```

```java
public class Figura {

  //SE BUSCA AÑADIR "STRING COLOR"
  public Figura(int posX, int posY) {
    this.posX = posX;
    this.posY = posY;
  }

}
```

Figura 37 Constructores equivalentes al añadir el parámetro "color"

El caso de los parámetros muestra una de las debilidades del detector de errores y, por consecuencia, del corrector en la actualidad. En la sección de trabajo futuro se indagará más en este aspecto y sus posibles soluciones.

5.1.2.2 Corrección de tipos

La corrección de tipos no dista demasiado de lo que ocurre con los nombres. Cuando tenemos disponibles ambos, el tipo incorrecto del alumno y el correcto del profesor, basta con sustituirlos, ya sean atributos o parámetros. La misma aproximación se toma cuando la solución es nula y, por tanto, muestra que “sobra” un elemento.

En los casos en los que tenemos el proyecto nulo, es decir, que falta un elemento, en primer lugar, debemos obtener el nodo padre para saber qué tipo de elemento falta: un atributo, un método o un parámetro.

La corrección sigue exactamente el mismo mecanismo que con la de nombres, pero debemos tener en cuenta ambos casos puesto que, en general, el profesor no va a obligar a que los atributos o métodos tengan los mismos nombres, así que ASys debe poder añadir atributos, métodos o parámetros sin depender de ellos.

5.1.2.3 Corrección de modificadores

Los modificadores se comportan igual que nombres y métodos, salvo que en este caso debemos tener en cuenta algunos detalles más.

Siempre que añadamos un modificador de visibilidad, debemos asegurarnos de que, de haber uno, debe ser eliminado, ya que el detector no distingue entre modificadores normales (`abstract`, `final`, etc.) y modificadores de visibilidad (`public`, `protected`).

Si el modificador es de método y es `abstract`, habrá que eliminar el cuerpo del método. En este caso también puede parecer necesario añadir el modificador `abstract` a la clase, pero el detector se encargará de ello en una propiedad diferente.

5.1.2.4 Corrección de número de métodos y constructores

Este tipo de corrección es una de las más peligrosas de cara a dar flexibilidad al alumno, ya que obliga a que se implemente un número exacto de métodos o constructores.

La corrección se resume en dos posibles casos:

1. Si el alumno tiene más métodos o constructores, se buscan aquellos que no se encuentran en la solución, comparando mediante las signaturas de estos, y se eliminan hasta que haya el número correcto.
2. Si el alumno tiene menos métodos o constructores, se van añadiendo aquellos que no se encuentren en su código hasta que se cumpla el número deseado, sin importar que los previos del alumno no estén en la solución. Por ende, aquellos que se encontraran previamente no son eliminados, incluso si estos no pertenecen a la solución del profesor.

5.1.2.5 Corrección de `Imports`

Para corregir los `imports`, el detector puede darnos información de dos maneras diferentes, o con un nodo por `import` individual, o con una lista de nodos para toda la declaración. En cualquier caso, simplemente se añadirán todos ellos sin eliminar ninguno del alumno, con el fin de no provocar errores de compilación.

5.1.2.6 Corrección de declaraciones de métodos y constructores

Las propiedades de tipo declaración de método o constructor son aquellas generadas cuando el profesor define que alguno de estos dos elementos debe tener exactamente la misma declaración que la solución.

Esto significa que si, por ejemplo, en lugar de declarar un método como público este se declara como privado, o, por otro lado, se tiene un atributo de más, ASys no intentará corregir estas propiedades de forma individual, sino que lo sustituirá al completo.

En el caso de que el atributo solución sea nulo, se eliminará el constructor o método correspondiente. Si, por el contrario, es el atributo solución el que se recibe como nulo, simplemente se añadirá este elemento a la clase.

Cuando ambos están presentes, significa que se deberán sustituir. No obstante, al contario que, con los métodos, tipos o modificadores, aquí estamos trabajando con los nodos que contienen los métodos o constructores al completo, lo que significa que, si fuéramos a simplemente sustituir la versión del alumno por la del profesor, estaríamos eliminando la lógica del alumno.

Desde luego, hacer esto no tendría ningún sentido, ya que, por el funcionamiento de esta propiedad, marcará como incorrecto un método al completo incluso si solamente es errónea su visibilidad. Por ende, se ha decidido que cuando se realice esta corrección y se deban intercambiar declaraciones del alumno y profesor, se mantenga la lógica del cuerpo escrita por el alumno, lo cual nos permite hacer JavaParser con el método "`getBody()`" presente en este tipo de elementos.

5.1.2.7 Corrección de clase a interfaz y viceversa

Este tipo de corrección se lleva a cabo cuando el profesor ha establecido que cierto archivo Java contendría una interfaz o una clase, y sin embargo el alumno ha puesto lo contrario.

El módulo detector de errores simplemente nos informa mediante valores booleanos: el atributo solución representando si este archivo debiera contener una interfaz y el atributo proyecto indicando si, en efecto, la contiene.

Cuando se ha de transformar una clase en una interfaz resulta sencillo, ya que basta con iterar por todos los miembros de la previamente incorrecta clase, eliminando los atributos y los cuerpos de los métodos, junto con cambiar el valor booleano del nodo que indica si esta unidad de compilación es una interfaz, el cual podemos observarlo en el ejemplo de árbol sintáctico presentado con anterioridad.

El problema viene cuando debemos transformar una interfaz en una clase, porque no basta con añadir cuerpos vacíos a los métodos, y resulta imposible añadir valores de retorno por defecto ya que no todos los posibles objetos de Java tienen un constructor por defecto.

La solución escogida ha sido que una vez se transforma una interfaz en una clase, el cuerpo de esta pasa a ser el del código del profesor, simplificando así el proceso y garantizando que el código generado es correcto.

5.1.2.8 Correcciones de métodos relacionadas con el tipo `void`

Cuando un método debe ser `void` y no lo es, se sustituye el tipo actual y, además, se elimina la línea de retorno del método, para así prevenir errores de compilación.

Existen ocasiones en las que el detector de errores enviará esta propiedad con el valor del atributo proyecto a nulo, lo que significa que un método `void` no se encontraba en el código del alumno. Cuando ocurre, basta con añadirlos.

5.1.2.9 Corrección de herencia e implementación de interfaces

Tanto las interfaces como la herencia son correcciones con las que basta realizar una sustitución de nodos, tanto si había una herencia incorrecta, como si hay que añadirla, puesto que en este caso el detector de errores jamás envía valores nulos, sino listas vacías, facilitando mucho el trabajo.

Cuando se añade o corrige una herencia, se hacen dos procesos adicionales: adaptar las sentencias de "`super`" de los constructores y, si el profesor no había utilizado ocultación, eliminar los atributos que ya estén presentes en la superclase.

Para corregir las sentencias "`super`", se exploran tanto los constructores del alumno como los del profesor, para compararlos y encontrar sus equivalentes. Si se encuentran constructores con la misma declaración, en la versión del alumno se eliminarán las líneas sobrantes y añadirán aquellas que no estén. Resulta de una corrección peligrosa, en el sentido de que puede generar errores de compilación, siempre y cuando los atributos del alumno difieran en nombre de los del profesor, pero de no hacerla, seguiría habiendo errores de compilación por la falta de la sentencia "`super`" correcta. Se hablará más en profundidad de este tema en el apartado de trabajo futuro.

La eliminación de los atributos sobrantes se realiza, en primer lugar, obteniendo la superclase. Esto lo hacemos buscando en los objetos `PropertyTree` de la lista de resultados. Solamente trabajamos con la clase directamente superior para realizar este trabajo, ya que, de buscar en todas las clases superiores, el tiempo de ejecución no escalaría adecuadamente en ejercicios mayores. Además, como no se trata de un error que impida la compilación, no resulta crítico hacerlo.

Una vez tenemos la clase y superclase del alumno, junto con la clase del profesor, obtenemos aquellos atributos de la clase del alumno que no estén en la clase del profesor, pero sí en la superclase del alumno. Por último, esos atributos son eliminados de la clase del alumno, ya que no son necesarios.

En el caso de las interfaces, se ha decidido simplemente añadirla a la declaración. En ningún caso se han añadido los métodos ni de las clases abstractas ni de las interfaces al mismo tiempo que se han añadido sus declaraciones. Esto se debe a que esas correcciones se hacen independientemente mediante otras propiedades, como pueden ser los tipos o los nombres, por lo que implementarlo aquí sería redundante.

5.1.2.10 Corrección de otras propiedades en formato de lista

Cuando los atributos proyecto y solución son listas de nodos, quiere decir que son propiedades en las que se ha establecido que la solución debe ser exactamente esa lista, como puede ser una lista de `extends`, `imports`, o modificadores en un método o atributo. Por tanto, siempre y cuando ambas soluciones tengan como tipo una lista de nodos, el método de corrección será el mismo, sustituir los elementos del profesor por los del alumno, proceso que es gestionado por el método `replacePropertyList`.

Hablando de tipos, como corrección es de las más sencillas cuando se trata de atributos. El único problema es que se le haya asignado incorrectamente un tipo incompatible en el cuerpo de algún método o en la misma declaración (por ejemplo, el tipo correcto era un `boolean` y el alumno puso un entero), como se muestra en el apartado de análisis del problema.

Para las propiedades de tipo *VoidType* basta con eliminar las líneas de `return` de los métodos y cambiar el tipo a `void`. Algo parecido debe hacerse también cuando una interfaz es incorrectamente una clase: se cambia el tipo a interfaz y se eliminan los cuerpos de todos los métodos. Otra corrección con una aproximación similar es cuando un método debe ser abstracto y no lo es. En este caso se añade el modificador y se elimina el cuerpo del método. No se hace nada a la clase al añadir el modificador porque se presupone que o ya es abstracta u otra propiedad se encargará de que lo sea.

Las propiedades de tamaño de constructores y métodos establecen que una clase debe tener un número fijo de cualquiera de estos elementos, por lo cual suelen ser desaconsejables si se quiere dar suficiente flexibilidad a los alumnos para resolver los problemas a su propia manera. En caso de estar activadas, se buscaría en la solución del profesor qué métodos faltan o sobran, añadiendo o eliminado respectivamente de la solución del alumno. Para la correcta identificación de los métodos no se utilizan los nombres, ya que debemos tener en cuenta el polimorfismo de sobrecarga y que dos métodos con el mismo nombre pueden no ser iguales dados sus argumentos.

Para este caso se utilizan las clases `MethodSignature` y `ConstructorSignature`, que nos sirven para comparar e identificar estos elementos.

5.1.3 Fase de manejo de errores de compilación

El control de errores de compilación se realiza en la clase `CompileErrorHandler`. Este control tiene dos fases, en función del momento de la corrección en el que nos encontremos.

Mientras todavía se esté llevando a cabo la corrección, se limita a compilar la solución del alumno con las correcciones hasta el momento y almacenar el resultado en una lista de objetos `SolvedProperty`. Este objeto contiene el índice de la propiedad corregida, las soluciones del alumno y profesor, el tipo de propiedad y el resultado de la compilación tras su corrección.

Desde el principio de la corrección se lleva cuenta del número de propiedades que deben ser corregidas utilizando la variable `propertiesToSolve`, siendo este número reducido en los casos en los que una propiedad se vio resuelta en la corrección de otra o si alguna ha resultado ser irreducible, como podemos observar en la Figura 8.

Sabemos que se ha completado la corrección cuando la longitud de la lista de elementos de tipo `SolvedProperty` es igual a la variable `propertiesToSolve`. Es en este momento en el que se inicia la segunda fase del control de errores.

A continuación, entramos en un bucle que se mantiene mientras todavía haya propiedades que no compilen en la lista. Dentro del mismo, obtenemos la raíz del error de compilación (utilizando el método que se ha explicado en el apartado de diseño de la solución) y la deshacemos. La forma de deshacer resulta bastante simple, ya que basta con corregir la propiedad con la solución del alumno y del profesor invertidas. Si compila después, hemos terminado y devolvemos los resultados. En caso contrario eliminamos esa propiedad de la lista y seguimos intentando deshacer las siguientes.

5.2 Migración de la comunicación de resultados a JSON

En la versión previa a este trabajo, cuando el módulo de corrección completaba el proceso de evaluación del código del alumno, la forma en la que se estructuraban los resultados para enviarlos a la interfaz web en Vue.js no seguía ningún formato estándar como lo puede ser XML o JSON. En adelante usaremos el término “formato propietario” para referirnos al formato utilizado por el módulo de corrección de la versión anterior de ASys para encapsular y enviar el resultado de la corrección al sistema.

Realmente no hay ventajas reales en haber tomado esta decisión de diseño, por lo que rehacerlo para adaptarlo a un formato estándar habría tenido sentido incluso como decisión fuera de este proyecto.

Figura 38 Ejemplo de respuesta con el formato propietario

5.2.1 Desventajas del formato propietario

A continuación, vamos a enumerar los problemas que se encontraban presentes debido al uso del formato propietario:

1. Gran cantidad de código destinado solamente al formateo de objetos Java a una cadena de caracteres. El formateo podría delegarse a una librería que construyera información en un formato estándar y a la que solo habría que pasarle los datos.
2. Poca eficiencia. Al tratarse el cliente de un programa JavaScript, si se hubiera elegido JSON como formato, se podría obviar por completo cualquier método de *parsing* de la información. Con el estado actual, se debe serializar a mano en el corrector y deserializar a mano en el cliente.
3. Poca flexibilidad, escalabilidad reducida. Este motivo en solitario habría sido suficiente para realizar este cambio, ya que con el formato propietario resultaba muy complicado e innecesariamente complejo el añadir nuevos campos o información mientras el sistema crece.

5.2.2 Implementación del nuevo formato

Como nuevo formato para la comunicación, JSON (Mozilla Foundation, 2022) resulta ser la opción la opción más adecuada, tanto por su sencillez como por la compatibilidad nativa con el Front-End, eliminando cualquier necesidad de añadir librerías en este módulo.

Para el corrector, sin embargo, se ha adoptado la librería de Google "GSON" (Google, 2022) por su facilidad de uso, rendimiento y ligereza, lo que la ha convertido en la librería de JSON más utilizada en entornos Android, basándonos en la diferencia de número de estrellas en los repositorios de GitHub con las de sus competidores, Moshi (Square, 2022) y Jackson (FasterXML, 2022).

Los métodos de corrección de propiedades y del ejercicio al completo devolvían una estructura de datos `Map` de clave y valor de tipo `String`, lo cual tenía sentido si se tenía que serializar a una cadena de texto para construir la respuesta en el formato anterior, pero con los cambios actuales y la adaptación a JSON se ha decidido crear tres nuevos objetos:

1. `CheckResult`*:* Para almacenar los resultados de la comprobación de errores de una propiedad.
2. `SolveResult`*:* Para almacenar y enviar los resultados de la corrección del código del alumno.
3. `AssessmentResult`*:* El objeto que constituye el resultado de todas las fases de corrección de ASys.

```java
public class AssessmentResult {
    private final String assessmentMode;
    private final List<CheckResult> checkResults;
    private final List<SolveResult> solveResults;

    private double mark;

    /*
        CONSTRUCTORS, GETTERS AND SETTERS
    */

    public JsonObject toJson() {
        JsonObject result = new JsonObject();
        result.addProperty("assessmentMode", assessmentMode);
        if (mark != -1) {
            result.addProperty("mark", mark);
        }

        JsonArray checkArray = new JsonArray();
        checkResults.forEach(checkResult ->
            checkArray.add(checkResult.toJson())
        );
        result.add("check", checkArray);

        JsonArray solveArray = new JsonArray();
        solveResults.forEach(solveResult ->
            solveArray.add(solveResult.toJson())
        );
        result.add("solve", solveArray);

        result.add("test", new JsonArray());

        return result;
    }
}
```

```java
public class CheckResult {
    private final String propertyId;
    private final String result;
    private final String hint;
    private final String phase;
    private final double mark;
    private final double highestMark;
    private final List<Review> reviews;

  /*
    CONSTRUCTORS, GETTERS, SETTERS, AND toJson
  */
}
```

```java
public class SolveResult {
    private final String filename;
    private final String code;

  /*
    CONSTRUCTORS, GETTERS, SETTERS, AND toJson
  */
}
```

Figura 39 Nuevas clases para representar los resultados de corrección

Todos los objetos van acompañados de un método "`toJson()`" que como su nombre indica, sirve para construir la respuesta final. Otra de las ventajas que nos ofrece esta aproximación es que en estos métodos podemos dedicarnos solamente a la lógica de la aplicación y el problema, relegando el código relacionado con la construcción de las respuestas a GSON y a los pequeños métodos presentes en cada objeto serializado.

Figura 40 Ejemplo de respuesta con el nuevo formato en JSON

5.3 Desarrollo de la presentación de resultados en la interfaz

Hasta el momento se tenía un módulo de corrección desarrollado que, tras detectar y evaluar los errores del alumno, era capaz de solucionar dichos errores en el propio código del alumno, y el resultado de esta corrección era enviado a la interfaz web, pero todavía había que implementar en la interfaz que estas correcciones fueran accesibles para los usuarios.

En la interfaz actual (véase la Figura 41), se tiene un entorno de edición de código a través del cual el alumno puede desarrollar su solución y corregirla en cualquier momento. Este entorno se divide en tres partes: a la izquierda, el árbol de ficheros; en el centro, el propio editor, en el cual se abren pestañas según se seleccionen archivos, como podemos esperar de cualquier editor de texto actual; y en la parte derecha, tres pestañas fijas que actualizan su contenido en función del fichero a editar:

1. La pestaña “Solución”, la cual aparece solamente en el contexto de un profesor y que muestra la versión del profesor de dicho archivo.

2. La pestaña “Solución Original”, que enseña el código que envió el alumno para la última corrección.
3. La pestaña “Test” que en el futuro mostrará los casos de prueba para la clase seleccionada.

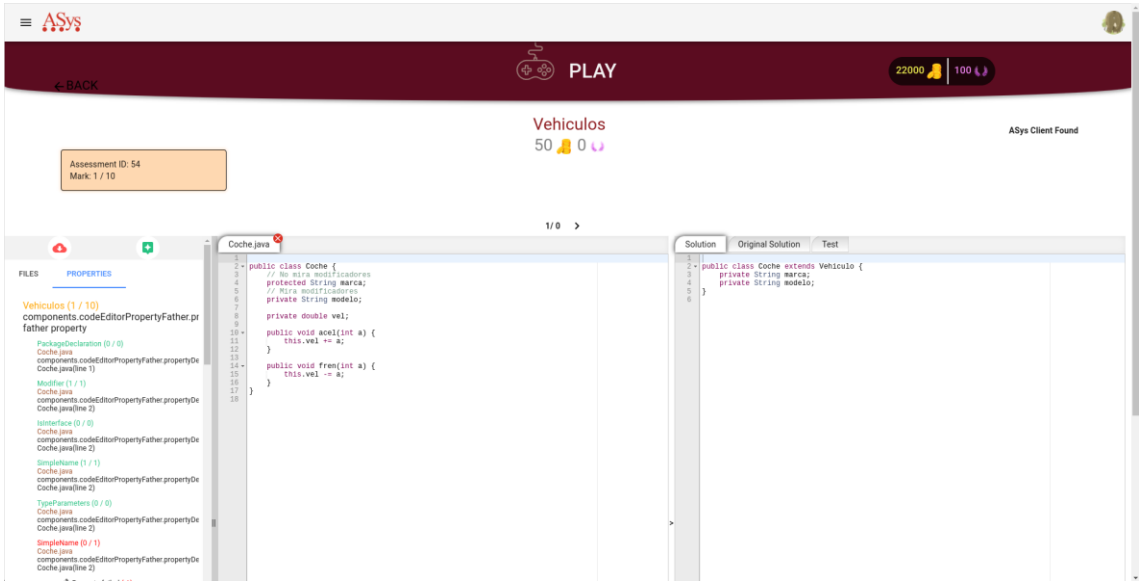


Figura 41 Captura de pantalla del editor de texto sin cambios

Por tanto, se ha decidido que, para integrar este trabajo con la interfaz, se ha de añadir una nueva pestaña fija en la parte derecha, que se llamará “Corrección” y mostrará, al igual que el resto de las pestañas de la sección, el código del alumno corregido por este trabajo para cada archivo que se tenga seleccionado.

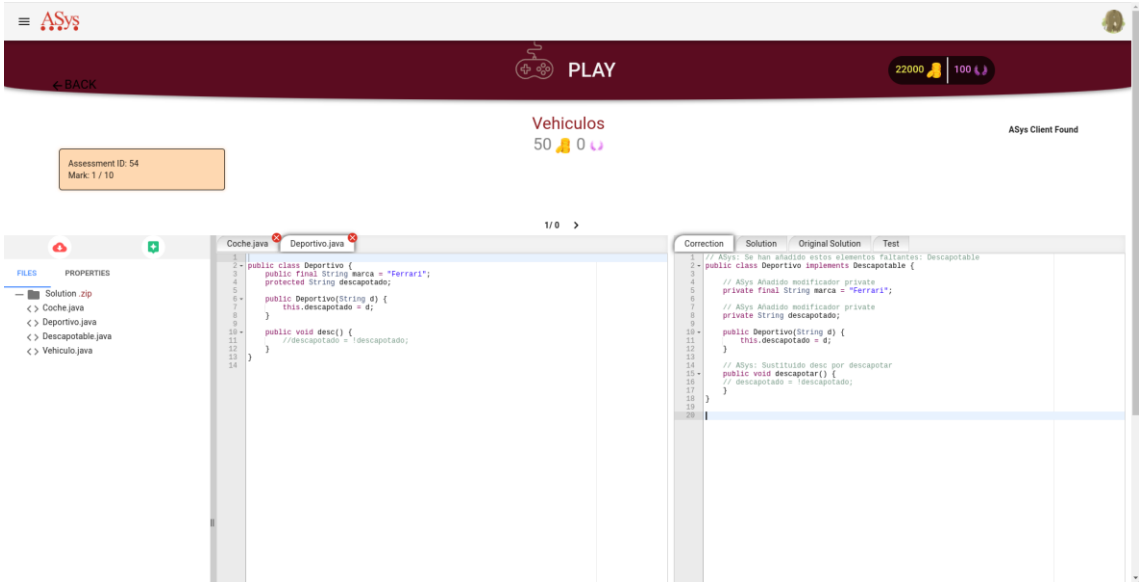


Figura 42 Captura de pantalla del editor con la nueva pestaña de Corrección

[redacted] Implantación

En esta sección se profundizará en cómo llevar a cabo el despliegue del sistema ASys, y, por tanto, del trabajo actual en un entorno de producción, tanto los requisitos previos del sistema, como qué software debe estar instalado, como los pasos a seguir para su ejecución.

6.1 Requisitos software

Dependiendo del servidor que estemos tratando, este deberá tener cierto *software* instalado. En esta guía sin embargo se asumirá un despliegue en una sola máquina.

6.1.1 Servidor web ASys-Web-Server

Para el servidor, se necesitan algunos servicios más. Algunos de estos se pueden dividir entre diferentes máquinas, pero en este caso asumiremos que se ejecutará en una sola.

- Docker y Docker Compose (Docker, Inc, 2022)
- MySQL o algún programa que facilite su gestión y/o instalación. En este caso, utilizaremos XAMPP (Apache Friends, 2022)
- JDK 1.8
- Tomcat (Apache Software Foundation, 2022)
- Maven 3.2 o superior (Apache Software Foundation, 2022)

6.1.2 Servidor cliente ASys-Web-Client

En el caso del cliente, los únicos requisitos previos para su instalación y ejecución son:

- Node JS (Mozilla Foundation, 2020)
- NPM (npm, Inc, 2022)

6.1.3 Servidor de corrección ASys-Corrector

En el caso del corrector necesitamos los siguientes paquetes:

- JDK 1.8
- Ant (Apache Software Foundation, 2021)

6.2 Configuración e instalación

Ahora detallaremos los pasos a seguir para realizar la configuración del sistema.

6.2.1 Servidor web ASys-Web-Server

El servidor tiene su configuración almacenada en el archivo "application.yml", junto con algunos detalles que en otros archivos ".java". Por lo tanto, deberemos seguir estos pasos:

1. En primer lugar, abriremos "application.yml", presente en la carpeta "/src/main/resources".
2. Debajo de las secciones "spring" y "datasource" encontraremos las opciones relacionadas con la base de datos, como lo son el usuario, la contraseña y su URL.
3. Para la base de datos también es necesario sustituir la propiedad "ddl-auto" en la sección "jpa/hibernate" al valor "none". Esto se debe a que esta propiedad controla qué se hace en la base de datos al iniciar y parar el servidor web, como crearla, eliminarla o validarla. Para el entorno de producción nos interesa que no se haga nada con la misma.
4. Más abajo podemos observar las opciones de CORS, que manejan qué *hosts* tienen permisos para hacer peticiones a nuestra API. Aquí deberemos colocar la dirección desde la que el cliente desplegado hará peticiones.
5. Fuera de "application.yml", deberemos modificar el archivo de configuración del servidor SMTP "src/main/java/com/asysweb/útil/JavaEmailUtil.java", donde estarán nuestras credenciales.
6. Además, si queremos cambiar las opciones relativas a la conexión con el servidor de archivos MinIO, tendremos que modificarlas en ambos, el archivo Java "src/main/java/com/asysweb/AmazonS3ServiceImpl.java" y el "docker-compose.yml" presente en la raíz del repositorio del servidor.

7. Por último, construimos el proyecto en formato ".war".

6.2.2 Servidor cliente ASys-Web-Client

En segundo lugar, configuraremos el servidor cliente de ASys de la siguiente forma:

1. Ejecutaremos "npm install" en la raíz del proyecto para instalar todas las dependencias necesarias.
2. Modificaremos las variables "baseURL" y "apiPath" de los ficheros "src/services/connection.js" y "src/store/index.js" respectivamente con la URL en donde se desplegará el servidor web.
3. Generaremos el directorio "/dist" en donde se encontrará el proyecto construido y preparado para producción ejecutando "npm run build" de nuevo desde la raíz del proyecto.

6.2.3 Corrector de ejercicios ASys-Corrector

Para el corrector, basta con utilizar Ant para construir los tres subproyectos, el cliente, el subproyecto de corrección y el de miscelánea. Por defecto, el formato configurado es ".jar".

Antes de construir el módulo, sin embargo, deberemos cambiar todas las opciones dependientes de la máquina, como las rutas de archivos del proyecto o el directorio en donde se debe generar el archivo a rutas que coincidan con el entorno en donde se va a construir.

6.2.4 Servidores adicionales

Además de los servicios propios de ASys, se necesitan servidores adicionales para su funcionamiento, como el servidor de base de datos y el de almacenamiento de archivos MinIO, además del servidor web Tomcat. Para su instalación y configuración se deben seguir los siguientes pasos:

1. Para lanzar el servidor de archivos MinIO basta con tener instalado Docker y Docker Compose, por lo que lo único que debemos hacer es asegurarnos de que la configuración coincide con la del archivo "AmazonS3ServiceImpl.java" antes de ejecutar "docker-compose up".
2. Para el servidor de bases de datos dependerá del gestor que se tenga instalado con MySQL. En el caso de este trabajo se utilizó el programa XAMPP. Una vez iniciada la base de datos se deberá configurar el usuario, contraseña y ejecutar el *script* que inicializará las tablas.

3. Para preparar Tomcat, añadiremos el fichero “.war” del servidor y la carpeta “/dist” del cliente a la carpeta “webapps”. Tras esto, utilizando el archivo “startup” podremos iniciar ASys. El corrector deberá iniciarse en el ordenador del alumno.

En el caso de este trabajo, para asegurar la disponibilidad, el sistema se va a alojar en el servidor Kaz del departamento DSIC. Ese servidor tiene copias de seguridad en espejo, independencia energética y todas las actualizaciones de seguridad que garantizan su integridad.

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Pruebas

Para validar el funcionamiento, detectar errores y verificar si el resultado se corresponde con la Especificación de Requisitos Software realizada, se ha sometido este proyecto a distintas series de pruebas de diferente tipo a lo largo del proceso de desarrollo.

Se llevaron a cabo pruebas unitarias para validar el funcionamiento de correcciones aisladas, pruebas de integración para comprobar que el sistema responde correctamente a correcciones donde intervienen múltiples propiedades y funciones, y, por último, pruebas de aceptación, donde junto con el tutor del trabajo se ha validado que el producto se comporta de la forma esperada.

Se han realizado junto con el tutor varias sesiones de pruebas de aceptación en las que se ha comprobado el comportamiento del sistema en tiempo real para verificar su correcto funcionamiento como producto final.

7.1 Pruebas unitarias

En el ámbito de este proyecto, las pruebas unitarias nos permiten comprobar el funcionamiento de correcciones individuales, como puede ser añadir una herencia, sustituir el nombre de un atributo incorrecto o añadir un método que no estaba presente. Para llevarlas a cabo se ha utilizado como librería de pruebas JUnit5.

Como el resultado de las correcciones se almacena en una carpeta temporal, la primera aproximación que se llevó a cabo fue la siguiente:

1. Utilizando los archivos de un ejercicio creado con solamente la propiedad a probar, se ejecutaba el corrector.
2. Utilizando JavaParser se parseaba el contenido del archivo resultado.
3. También mediante JavaParser se comprobaba que el código había sido corregido con éxito.

Sin embargo, el tutor Josep Silva sugirió que quizás esta aproximación era mejor evitarla siempre que fuera posible. Esto se debe a que, cuando la corrección se ha realizado mediante los mecanismos de JavaParser, es posible que, si la propia librería contiene errores, los casos de prueba que utilicen las mismas funciones darían resultados erróneos.

Finalmente se ideó un método alternativo. Se utilizarían los mismos ejercicios, pero se cambiaría la forma de validación de la corrección. En lugar de hacer *parsing* del resultado, se añade un fichero Java en el mismo directorio que, en función de cuál sea el objetivo de la prueba, haga que el resultado continúe compilando si es correcto, o deje de hacerlo si no es correcto.

Para llevar a cabo estas pruebas se han desarrollado varias clases. En primer lugar, tenemos a la clase `TestUtils`, que se trata de, como su nombre indica, una simple clase utilitaria para escribir archivos y compilar.

```java
public class TestUtils {

    final static String correctedFilePath = buildCorrectedFilePathU();

    public static boolean compileSolvedFiles() {
        return JavaCompiler.compileProject(new File(correctedFilePath));
    }

    public static void writeToFile(File file, String text) throws IOException {
        PrintWriter printWriter = new PrintWriter(file);
        printWriter.println(text);
        printWriter.flush();
        printWriter.close();
    }

    private static String buildCorrectedFilePathU() {
        String s = File.separator;
        String asysPath = new File("").getAbsolutePath();
        asysPath = asysPath.replace("Codeassess", "");
        return asysPath + s + "ASysClient" + s + "tmp" + s + "solved";
    }
}
```

Figura 43 Clase TestUtils

Junto con ella, dado que el mecanismo para cada prueba siempre es el mismo: corregir, escribir el método de prueba en un archivo junto con el resultado y compilar. Se ha desarrollado una clase abstracta `SolveTest` que contiene todos los pasos necesarios para realizar la corrección del ejercicio que le indiquemos.

```java
public abstract class SolveTest {
    static final Logic logic = new Logic();
    static Properties properties;
    static int assessmentIndex;

    static void setUpTest(String testName) {
        File exerciseFile = new File("test_files/" + testName);
        final File projectFile = new File("test_files/" + testName + "/Projects/Handovers/unassessed");
        final String projectName = projectFile.getName();
        assessmentIndex = logic.newAssessment(exerciseFile, exerciseFile.getName());
        logic.prepareProject(assessmentIndex, projectName);
        properties = logic.getCompleteProperties(assessmentIndex);
        logic.assessProject(assessmentIndex);
    }
}
```

Figura 44 Clase abstracta SolveTest

En ella podemos ver el método "`setUpTest()`" que contiene todo el código encargado de iniciar la corrección del ejercicio que le indiquemos, básicamente ahorrándonos escribir todo lo que iría en los métodos "`@BeforeAll`" de las pruebas.

Además de `SolveTest`, se han creado dos clases abstractas más que heredan de ella: `CompileTest` y `NotCompileTest`. Cada prueba heredará de la que necesite en función de si la propiedad se comprueba haciendo que el ejercicio siga compilando o deje de compilar.

```java
public abstract class CompileTest extends SolveTest {

    @Test
    void propertyTest() {
        assertTrue(TestUtils.compileSolvedFiles());
        File testClassFile = new File(TestUtils.correctedFilePath + File.separator + "TestClass.java");
        try {
            TestUtils.writeToFile(testClassFile, getTestClass());
        } catch (IOException e) {
            fail("No se pudo crear el archivo con los tests");
        }
        assertTrue(TestUtils.compileSolvedFiles());
    }

    abstract String getTestClass();
}
```

Figura 45 Clase CompileTest.java

Como se muestra en la anterior figura, `CompileTest` es una clase abstracta que contiene el método propiamente anotado con "`@Test`", que realiza el proceso de escritura de la clase escrita en "`getTestClass()`" (la cual contiene la prueba real) para así añadirla al directorio de la solución. La idea de la prueba es asegurarnos de que la solución del corrector compila, añadir nuestro nuevo fichero y probar que, puesto que se ha corregido correctamente, continúa compilando.

Para cada prueba unitaria que herede de `CompileTest` deberemos hacer dos cosas:

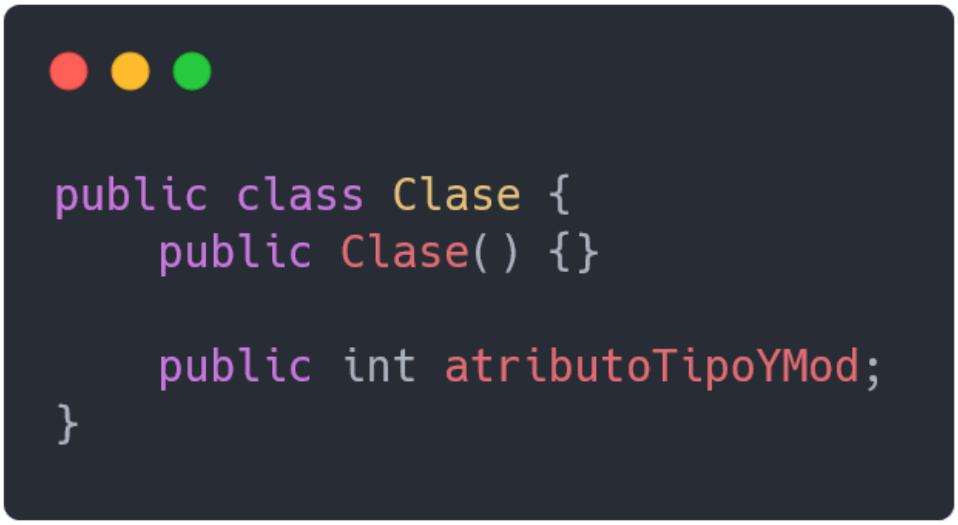
1. Llamar al método "`setUpTest()`" dentro de una función anotada con "`@BeforeAll`", pasándole el nombre que le hemos puesto al ejercicio a corregir.
2. Escribir la clase que hará las pruebas en el método "`getTestClass()`".

Para visualizar mejor este proceso de pruebas pongamos un ejemplo: probar que se añade a la clase un atributo faltante para el cual se ha definido que se corrija el nombre, el tipo y los modificadores. En primer lugar, necesitaremos la clase correcta:

```
public class Clase {
    public Clase() {}

    public int atributoTipoYMod;
    public float atributoTipoModNombre;
}
```

Figura 46 Código correcto de la prueba unitario de atributos no presentes

También hará falta la clase que deberá ser corregida, en este caso, una clase sin el atributo "`atributoTipoModNombre`":



```
public class Clase {
    public Clase() {}

    public int atributoTipoYMod;
}
```

Figura 47 Código incorrecto de la prueba unitaria de atributos no presentes

Una vez tenemos el ejercicio preparado, la implementación de la clase de prueba se ve de la siguiente forma:

```
class ClassToInterfaceTest extends SolveTest {

    @BeforeAll
    static void setUp() {
        SolveTest.setUpTest("missingAttributeTest");
    }

    @Override
    String getTestClass() {
        return
            "public class TestClass {\n" +
            "    public void missingAttributeTest() {\n" +
            "        Clase clase = new Clase();\n" +
            "        clase.atributoTipoModNombre = 0.0f;\n" +
            "    }\n" +
            "}\n";
    }

    @Override
    boolean shouldCompile() {
        return true;
    }
}
```

Figura 48 Clase de prueba de atributos faltantes

Por tanto, si la corrección se ha llevado a cabo satisfactoriamente, al añadir la clase `TestClass` a un archivo Java en el mismo directorio que el resultado e intentar compilar, debería hacerlo sin ningún problema. El directorio donde se encuentra el resultado se vería de la siguiente forma mientras durante la realización de la prueba:

Figura 49 Directorio del resultado mientras se realizan las pruebas

Otros ejemplos de pruebas serían, por ejemplo, uno para comprobar que una interfaz se convierte satisfactoriamente en una clase. Para ello, los códigos correcto e incorrecto son los siguientes:

```
public interface Clase {
    public void method1();
}
```

Figura 50 Código a corregir en la prueba de transformar una interfaz en una clase

```java
public class Clase {
     private int atributo1;
     private float atributo2;

     public Clase(int at1, float at2) {
        this.atributo1 = at1;
        this.atributo2 = at2;
     }
     public void method1() {
        float suma = atributo1 + atributo2;
     }
}
```

Figura 51 Código correcto en la prueba de transformar una interfaz en una clase

Y, por último, el código de la clase de prueba; el cual, si Clase siguiera siendo una interfaz, no compilaría, puesto que una interfaz no puede ser instanciada:

```java
class ClassToInterfaceTest extends CompileTest {

    @BeforeAll
    static void setUp() {
        SolveTest.setUpTest("classToInterfaceTest");
    }

    @Override
    String getTestClass() {
        return
            "public class TestClass {\n" +
            "    public void classToInterfaceTest() {\n" +
            "        Clase clase = new Clase(1, 1.2f);\n" +
            "    }\n" +
            "}\n";
    }
}
```

Figura 52 Clase de prueba de transformación de interfaz a clase

El caso de las pruebas que heredan de la clase `NotCompileTest` es diferente, ya que las clases que contienen están diseñadas de forma que compilen correctamente si y solo si se ha llevado a cabo la corrección de forma incorrecta.

```java
public abstract class NotCompileTest extends SolveTest {
    @Test
    void propertyTest() {
        List<String> classes = getTestClasses();
        for(String testClass: classes) {
            assertTrue(TestUtils.compileSolvedFiles());
            File testClassFile = new File(TestUtils.correctedFilePath + File.separator + "TestClass.java");
            try {
                TestUtils.writeToFile(testClassFile, testClass);
            } catch (IOException e) {
                fail("No se pudo crear el archivo con los tests");
            }
            assertFalse(TestUtils.compileSolvedFiles());
            boolean deleted = testClassFile.delete();
            if (!deleted) fail("No se pudo eliminar el archivo previo");
        }
    }

    abstract List<String> getTestClasses();
}
```

Figura 53 Clase NotCompileTest.java

La mayor diferencia entre `CompileTest` y `NotCompileTest`, además de sus comprobaciones, es que el método "`getTestClasses()`" de `NotCompileTest` devuelve una lista de clases, en lugar de una sola, aunque esto solamente tomará relevancia en el apartado de pruebas de integración.

Para comprobar su funcionamiento, utilizaremos la clase que representaba el código correcto de la transformación de interfaz a clase, pero esta vez asumiremos que el método "`method1()`" es innecesario y, por tanto, debe ser eliminado.

Esta propiedad no puede ser comprobada de la misma forma que las anteriores, ya que no hay manera de validar que un método no existe sin que haya un error de compilación o, utilizar reflexión. Por tanto, la clase de pruebas quedaría de la siguiente forma:

```java
public class DeletedMethod extends NotCompileTest {

    @BeforeAll
    static void setUp() {
        SolveTest.setUpTest("integrationTest");
    }

    @Override
    List<String> getTestClasses() {
        List<String> testClasses = new ArrayList<>();
        testClasses.add(
            "public class TestClass {\n" +
                "public void method() {" +
                    "Clase clase = new Clase(42, 0.0f);" +
                    "clase.method1();" +
                "}" +
            "}\n"
        );
        return testClasses;
    }
}
```

Figura 54 Clase de prueba DeleteMethod.java

Si se ha corregido correctamente, la llamada a “method1” hará que, al añadir este archivo, el directorio deje de compilar, asegurándonos de que se ha corregido correctamente.

7.2 Pruebas de integración

Para las pruebas de integración, en lugar de corregir y comprobar una sola propiedad en cada prueba, se han construido ejercicios con muchas más propiedades, las cuales se validarán al final de la corrección al completo, permitiendo probar el funcionamiento del sistema cuando se modifica el código repetidas veces de diferentes formas.

Vamos a ver las pruebas que se han desarrollado para el caso de prueba realizado con un ejercicio compuesto por múltiples clases, correspondientes a una de las prácticas de la asignatura LTP y muy similar al ejemplo Figura, que se ha utilizado a lo largo de este documento. Las clases completas, tanto correctas como incorrectas, así como su versión corregida, se añadirán en forma de anexo.

Para poder comprobar todas las propiedades que se necesitaban se han tenido que construir tres clases de prueba:

1. **`CompileIntegrationTest`.** Para las propiedades que se pueden probar con clases con las que el ejercicio enviado no habría compilado, pero su corrección sí.
2. **`NotCompileIntegrationTest`.** Para las propiedades que se pueden probar con una clase con la que el ejercicio enviado sí habría compilado, pero su versión corregida no.
3. **`ReflectionIntegrationTest`.** Para aquellas propiedades que no pueden ser comprobadas sin utilizar reflexión. Resultan ser el caso menos deseado, por ello se intenta minimizar su uso.

7.2.1 Clase CompileIntegrationTest

El método "`getTestClass()`" de la clase `CompileIntegrationTest` se ve de la siguiente forma:

```
@Override
  String getTestClass() {
      return
          "public class TestClass {\n" +
              "public void method() {"+
                  //Correccion propiedad herencia Circle extends Figure
                  "Figure circle = new Circle(0.0, 0.0, 0.0);" +
                  //Correccion metodos no presentes
                  "Circle circle2 = new Circle(0.0, 0.0, 0.0);" +
                  "double circleArea = circle2.area();" +
                  "double circlePerimeter = circle2.perimeter();" +
                  //Correccion parametro de equals a tipo Objeto
                  "FiguresGroup fg = new FiguresGroup();" +
                  "fg.equals(new Object());" +
              "}" +
          "}\n";
  }
```

Figura 55 getTestClass de la clase CompileIntegrationTest

Ahora se puede apreciar de manera más clara que al añadir esta clase, si no se hubiera corregido la propiedad de que `Figure` debe heredar de `Circle`, o no se hubieran añadido los métodos "`area()`" o "`perimeter()`" o no se hubiera cambiado el tipo del parámetro de entrada del método "`equals`" de la clase `FiguresGroup`, al añadir esta clase, el código no hubiera compilado y la prueba habría fallado. Solamente con que fallara una de estas correcciones, el código habría dejado de compilar, por lo que basta con ponerlas todas en la misma clase para asegurarnos de que todas se han corregido satisfactoriamente.

7.2.2 Clase NotCompileIntegrationTest

Sin embargo, echemos un ojo al mismo método, pero de la clase `NotCompileIntegrationTest`:

```java
public class NotCompileIntegrationTest extends NotCompileTest {

    @BeforeAll
    static void setUp() {
        SolveTest.setUpTest("integrationTest");
    }

    @Override
    List<String> getTestClasses() {
        List<String> testClasses = new ArrayList<>();
        testClasses.add(
            "public class TestClass {\n" +
                //Correccion ahora figure es abstracto
                "public void method() {" +
                    "Figure figure = new Figure(0.0, 0.0);" +
                "}" +
            "}\n"
        );
        testClasses.add(
                "public class TestClass {\n" +
                    //Correccion figure no tiene el metodo add(int)
                    "public void method() {" +
                        "Figure figure = new Triangle(0.0, 0.0, 0.0, 0.0);" +
                        "figure.area(42);" +
                    "}" +
                "}\n"
        );
        testClasses.add(
                "public class TestClass {\n" +
                    //Correccion figure no tiene el metodo perimeter(int, double)
                    "public void method() {" +
                        "Figure figure = new Triangle(0.0, 0.0, 0.0, 0.0);" +
                        "figure.perimeter(42, 0.0);" +
                    "}" +
                "}\n"
        );
        testClasses.add(
                "public class TestClass {\n" +
                    //Correccion se ha eliminado el constructor incorrecto Rectangle(double, double, double)
                    "public void method() {" +
                        "Rectangle rectangle = new Rectangle(0.0, 0.0, 0.0);" +
                    "}" +
                "}\n"
        );
        return testClasses;
    }
}
```

Figura 56 Clase NotCompileIntegrationTest.java

Ahora cobra sentido el que, al contrario que en `CompileTest`, se utilice una lista de clases en lugar de una sola. Si nos queremos asegurar de que todas las propiedades se han corregido correctamente, todas deben de generar errores de compilación.

Si utilizáramos una sola clase, esto causaría que desde que una sola propiedad se corrigiera bien, el código ya dejaría de compilar y la prueba daría como resultado que la corrección es correcta, cuando no es el caso. El utilizar una lista de clases nos asegura que todas y cada una de las propiedades generen un error de compilación.

Por ejemplo, pongamos que en lugar de utilizar clases separadas como se ve en la figura anterior, utilizáramos solamente una, al igual que se hace en `CompileTest`, generando una clase de la siguiente forma:

```
public class TestClass {
    public void method() {
        //Correccion ahora figure es abstracto
        Figure figure = new Figure(0.0, 0.0);
        //Correccion se ha eliminado el constructor
        //incorrecto Rectangle(double, double, double)
        Rectangle rectangle = new Rectangle(0.0, 0.0, 0.0);
    }
}
```

Figura 57 Versión incorrecta de TestClass para NotCompileIntegrationTest

Nuestro criterio de éxito es que, una vez este archivo se añada al directorio de la corrección, este deje de compilar. En esta versión de `TestClass`, se comprueba que ahora `Figure` es abstracto (por lo que no puede ser instanciado), y que hemos eliminado un constructor incorrecto de `Rectangle`. Es absolutamente necesario que, si la prueba da correcta, signifique que todas las propiedades se han corregido correctamente.

Ahora pongamos que se ha corregido el modificador `abstract` en la clase `Figure`, pero el constructor de `Rectangle` no ha podido ser eliminado. En este caso, al añadir la clase `TestClass`, el código no compilará porque `Figure` no puede ser instanciado (se cumple el criterio de éxito), por lo que la prueba dará correcta pese a que no se han corregido todas las propiedades satisfactoriamente. Es por este motivo que se utiliza una lista con clases individuales para cada propiedad, y que todas deben provocar un error de compilación.

7.2.3 Clase ReflectionIntegrationTest

Como ya se ha establecido, la reflexión es el método menos deseado para realizar estas pruebas. No obstante, existen propiedades que solamente se pueden comprobar utilizándola, como la corrección de determinados modificadores o los ejemplos que se van a mostrar a continuación:

```java
public class ReflectionIntegrationTest extends SolveTest {
    @BeforeAll
    static void setUp() {
        SolveTest.setUpTest("integrationTest");
    }

    @Test
    public void circleSuperClass() {
        File circleFile = new File(TestUtils.correctedFilePath + File.separator + "Circle.java");
        CompilationUnit circle = null;
        try {
            circle = JavaParser.parse(circleFile);
        } catch (FileNotFoundException e) {
            fail("Class circle does not exist");
        }
        List<FieldDeclaration> circleFields = circle.getPrimaryType().get().getMembers().stream()
                .filter(BodyDeclaration::isFieldDeclaration)
                .map(BodyDeclaration::asFieldDeclaration)
                .collect(Collectors.toList());

        if (circleFields.size() > 1) {
            fail("No se han eliminado las variables de la superclase");
        }

    }

    @Test
    public void triangleAttributeNameChange() {
        File triangleFile = new File(TestUtils.correctedFilePath + File.separator + "Triangle.java");
        CompilationUnit triangle = null;
        try {
            triangle = JavaParser.parse(triangleFile);
        } catch (FileNotFoundException e) {
            fail("Class circle does not exist");
        }
        List<VariableDeclarator> variables = triangle.getPrimaryType().get().getMembers().stream()
                .filter(BodyDeclaration::isFieldDeclaration)
                .map(BodyDeclaration::asFieldDeclaration)
                .flatMap(fieldDeclaration -> fieldDeclaration.getVariables().stream())
                .collect(Collectors.toList());

        assertFalse(variables.stream().anyMatch(variable -> variable.getNameAsString().equals("altura")));
        assertTrue(variables.stream().anyMatch(variable -> variable.getNameAsString().equals("height")));

    }
}
```

Figura 58 Clase ReflectionIntegrationTest.java

En el primer método se comprueba que los atributos de la clase `Circle` heredados de su superclase se hayan eliminado al añadir la herencia. Ya que podríamos acceder a ellos puesto que los hereda de `Figure`, ni la aproximación de `CompileTest` ni la de `NotCompileTest` serían adecuadas.

En el segundo método se comprueba que se ha sustituido el nombre de un atributo privado por su nombre correcto, este caso pasando de "`altura`" a "`height`". Como se trata de un atributo de visibilidad privada, este no puede ser accedido desde el exterior, por lo que intentar hacerlo desde una clase de prueba externa no es posible.

7.3 Pruebas de aceptación

El objetivo de las pruebas de aceptación es comprobar junto con los usuarios finales del sistema que este se comporta correctamente y de la manera esperada, además de cumplir con las expectativas de rendimiento de estos.

En total se han realizado cinco sesiones junto al tutor y dos sesiones con usuarios externos, las cuales dieron lugar a importantes correcciones y, sobre todo, evoluciones en el enfoque del desarrollo.

7.3.1 Primera sesión con el tutor

En la primera sesión de pruebas, se probaron ejemplos sencillos, en los cuales se corregían ejercicios con un solo archivo y pocos elementos.

En esta sesión se detectó un problema que definiría en gran parte el enfoque del desarrollo a futuro de este proyecto: existían elementos en el código que no solo eran irresolubles, sino que provocaban errores de compilación tras aplicar las correcciones, provocando que siempre que se diera alguno de estos casos se devolviera una solución errónea.

El problema detectado fue que al inicializar un atributo en su declaración con un tipo incompatible con el de la solución del profesor, pese a que este se corregía perfectamente, resultaba en un error de compilación, como se puede ver en la siguiente figura:

Figura 59 Error encontrado en la primera sesión de pruebas de aceptación

7.3.1.1 Resultados

El resultado de esta sesión fue, por tanto, que se debía implementar un algoritmo que además de tener en cuenta la posibilidad de generar errores de compilación, fuera no solo capaz de revertir

las propiedades que las provocaran en la mayoría de los casos, sino que además lo hiciera con el mínimo número de compilaciones y el menor número de correcciones deshechas posible.

Por ello, no solo se diseñó e implementó el algoritmo especificado en el apartado de desarrollo, sino que se llegó a la conclusión de que era necesario ordenar las propiedades de forma que aquellas que generen el mayor número de compilaciones se corrigieran las últimas, llegando finalmente al orden ya mostrado previamente.

7.3.2 Segunda sesión con el tutor

En la segunda sesión de pruebas, en lugar de utilizar de nuevo ejemplos sencillos se optó por utilizar ejercicios de múltiples clases, entre las que se incluía una práctica de la asignatura LTP.

Al corregir los ejercicios y explorar los resultados parciales del proceso de corrección se pudieron observar errores menores en el corrector, como falta de comentarios en algunas correcciones, pero principalmente que había propiedades para las cuales el módulo de corrección había actuado conforme a la información provista por el detector, pero el resultado era erróneo.

7.3.2.1 Resultados

Como resultado de esta sesión se solucionaron los comentarios faltantes, se añadió información descriptiva a estos para que el alumno pudiera entender qué había arreglado específicamente el sistema, y más importante, se añadió al detector de errores un método que evaluaba el mejor candidato para cada propiedad, como, por ejemplo, cuál era el atributo que era más probable que tuviera el tipo erróneo, aumentando enormemente su fiabilidad y mejorando los resultados.

7.3.3 Tercera sesión con el tutor

La tercera sesión repitió los casos de prueba generados para la segunda, pero esta vez con resultados mucho mejores gracias a los detalles previamente arreglados y, sobre todo, a la mejora que el método de análisis de candidatos supuso para el detector.

7.3.3.1 Resultados

En esta sesión más que errores, se llegaron a ver casos no deseados para la seguridad de las correcciones, como la sustitución de nombres de métodos. Esta corrección es tan problemática puesto que puede afectar a funciones adicionales definidas por el alumno de forma correcta, y que el sistema interpreta como elementos incorrectos, ya que no se encuentran en la solución del profesor.

Se trata de problemas inherentes de la forma de corrección actual que se salen del alcance de este proyecto y deben atajarse en futuros trabajos en todo el sistema de corrección.

7.3.4 Cuarta sesión con el tutor

En esta sesión se realizaron correcciones en dos ejercicios compuestos de múltiples clases, añadiendo errores variados para ver el comportamiento del sistema. Tras esto, se llegaron a encontrar algunos desperfectos en algunas propiedades.

7.3.4.1 Resultados

Los errores localizados en esta sesión fueron, en primer lugar, que, si un método tenía un nombre incorrecto, y a su vez, un atributo tenía el mismo nombre, al intentar sustituir todas las apariciones de las llamadas a este, se sustituía también el nombre del atributo. Para comprender mejor el problema, vamos a ver el caso en el que se daba mediante el siguiente código:

Figura 60 Versión incorrecta del método "descapotar"

En este caso, al atributo "`descapotado`" no se le corrige el nombre para así dar libertad al alumno, por lo que en cualquier caso es correcto. Sin embargo, al método sí que se exige que se llame "`descapotar()`", por lo que habrá que sustituirlo. En la versión del sistema probada en esta sesión, la corrección generada se veía así:

```java
// ASys: Sustituido descapotado por descapotar
private boolean descapotar;

// ASys: Sustituido descapotado por descapotar
private void descapotar() {
  // ASys: Sustituido descapotado por descapotar
    descapotar = !descapotar;
}
```

Figura 61 Corrección errónea del método "descapotar"

Tras corregir este error en el código del sistema, la salida para este mismo caso pasó a ser la esperada, quedando el código del alumno así:

```java
private boolean descapotado;

// ASys: Sustituido descapotado por descapotar
private void descapotar() {
    descapotado = !descapotado;
}
```

Figura 62 Corrección correcta de "descapotar"

El segundo error detectado fue que, cuando un atributo no era de tipo primitivo (`int, long, double, float…`), sino que se trataba de un objeto (`String, Object, Figura, Rectángulo…`), este no era añadido en la solución al corregir. Tras esta sesión, todos los atributos faltantes en la solución del alumno son añadidos por el módulo corrector.

7.3.5 Quinta sesión con el tutor

En esta última sesión con el tutor se validó que se hubieran solucionado los errores presentes en la anterior, además de probar la introducción de nuevos tipos de errores en los ejercicios ya utilizados. En ella se consiguió refinar la información que proporciona ASys sobre las correcciones realizadas y se encontró un defecto en el detector de errores.

7.3.5.1 Resultados

Cuando ASys realiza una corrección, como se ha podido ver en ejemplos anteriores, se añade un comentario a la línea corregida, indicando qué ha cambiado. Cuando la línea no tenía comentarios previos, se añadía un comentario de línea con la doble barra, pero cuando ya existían comentarios previos (ya sea del alumno o de correcciones anteriores), se convertía en un comentario de bloque y se añadía al final la nueva información de ASys. Es necesario añadir el prefijo "ASys:" a todos los comentarios introducidos por el sistema para poder distinguirlos.

Había dos problemas: que no todos los comentarios tenían el prefijo y que los comentarios de línea eran de color diferente en el editor y resultaba fácil perdérselos o confundirlos. Por ello, se decidió que todos los comentarios añadidos por ASys deberían ser de bloque y con el prefijo.

El caso en el que se detectaron ambos errores fue el siguiente:

```
public abstract class Coche extends Vehiculo {

    //Añadido atributo faltante marca
    private String marca;

    private String modelo;

    protected boolean arrancado;

    /**
     * ASys: El metodo de tipo void arrancar no estaba presente
     * ASys: Se ha añadido el modificador public
     * ASys: Se ha añadido el modificador abstract
     */
    public abstract void arrancar();
}
```

Figura 63 Comentarios con formato poco apropiado en la corrección

Como podemos observar, al primer comentario le falta el prefijo "ASys:", y, además, en la interfaz de ASys los comentarios de línea son mucho menos visibles que los de bloque.

El defecto encontrado en el detector de errores tiene que ver con la identificación de métodos a sustituir cuando se ha definido que una función debe tener una declaración exacta. Es decir, que

no se corrijan los modificadores, la visibilidad, nombre y parámetros como elementos separados, sino que se sustituya al completo la cabecera de un método si no cumple todos los requisitos.

En el siguiente ejemplo, el método `toString` debe tener como cabecera "`public String toString()`". No obstante, el alumno por equivocación le ha añadido un parámetro de forma "`String a`". También, debería haber solamente un método `area()` abstracto y sin argumentos, pero el alumno ha puesto dos variantes, con uno y dos argumentos respectivamente. Como resultado, tenemos el siguiente código erróneo:

```java
public class Rectangulo extends Figura {
    private double base12;
    private double height;

    public Rectangulo(float x, double y, double b, double h) {
        super(x, y);
        base12 = b;
        height = h;
    }

    public String toString(String a) {
        String t = "Rectangulo";
        if (base12 == height) t = "Cuadrado";
        return t + ":\n\t" +
            super.toString() +
            "\n\tBase: " + base12 +
            "\n\tHeight: " + height;
    }

    public double area(int s) {
        return base12;// * height;
    }

    public double area(int x, int y) {
        return base12;// * height;
    }

}
```

Figura 64 Código de la clase Rectangulo erróneo

La corrección esperada sería que se sustituyera por completo la cabecera del método `toString` por la especificada anteriormente (aunque en este caso en específico se corresponde solamente con eliminar el parámetro) y que uno de los dos métodos `area` se corrija por la versión correcta del profesor. El resultado, al contrario de lo esperado, es el siguiente:

```java
public class Rectangulo extends Figura {
    //ASys: Sustituido base12 por base
    private double base;
    private double height;

    //ASys: Sustituido float por double
    public Rectangulo(double x, double y, double b, double h) {
        super(x, y);
        //ASys: Sustituido base12 por base
        base12 = b;
        height = h;
    }

    public String toString(String a) {
        String t = "Rectangulo";
        //ASys: Sustituido base12 por base
        if (base == height) t = "Cuadrado";
        return t + ":\n\t" +
            super.toString() +
            //ASys: Sustituido base12 por base
            "\n\tBase: " + base +
            "\n\tHeight: " + height;
    }

    //ASys: Esta es la declaración correcta del método toString
    public String toString() {
    }

    /**
    * ASys: Se han eliminado los siguientes elementos: x
    * ASys: Se han eliminado los siguientes elementos: y
    */
    public double area() {
        return base12;// * height;
    }

}
```

Figura 65 Corrección incorrecta debido a una detección de errores defectuosa

El detector de errores elige a un candidato para la sustitución que no es el ideal, puesto que no evalúa las propiedades de ambos métodos a la hora de elegir cuál debería ser sustituido. Por tanto, de forma contraintuitiva, escoge como candidato a la primera versión del método `area`.

La prueba de que no se analiza cuál es el mejor candidato para la sustitución en la detección de este error, sino que se elige al último posible, es que si intercambios el orden en el código del alumno entre el método `toString` y la primera versión de `area`, quedando de forma `area(int`

`s)`, `toString(String a)` y `area(int x, int y)`, la corrección sería la esperada: el primer `area` se queda igual, al `toString` se le corrige la declaración y el último `area` queda como la versión del profesor.

La solución de este problema se sale del alcance de este TFG, puesto que la raíz de este se encuentra en un módulo externo, quedando como trabajo futuro para el sistema.

7.3.6 Primera sesión con usuarios externos

En esta sesión se probaron los ejercicios que fueron utilizados para la cuarta y quinta sesión con el tutor. De cara a errores en las correcciones, se encontraron varias situaciones en las que el sistema no era capaz de corregir el ejercicio sin generar errores de compilación, pese a que a primera vista no debería haber problema.

También se indicaron situaciones en las cuales los mensajes informativos sobre las correcciones de ASys no tenían la claridad suficiente para el usuario. Por último, se generó de nuevo el defecto del detector de errores explicado en la quinta sesión con el tutor y se propusieron mejoras para futuro trabajo en la interfaz.

7.3.6.1 Resultados

De las mejoras propuestas en la sesión, se ha trabajado en solucionar aquellos casos en los que ASys debería ser capaz de corregir el ejercicio, eliminando pequeños errores presentes en el sistema. Por otra parte, se han incluido mejoras en los comentarios añadidos por el corrector con el fin de aumentar la claridad de la información.

Las mejoras propuestas en el apartado de la interfaz web se salen del alcance de este TFG y por tanto serán atajadas en futuros trabajos sobre ASys.

7.3.7 Segunda sesión con usuarios externos

El segundo usuario externo, con previa experiencia en ASys, sin embargo, destacó un gran número de defectos presentes en la interfaz de usuario, los cuales confirman que todavía queda mucho trabajo de desarrollo de cara a que el sistema esté listo para ser desplegado en producción.

7.3.7.1 Resultados

Pese a que se detectaron un gran número de errores y posibles mejoras en el apartado visual y funcional de la interfaz de usuario, esta no forma parte de lo desarrollado en este trabajo, salvo por una pequeña parte de integración con el mismo.

No obstante, toda la información obtenida en esta sesión será aplicada para futuras y necesarias mejoras en el apartado visual de ASys.

7.4 Pruebas de eficiencia

Uno de los requisitos de calidad presente en la ERS era que las peticiones de corrección debían resolverse en aproximadamente un segundo, y nunca superar los cinco segundos. Para validar este requisito, se ha desarrollado una prueba de eficiencia.

Esta consiste en medir el tiempo que tarda el sistema en realizar la corrección de cinco ejercicios diferentes, con un número variable de archivos y propiedades. Si alguna de ellas pasa de los cinco segundos, la prueba fallará. Como información adicional, se recogen los segundos máximos, mínimos y medios que se han tardado a lo largo de todas las correcciones, así como el tiempo acumulado.

La prueba de eficiencia se encuentra escrita de la siguiente forma:

```
@Test
    public void testEfficiency() {
        long accumulatedNano = 0;
        long minNano = Long.MAX_VALUE;
        long maxNano = 0;

        for (String file: files) {
            prepareAssessment(file);

            LocalDateTime currentTime = LocalDateTime.now();
            logic.assessProject(assessmentIndex);
            LocalDateTime finishedTime = LocalDateTime.now();

            long correctionNano = Duration.between(currentTime, finishedTime).getNano();
            if (correctionNano > MAX_ACCEPTABLE_NANO) {
                fail("Se ha tardado demasiado en corregir el ejercicio " + file);
            }
            if (correctionNano < minNano) {
                minNano = correctionNano;
            }
            if (correctionNano > maxNano) {
                maxNano = correctionNano;
            }
            accumulatedNano += correctionNano;
        }

        double averageSeconds = ((double) accumulatedNano / 1000000000.0)/ (double) files.size();
        double minSeconds = minNano / 1000000000.0;
        double maxSeconds = maxNano / 1000000000.0;
        double accumulatedSeconds = accumulatedNano / 1000000000.0;
        System.out.println("Los resultados de eficiencia estan entre los margenes aceptables:\n" +
                "Segundos minimos: " + minSeconds + "\n" +
                "Segundos de media: " + averageSeconds + "\n" +
                "Segundos maximos: " + maxSeconds + "\n" +
                "Tiempo total: " + accumulatedSeconds
        );
    }
```

Figura 66 Test de eficiencia

En ella, podemos observar cómo se itera sobre los archivos que contienen los ejercicios a corregir, se prepara la corrección y, posteriormente, se mide el tiempo que tarda el módulo corrector en ejecutar el código desarrollado en este trabajo.

Por último, se procesan los valores generados en nanosegundos para mostrarlos como información adicional en segundos, ya que resultan más intuitivos de esta manera. Los resultados que se han obtenido son los siguientes:

```
Los resultados de eficiencia estan entre los margenes aceptables:
Segundos minimos: 0.046
Segundos de media: 0.29475
Segundos maximos: 0.694
Tiempo total: 1.179
```

Figura 67 Resultados de las pruebas de eficiencia

Conclusiones

En este trabajo se ha abordado el desarrollo de una herramienta de corrección de código que podemos afirmar que resulta extremadamente novedosa y útil; y que abrirá las puertas a futuros alumnos, tanto de Ingeniería Informática como de otras carreras en las que se aprenda programación Java, a aprender de manera más sencilla, directa, rápida y personalizada.

El proyecto tendrá un impacto real puesto que va a ser implantado en diversas asignaturas de la Universitat Politècnica de València y se prevé que alrededor de 500 alumnos anualmente la usarán para autocorregir sus prácticas de programación.

El trabajo de implementación y diseño de este módulo no solamente ha ampliado enormemente mi conocimiento en lo que respecta a desarrollo Java en específico, mejorando mis habilidades en los campos de meta programación, reflexión, y sus funciones de programación funcional, sino toda mi perspectiva respecto a lenguajes de programación, su estructura y funcionamiento.

Por supuesto, el desarrollo de una herramienta correctora de código de estas características también ha supuesto una enorme mejora en mis habilidades de depuración y búsqueda de errores, además de una mayor capacidad de resolución de problemas y desarrollo de algoritmos.

No cabe duda de que cuando ASys se encuentre en un estado óptimo para ser lanzado a producción, supondrá un antes y un después en la enseñanza de programación en nuestra escuela, además de una herramienta educativa de referencia en su sector por sus innovadoras capacidades.

Sin duda, he de agradecer el trabajo de apoyo, orientación y guía de mi tutor Josep Silva, además de su confianza en mí para realizar un trabajo en una de las partes más fundamentales e innovadoras de ASys.

En lo que concierne a la consecución de objetivos, en este proyecto se pretendía desarrollar un sistema de corrección estructural automática de código Java, en la que además se manejaran correctamente los errores que podían surgir de la misma. Podemos afirmar que, tras haber completado su desarrollo, ASys dispone de un sistema de corrección que cumple con la especificación definida, el cual es capaz tanto de corregir correctamente la enorme mayoría de propiedades estructurales de los programas Java, como de manejar y adaptarse a los errores que puedan surgir en el proceso, devolviendo al alumno en cualquier caso un código correcto.

8.1 Trabajo futuro

Después de haber completado el desarrollo de este trabajo, ASys dispone de un sistema de corrección de código funcional y robusto, capaz de detectar y corregir de forma satisfactoria prácticamente todas las propiedades estructurales a evaluar dentro del contexto y tipo de ejercicios que se plantean en el sistema.

A pesar de ello, todavía hay mucho trabajo que puede llevarse a cabo en el futuro para ampliar la funcionalidad de corrección y mejorar lo ya implementado, con el fin de conseguir un módulo de corrección aún más confiable, flexible e inteligente, que haga lo mejor posible tanto para el alumno como para el profesor.

La siguiente prioridad, la cual se desarrollará poco después de la terminación de este trabajo, es la implementación del módulo de pruebas, el cual eliminará la limitación de no contar con corrección de la lógica de los programas y, por tanto, de los cuerpos de los métodos y funciones.

También será importante realizar varias mejoras en el detector de errores, ya que se trata de la piedra angular del sistema y del factor limitante de cara al crecimiento del módulo de corrección. En primer lugar, se deben tener en cuenta correcciones que pueden resultar demasiado peligrosas de cara a generar errores de compilación y que en la actualidad se llevan a cabo sin forma de ser configuradas, como puede ser sustituir el nombre de un método.

Correcciones como esta no tienen en cuenta que el alumno debería tener la libertad de resolver problemas a su propia manera, siempre y cuando el código sea de calidad. Definir métodos auxiliares es una buena forma de distribuir el trabajo y encapsular lógica, y si el ejercicio requiere que solamente se utilice un número determinado, existe la opción de especificarlo como propiedad extra. Sin embargo, esta sustitución de nombres o parámetros de métodos se lleva a cabo en cualquier caso en el que un método no se encuentre en el código del alumno, sin forma de ser configurada.

Una solución posible, además de añadir esta corrección como una propiedad, sería clasificar aquellas correcciones que puedan generar errores e implementar un valor booleano en el sistema que omita estas, lo que permitiría realizar solo correcciones seguras y correctas a cambio de sacrificar completitud.

Otra gran mejora para el detector sería ampliar la cantidad de información que ofrece como salida. En muchos casos la corrección se complica excesivamente para poder saber qué se debe hacer y, sobre todo, dónde debe hacerse en el código del alumno.

Esto ocurre, sobre todo, en los casos en los que las propiedades devuelven un valor nulo en el código del alumno, es decir, en los que ha de añadirse un elemento a este. En la actualidad, si por ejemplo se recibe una propiedad de tipo `SimpleName` con el valor del alumno a nulo, esto podría indicar muchos casos: que se debe añadir un atributo, que se debe añadir un método, una herencia, un parámetro, etc. Una gran cantidad de opciones que obligan al corrector a analizar los nodos superiores del código del profesor para saber qué debe de añadir, pero ese no es el peor caso.

El peor caso es el que se ha mencionado en el apartado de desarrollo cuando se habla de los parámetros de métodos y constructores. En el caso de los constructores, se debe de hacer una búsqueda que, aunque a veces efectiva, en muchos casos resulta ser como mínimo, imperfecta, para poder encontrar dónde colocar ese parámetro que falta. Un atributo indicando el constructor o método más probable, en el que además se coloque otro tipo de información útil dependiendo del tipo de corrección, podría simplificar muchísimo el código del corrector y proporcionar resultados más fiables y consistentes en un mayor número de ejercicios, pudiendo corregir más variedad de errores posibles.

Por último, otro apartado en el que se debe trabajar en el detector es la categorización de errores. Un ejemplo claro ocurre cuando se define una propiedad de herencia sin especificar que la declaración debe ser exacta (es decir, que incluya no solo sus `extends` sino todos los `implements`). Cuando se da la situación en la que el alumno no implementa la herencia, esperaríamos que se generara una propiedad de tipo `ExtendedTypes`, con el nodo solución siendo la clase de la que debe heredar y el del alumno a nulo. Sin embargo, la propiedad que se genera es de tipo `SimpleName`, obligando al corrector a analizar a qué parte del código pertenece en el código solución para ver si puede encontrarse y añadirse en el del alumno, una tarea que resultaría más sencilla o bien aplicando la propuesta anterior de añadir más información, o ajustando de mejor manera el tipo de cada propiedad.

De cara al corrector específicamente, uno de los desarrollos más importantes de cara a futuras mejoras es dar más opciones a la hora de llevar o no a cabo las correcciones más peligrosas, como puede ser detectar y sustituir automáticamente y el tipo incorrecto al añadir un tipo paramétrico.

A su vez, se deben llevar a cabo una mayor cantidad de pruebas enfocadas única y exclusivamente en analizar los resultados obtenidos, tanto de calidad de la corrección como de número de compilaciones en lo que respecta a la ordenación de propiedades. La optimización aún mayor de esta ordenación puede no solamente incrementar de forma muy importante la calidad de los resultados (como ya vimos que mejoraron solo con añadirla), sino también jugar un papel clave en el rendimiento del sistema cuando se realicen tareas con ejercicios a mayor escala.

8.2 Agradecimientos

Para concluir, quería dedicar un apartado para agradecer a todas aquellas personas que me han apoyado durante estos últimos años y que han jugado un papel muy importante para que haya llegado aquí.

En primer lugar, a mis padres y a toda mi familia, que, pese a que estos años nos hemos podido reunir menos de lo deseado, siempre me han ayudado con todo lo que han podido y nunca han dejado de apoyarme. En segundo lugar, a mi novia, que siempre me ha empujado a seguir pese a tener que aguantar mis quejas de los estudios, estando ahí tanto en los mejores como en los peores momentos. Por último, a mis amigos, tanto aquellos que llevan ahí desde siempre, como los que he conocido durante este grado; por mantenerse a mi lado y, también, por los buenos momentos que hemos compartido durante estos años.

Bibliografía

Apache Friends. (14 de junio de 2022). *Apache Friends*. Recuperado el 14 de junio de 2022, de XAMPP: Acerca de: https://www.apachefriends.org/es/about.html

Apache Software Foundation. (19 de octubre de 2021). *The Apache Ant Project*. Recuperado el 14 de junio de 2022, de Welcome: https://ant.apache.org/

Apache Software Foundation. (11 de junio de 2022). *Apache Maven Project*. Recuperado el 14 de junio de 2022, de Welcome to Apache Maven: https://maven.apache.org/

Apache Software Foundation. (11 de junio de 2022). *Apache Tomcat*. Recuperado el 14 de junio de 2022, de Apache Tomcat: https://tomcat.apache.org/

Crusoveanu, L. (14 de junio de 2022). *Baeldung*. (Baeldung, Editor) Recuperado el 14 de junio de 2022, de Intro to Inversion of Control and Dependency Injection with Spring: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Danny van Bruggen, F. T. (17 de septiembre de 2021). *Página principal de JavaParser*. Recuperado el 9 de junio de 2022, de JavaParser: https://javaparser.org

Docker, Inc. (2 de junio de 2022). *Docker Docs*. Recuperado el 14 de junio de 2022, de Get Started with Docker Compose: https://docs.docker.com/compose/gettingstarted/

Docker, Inc. (17 de mayo de 2022). *Docker Docs*. Recuperado el 14 de junio de 2022, de Orientation and setup: https://docs.docker.com/get-started/

Facebook. (14 de julio de 2021). *React*. Recuperado el 14 de junio de 2022, de React: Empezando: https://es.reactjs.org/docs/getting-started.html

Facebook. (4 de abril de 2022). *React*. Recuperado el 14 de junio de 2022, de Presentando JSX: https://es.reactjs.org/docs/introducing-jsx.html

FasterXML. (26 de mayo de 2022). *Github*. Recuperado el 14 de junio de 2022, de Jackson: https://github.com/FasterXML/jackson

Google. (28 de febrero de 2022). *Angular Docs*. Recuperado el 14 de junio de 2022, de What is Angular: https://angular.io/guide/what-is-angular

Google. (7 de mayo de 2022). *Documentación de Google*. Recuperado el 14 de junio de 2022, de Proteger sitios con el protocolo HTTPS: https://developers.google.com/search/docs/advanced/security/https?hl=es

Google. (11 de febrero de 2022). *Github*. Recuperado el 14 de junio de 2022, de GSON User Guide: https://github.com/google/gson/blob/master/UserGuide.md

Google. (26 de mayo de 2022). *Google Cloud*. Recuperado el 14 de junio de 2022, de ¿Qué es una base de datos relacional?: https://cloud.google.com/learn/what-is-a-relational-database?hl=es-419

Google. (11 de febrero de 2022). *Kubernetes*. Recuperado el 14 de junio de 2022, de ¿Qué es Kubernetes?: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

Hibernate. (14 de junio de 2022). *Hibernate*. Recuperado el 14 de junio de 2022, de Hibernate ORM: https://hibernate.org/orm/

Lee, J. K. (2018). *A Study on Abstract Syntax Tree for Development of a JavaScript.* Seoul: International Journal of Grid and Distributed Computing. Recuperado el 16 de Junio de 2022, de http://article.nadiapub.com/IJGDC/vol11_no6/4.pdf

Microsoft. (30 de mayo de 2022). *TypeScript*. Recuperado el 14 de junio de 2022, de The TypeScript Handbook: https://www.typescriptlang.org/docs/handbook/intro.html

MinIO, Inc. (14 de junio de 2022). *MinIO Docs*. Recuperado el 14 de junio de 2022, de MinIO High Performance Object Storage: https://docs.min.io/minio/baremetal/

Moura, M. (13 de agosto de 2020). *Vue Material*. Recuperado el 14 de junio de 2022, de Getting Started: https://www.creative-tim.com/vuematerial/getting-started

Mozilla Foundation. (8 de diciembre de 2020). *MDN Web Docs*. Recuperado el 14 de junio de 2022, de Node.js: https://developer.mozilla.org/es/docs/Glossary/Node.js?utm_campaign=feed&utm_medium=rss&utm_source=developer.mozilla.org

Mozilla Foundation. (20 de abril de 2021). *MDN Web Docs*. Recuperado el 14 de junio de 2022, de MDN Web Docs: HTML: https://developer.mozilla.org/es/docs/Web/HTML

Mozilla Foundation. (7 de julio de 2021). *MDN Web Docs*. Recuperado el 14 de junio de 2022, de CSS: https://developer.mozilla.org/es/docs/Web/CSS

Mozilla Foundation. (30 de mayo de 2022). *MDN Web Docs*. Recuperado el 14 de junio de 2022, de JavaScript: https://developer.mozilla.org/es/docs/Web/JavaScript

Mozilla Foundation. (3 de marzo de 2022). *MDN Web Docs*. Recuperado el 14 de junio de 2022, de JSON: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON

npm, Inc. (24 de enero de 2022). *npm Docs*. Recuperado el 14 de junio de 2022, de About npm: https://docs.npmjs.com/about-npm

Oracle. (11 de enero de 2013). *The Java EE 6 Tutorial*. Recuperado el 14 de junio de 2022, de Introduction to the Java Persistence API: https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

Oracle. (14 de septiembre de 2021). *Java*. Recuperado el 14 de junio de 2022, de Getting Started with Java: https://dev.java/learn/getting-started-with-java/

Oracle. (13 de junio de 2022). *MySQL*. Recuperado el 14 de junio de 2022, de MySQL 8.0 Reference Manual: https://dev.mysql.com/doc/refman/8.0/en/

Simoes, A. (1 de enero de 2021). *Interaction Design Foundation*. (I. D. Foundation, Editor) Recuperado el 14 de junio de 2022, de Google’s Material Design - Android Design

Language: https://www.interaction-design.org/literature/article/google-s-material-design-android-design-language

Square. (14 de junio de 2022). *Github*. Recuperado el 14 de junio de 2022, de Moshi: https://github.com/square/moshi

Talend. (20 de agosto de 2021). *Restlet*. Recuperado el 14 de junio de 2022, de Overview: https://restlet.talend.com/documentation/tutorials/2.4/overview

VMware. (11 de mayo de 2022). *Spring Framework*. Recuperado el 16 de junio de 2022, de Documentación Spring Framework: https://spring.io/projects/spring-framework

Vue.js Team. (3 de junio de 2022). *Introducción a Vue*. Recuperado el 10 de junio de 2022, de Vue Guía de introducción: https://vuejs.org/guide/introduction.html#api-styles

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**	X			
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**				X
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Este TFG ha abordado el desarrollo de un módulo de corrección automática de código Java, el cual forma parte de un sistema mayor llamado ASys. Se trata de una plataforma educativa diseñada para mejorar la enseñanza de programación Java y Haskell, que proveerá a los alumnos con una gran cantidad de ejercicios a resolver y reducirá en gran parte el trabajo del equipo docente en materia de corrección.

El objetivo final es que ASys se integre completamente dentro de las herramientas docentes utilizadas por la ETSINF para la enseñanza en el grado de Ingeniería Informática. Una vez alcance una versión estable apta para producción, supondrá un salto de calidad enorme con respecto a las herramientas disponibles previamente.

Los profesores podrán enviar tareas e incluso preparar sesiones de prácticas directamente desde el sistema, y estas podrán no solamente ser evaluadas de forma automática con respecto al comportamiento del código (como ya se hace en algunos exámenes de prácticas actuales), sino que se podrán introducir ejercicios que valoren la estructura y forma de escribir el código.

Pero las mejoras no vendrán solamente por la posibilidad de tener una plataforma centralizada para preparar ejercicios, prácticas y tareas. Las ventajas de ASys incluyen la evaluación personalizada de las habilidades del alumno en cada una de las áreas evaluables en la programación orientada a objetos (herencia, polimorfismo, etc.). Para ello, ASys ahora dispone de la función implementada en este TFG: la corrección automática.

Con la corrección automática, los alumnos serán capaces de no solamente conocer sus errores mediante un mensaje, como ocurría hasta ahora, sino de ver su propio código modificado para cumplir con las propiedades evaluables que estableció el profesor, lo que resulta de una mucho mejor experiencia que ver directamente la solución. Gracias a esta nueva característica la evaluación no es sumativa, sino que se transforma en una evaluación formativa; en la que el alumno puede ver sus fallos y aprender cómo corregirlos.

Los profesores, por otra parte, serán liberados de la carga que supone corregir el código de cada alumno de uno en uno de forma manual, pudiendo en su lugar, utilizar su tiempo y recursos en desarrollar nuevos ejercicios y material docente que mejore notablemente la experiencia y aprendizaje del alumnado. Por tanto, el profesorado podrá dedicarse a enseñar, sin tener que preocuparse por corregir.

El desarrollo e implantación de herramientas innovadoras como ASys dentro de la enseñanza en la Universitat Politècnica de València, apoya la consecución del objetivo de desarrollo sostenible número 4: “Educación de calidad”, gracias a que se conseguirá aumentar el nivel de calidad de la docencia, pudiendo influir positivamente en el número de jóvenes y adultos que consiguen obtener los conocimientos necesarios y, como consecuencia, el título de graduado universitario, abriendo las puertas a un mayor número de personas al mercado laboral en sectores técnicos e industriales, lo que supondrá una gran mejora en la calidad de vida y empleo de una gran cantidad de personas.

En conclusión, la investigación y desarrollo de nuevas herramientas enfocadas a la mejora de la situación actual de alumnos y profesores, tendrá un gran impacto en el aumento de la calidad de la enseñanza, la cual tiene una relación directa con el número de graduados y profesionales cualificados que se pueden generar en cada universidad. Este trabajo aporta su granito de arena en la implantación de una de estas herramientas, enfocada en la mejora de la experiencia educativa presente en la carrera de Ingeniería Informática.

Anexo: Sesiones de pruebas con usuarios externos

Prueba Aceptación 1

INTERFAZ

* Las propiedades que aparecen en la interfaz de la izquierda que valen 0 puntos no deberían estar.
* Cuando el programa no compila sería bonito enseñar el error de compilación

FUNCIONALIDAD

CLASE Coche.java:

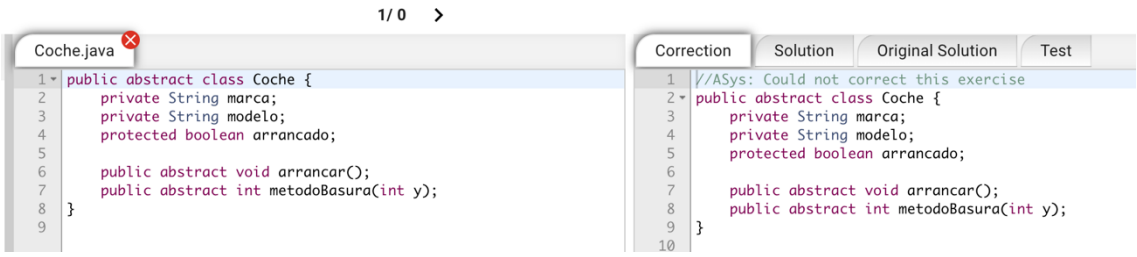
Consideraciones del escenario con respecto a la solución:

* Coche no hereda de *Vehiculo*
* Deportivo no hereda de coche ni tiene referencias a *super.arrancar()*

- Incluir un método con cuerpo vacío donde se esperaba un método abstracto genera que ASys no sepa qué hacer.

Error#1

- Añadir un segundo método abstracto con un parámetro que no debería estar en la implementación genera una situación que ASys no puede resolver.

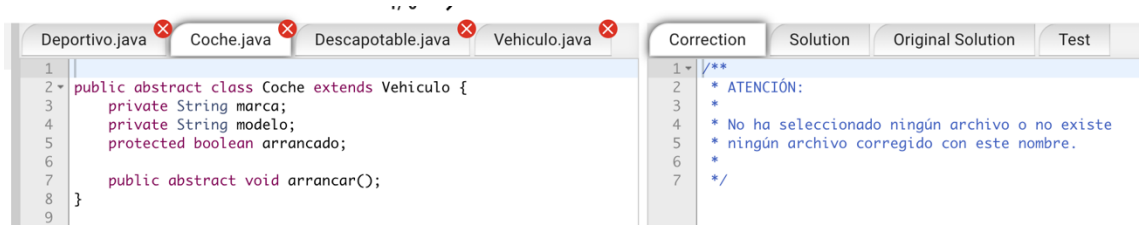


Error#2

- El mensaje de error de **Warning#3** no habla de la palabra reservada utilizada (cambia de *implements* a *extends* sin aviso)

Warning#3

- No se generan ficheros de corrección cuando todas las clases son iguales a la solución. El mensaje elegido entonces para decir que no hay errores es bastante regulero y mejorable, parece que haya fallado todo.



Warning#4

<u>CLASE Descapotable.java:</u>

Consideraciones del escenario con respecto a la solución:

* Añadido método *encapotar()*, implementado en Deportivo.java

- El mensaje de error de Warning#5 no hace referencia al método eliminado que le ha costado 1 punto de la nota al alumno.

Warning#5

<u>CLASE Vehiculo.java:</u>

Consideraciones del escenario con respecto a la solución:

* Ninguna

- El error asociado a la aridad de la función acelerar no se muestra en la corrección y descuenta 1 punto.

Vehiculo.java

```java

public abstract class Vehiculo {
    private double posX;
    private double posY;
    private double velocidad;

    public void acelerar(double v, int j) {
        this.velocidad += v;
    }

    public void frenar(double v) {
        this.velocidad -= v;
    }
}

```

Correction | Solution | Original Solution | Test

```
/**
 * ATENCIÓN:
 *
 * No ha seleccionado ningún archivo o no existe
 * ningún archivo corregido con este nombre.
 *
 */
```

Error#6

Prueba de Aceptación 2

Este caso estaba puesto directamente en la web, así que imagino que este error ya estará detectado:

CLASE Rectangulo.java:

He visto que al cambiar el nombre de un atributo en la corrección se propaga con un refactoring por todo el archivo incluyendo un mensaje en cada sitio implicado (imagino es la única corrección dentro de métodos que se aplica). En el ejemplo ha afectado al método toString al tener 2 tipos de error diferentes:

1) El refactoring
2) La cabecera incorrecta del método

Ambas correcciones deberían estar juntas y no por separado, ya que de hecho la segunda de ellas da un mensaje de error muy raro donde pone la cabecera correcta con una implementación incorrecta y un poco random.

Rectangulo.java

```java

public class Rectangulo extends Figura {
    private double base12;
    private double height;

    public Rectangulo(float x, double y, double b, double h) {
        super(x, y);
        base12 = b;
        height = h;
    }

    public String toString(String a) {
        String t = "Rectangulo";
        if (base12 == height) t = "Cuadrado";
        return t + ":\n\t" +
            super.toString() +
            "\n\tBase: " + base12 +
            "\n\tHeight: " + height;
    }

    public double area(int s) {
        return base12;// * height;
    }

    public double area(int x, int y) {
        return base12;// * height;
    }

}

```

Correction | Solution | Original Solution | Test

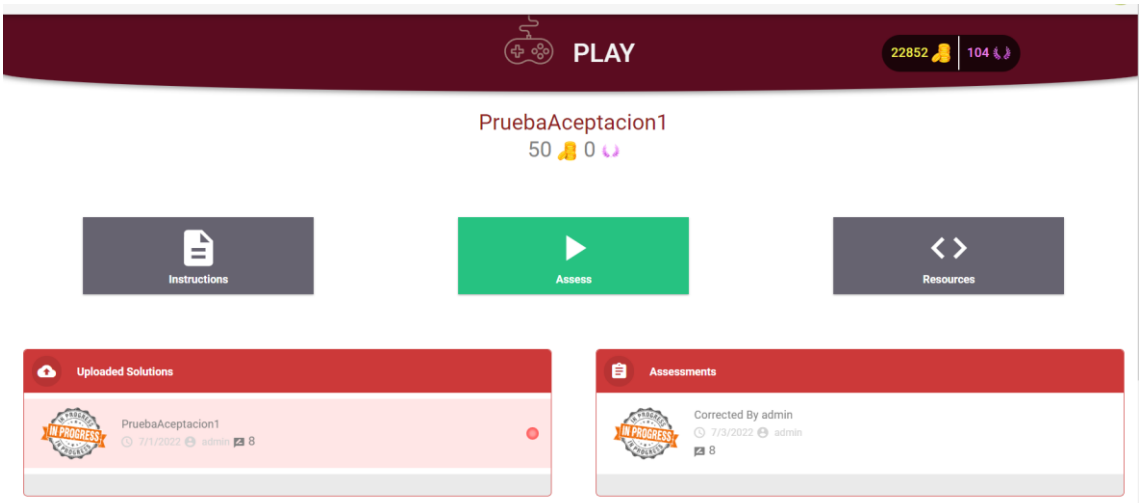
```
 */
public Rectangulo(double x, double y, double b, double h) {
    super(x, y);
    /**
     * ASys: Sustituido base12 por base
     */
    base = b;
    height = h;
}

public String toString(String a) {
    String t = "Rectangulo";
    /**
     * ASys: Sustituido base12 por base
     */
    if (base == height)
        t = "Cuadrado";
    /**
     * ASys: Sustituido base12 por base
     */
    return t + ":\n\t" + super.toString() + "\n\tBase: " + bas

/**
 * ASys: Esta es la declaracion correcta del metodo toString
 */
public String toString() {
    /**
     *  height;
     * ASys: Sustituido base12 por base
     */
    return base;
}
```

Error#7

En resumen, varias correcciones sobre un mismo método deben expresarse siempre juntas

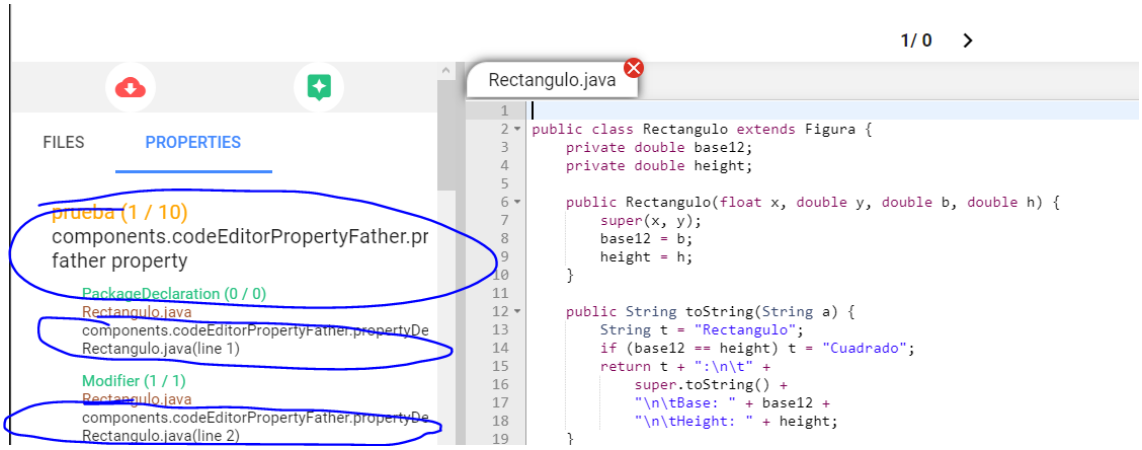
Estética y funcionalidad

1. Falta un botón para volver para atrás o barra de migas

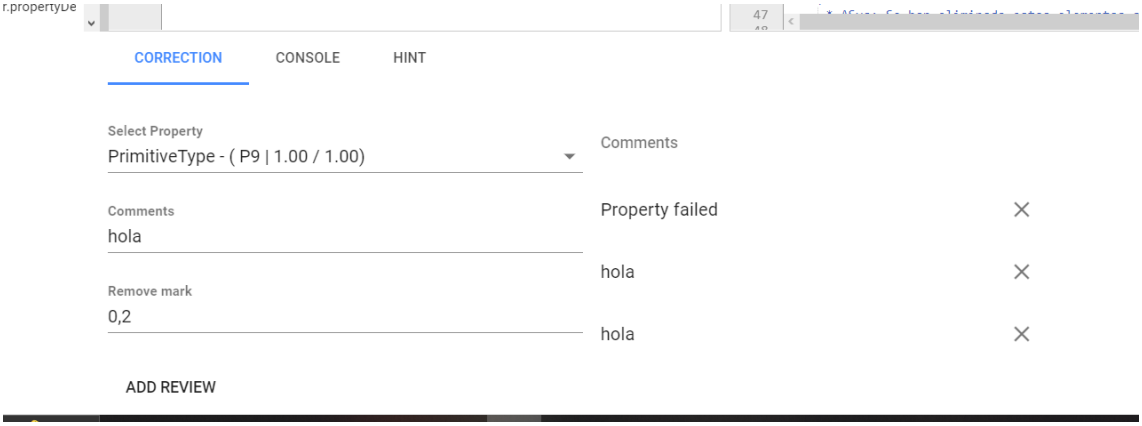


2. No entiendo bien este funcionamiento: He clicado en PruebaAceptacion2 en Uploaded solutions y se ha añadido en *Assesments* una entrada de Corrected By admin. Pero creo que puedo acceder tanto clicando ahí como clicando en el botón verde Assess

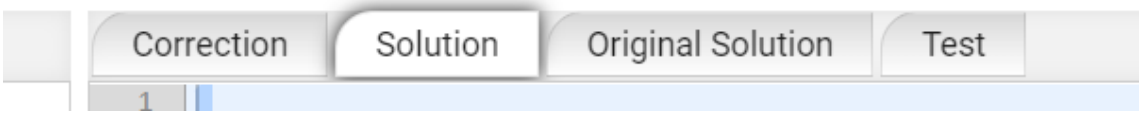
3. Hay ejercicios donde el botón verde pone “Assess” y otros donde pone “Solve”. Entiendo que son dos tipos de ejercicios diferentes, pero igual sería mejor un nombre más genérico: “Start” o algo así
4. Error traducción



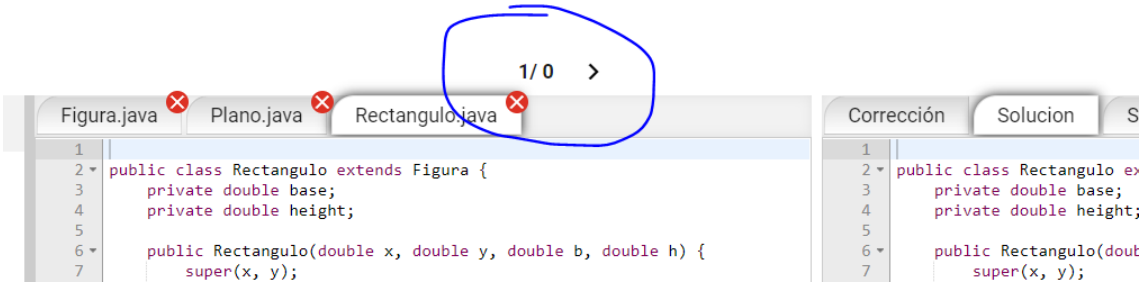
5. No entiendo bien que es esto:



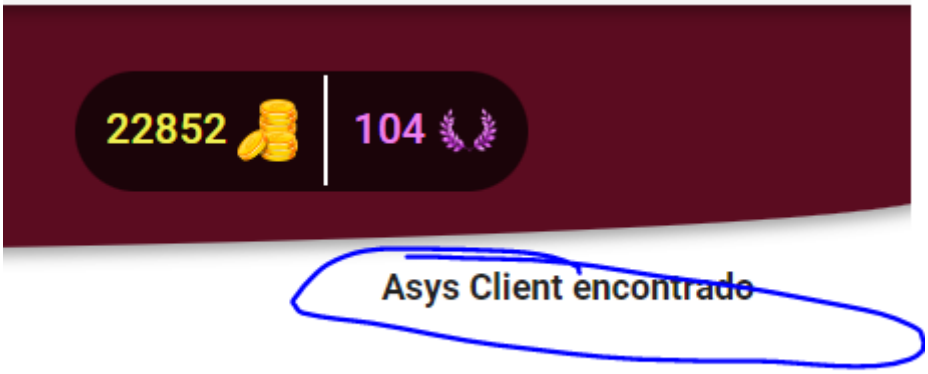
6. ¿Es normal que siempre tenga acceso a la solución? Acabo de sacar un 10 simplemente copiando y pegando la solución que me muestra el ejercicio. Esto es un poco incoherente con el ranking y los métodos de gamificación de ASys. Y aunque no existan estos métodos de gamificación, si al alumno le das la solución completa, este no se esforzará en aprender e intentar corregir.
7. ¿Diferencia entre Solution y Original Solution?



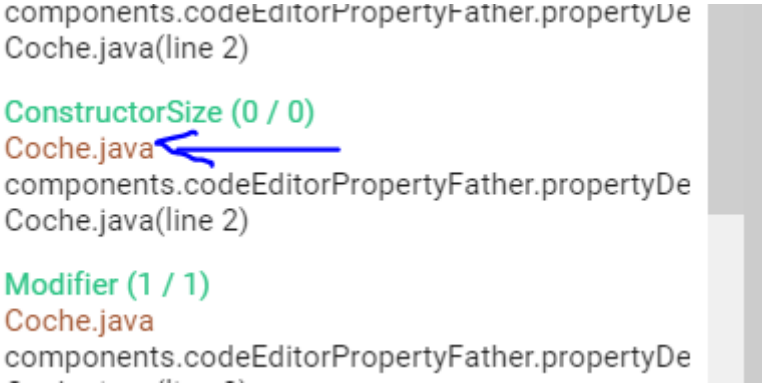
8. ¿Qué es esto?



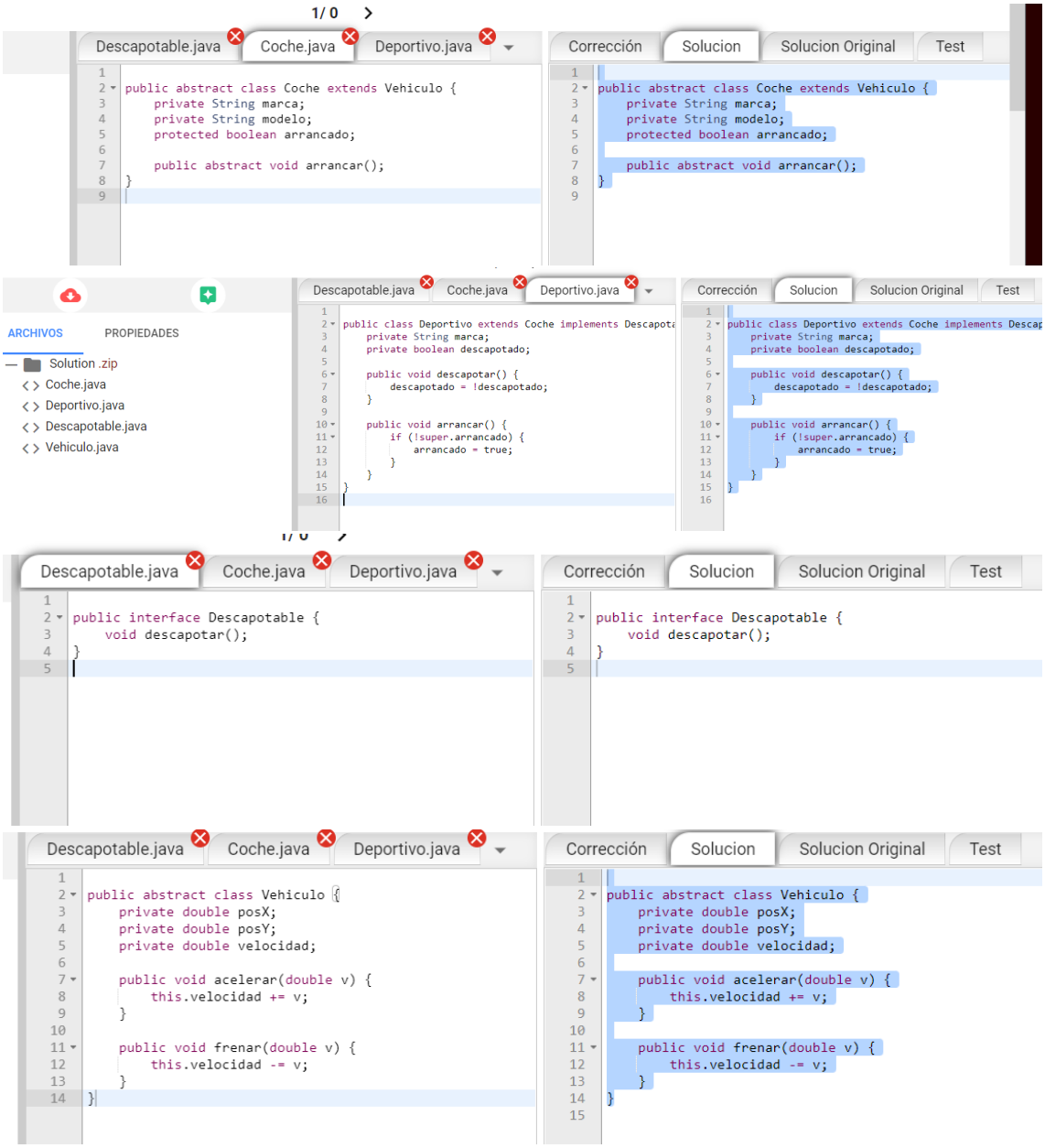
9. ¿Es necesario que aparezca que se ha encontrado el corrector de ASys?



10. Cuando el cursor pasa sobre el nombre del fichero, debería cambiar a la mano, puesto que es un texto clicable.



11. No sé qué he tocado pero he puesto la solución de PruebasAceptación1 pero le doy a corregir y no ocurre nada



Assessment ID: 417
Nota: 8 / 10

Resposive

1. Con una vista de móvil no consigo abrir el selector de ficheros y tampoco puedo ver los errores

2. La nota aparece sobre las monedas