UNIVERSIDAD
COMPLUTENSE
MADRID

Debuggy:
Depurador Declarativo de Python

Trabajo de fin de grado del Grado en Ingeniería Informática,
Facultad de Informática, Universidad Complutense de Madrid
2017-2018

José Javier Escudero Gómez
Sergio Freire Fernández
Sergio Ulloa López

Director: Adrián Riesco Rodríguez
Codirector: Enrique Martín Martín

Resumen

Nuestro proyecto consiste en un depurador declarativo para Python. Una herramienta con la que un programador puede depurar un programa en Python de manera sencilla. A través de preguntas que hace el depurador y el programador contesta, se va recorriendo un árbol que contiene las funciones que el programador quiere depurar, y se les asignan estados que representan la respuesta del programador. Consta de varias opciones de configuración, como el poder transformar el árbol en uno más reducido y el poder cambiar el tipo de estrategia de navegación por este. También posee dos tipos de interfaces de visualización, una interfaz gráfica(GUI) y otra de consola.

Palabras clave: depuración declarativa, recorridos, trazas, árbol, nodos, Python 3, Kivy.

Abstract

Our project is a declarative debugger for Python. It is a tool that a developer can use to debug his Python code in an easy way. Through questions the debugger asks to the developer, it traverses a tree that contains the functions executed during the erroneous computation, and establish a status that represents the answer given by the programmer. It consists of several configuration options, such as being able to transform the tree into a smaller one and being able to change the type of navigation strategy. It also has two types of visualization interfaces, a graphical interface (GUI) and a console interface.

Keywords: declarative debugging, routes, traces, tree, nodes, Python 3, Kivy.

Índice

Introducción

En esta sección trataremos las motivaciones que han impulsado la realización de este proyecto, así como las necesidades que se esperan satisfacer al finalizarlo.

Un depurador declarativo[1], al contrario de los depuradores tradicionales que se encuentran en los IDEs mejor conocidos, abandona la idea de obligar al programador a colocar puntos de interrupción en las partes del código que este viese oportuno. En vez de esto, se generará un árbol a partir de información relevante obtenida de diversas partes de la ejecución del programa a analizar para posteriormente recorrerlo preguntando al programador sobre la corrección de los resultados obtenidos hasta encontrar un fallo.

Esta alternativa a la hora de analizar y corregir un programa facilita en enorme medida la tarea de un programador, ya que resulta mucho más sencillo responder a simples cuestiones sobre si un valor es correcto o incorrecto, que comprobar cada una de las líneas de instrucciones del código o recorrer la traza. Esta ha sido la primera y más importante motivación de nuestro depurador.

Además, el hecho de la inexistencia de un depurador declarativo en Python en contrapartida a otros lenguajes como Java o Haskell que sí poseen un depurador de este tipo, también representa una gran necesidad para los desarrolladores de este lenguaje de programación. Adicionalmente, estos depuradores no solo existen para los nombrados lenguajes, sino que además cuentan con una interfaz gráfica (GUI) que facilita enormemente su uso. Esta constituye la segunda gran motivación de nuestro depurador.

Por último, Python es un lenguaje fácil de comprender y muy potente. Tiene estructuras de datos altamente eficientes y un acercamiento a la programación orientada a objetos simple pero muy efectivo. Su elegante sintaxis y su tipado dinámico lo hacen un gran lenguaje para diferentes usos como *scripting*, desarrollo de aplicaciones o, su uso más conocido, el análisis de *Big Data*[2].

La meta a alcanzar era conseguir un depurador declarativo para Python que cumpliese con los siguientes objetivos:

- Obtener la información esencial de un programa mediante la captura de ciertos datos de las funciones durante su ejecución. Estos datos son los devueltos por la función, los parámetros que recibe y los efectos laterales que sufren durante la ejecución de esta.

- Almacenar dicha información de manera arborescente en una clase propia que satisfaga todas nuestras necesidades.

- Recorrer los nodos siguiendo distintas estrategias de navegación.

- Tener diferentes tipos de respuestas para las preguntas de los recorridos.

- Lograr una reducción del tamaño final del árbol mediante una estrategia de transformación.

- Crear una interfaz gráfica como alternativa a la terminal.

Estas metas ha sido alcanzadas con éxito y, además, las funcionalidades de este depurador han sido ampliadas y mejoradas con más tipos de respuestas.

El código del proyecto se encuentra disponible para su descarga en GitHub con una licencia **GNU General Public License v3.0** ya que todos los miembros del equipo nos consideramos amantes y defensores del software libre. El enlace al repositorio es el siguiente: https://github.com/Wizsmiles/TFG_UCM/

Plan de trabajo

En este apartado explicamos qué pautas se han seguido y cómo han sido llevadas a cabo a la hora de organizar y trabajar en el proyecto.

Metodologías Ágiles - Extreme Programming

Durante el transcurso del proyecto han sido puestas en práctica las características que definen el desarrollo ágil de software.

Cada dos o tres semanas el equipo de desarrollo ha tenido reuniones presenciales con el cliente, el director y codirector del proyecto en este caso, cuya duración nunca ha excedido una hora. En el transcurso de estas reuniones, el equipo ha presentado los avances funcionales del proyecto al cliente y se han realizado una serie de pequeñas pruebas para verificar la integridad del código desarrollado hasta la fecha.

Tras verificar los avances, se pasaba a pensar y analizar los nuevos requisitos expuestos y se comenzaba a organizar brevemente dividiendo todo el desarrollo que debía llevarse a cabo para el próximo hito quincenal en pequeñas tareas independientes. Por último, cada miembro del equipo de desarrollo expresaba sus dudas personales surgidas durante el transcurso de la reunión y tras resolverlas se daba por concluido el encuentro.

Fuera de las reuniones siempre se ha tratado de trabajar en parejas de programadores rotando los integrantes de esta, aunque debido a los horarios tan dispares del equipo esto se ha tenido que hacer mediante el uso de herramientas tales como Skype o Discord. Esta práctica no solo se ha llevado a cabo a la hora de desarrollar el software, sino también a la hora de llevar a cabo una serie de diversas pruebas de la lógica del programa previas a la reunión presencial con los tutores.

La práctica descrita en los párrafos anteriores tuvo el efecto favorable de que todos los integrantes del equipo de desarrollo han podido llegar a conocer el código en su totalidad al estar casi siempre presentes durante su construcción y posterior prueba.

Hitos de trabajo

El Trabajo de Fin de Grado ha sido dividido en una serie de hitos quincenales donde se han ido presentando funcionalidades y depurando errores con la finalidad de cumplir las metas descritas en la introducción. A continuación se presenta cada uno de los hitos realizados y cuáles fueron sus principales características en el proceso de construcción del proyecto.

- **Primer hito**: Realizado durante la segunda semana del mes de enero. En este hito fue presentada una primera versión funcional de la función encargada de capturar la información necesaria del programa que se está analizando y que en Python se conoce como método trazador. Este método fue presentado junto a una versión básica de la clase, llamada **Nodo**, encargada de almacenar dichas trazas y organizarlas siguiendo una estructura de árbol.

- **Segundo hito**: Realizado durante la primera semana del mes de febrero. En esta iteración del proyecto se implementó el sistema de estados que podían ser asignados a los nodos del árbol, así como un sistema encargado de asignar el peso que tiene cada nodo y un recorrido auxiliar para poder verificar la integridad de los datos almacenados de una manera visual y sencilla. También fue implementada una serie de pequeños ejemplos para verificar la correcta construcción y recorrido de los datos capturados mediante las trazas.

- **Tercer hito**: Realizado durante la tercera semana del mes de febrero. Se implementaron los tres tipos de recorrido disponibles para que el usuario pudiera depurar su programa. También se definió una primera versión de la interfaz por terminal para que el usuario pudiera interaccionar con el depurador. Este hito supuso tener operativa una primera versión del depurador bastante limitada pero completamente funcional.
- **Cuarto hito**: Realizado durante la primera semana del mes de marzo. El sistema de estados fue ampliado con dos nuevas opciones. Se depuró la interfaz por terminal para que pudiera mostrar con diferentes colores cada nodo mostrado del árbol según el estado indicado por el usuario. En cuanto a la clase encargada de crear el árbol, se añadió la opción de transformar el árbol para reducir el número de preguntas.

- **Quinto hito:** Realizado a finales del mes de marzo. Tanto la función trazadora como la clase encargada de generar el árbol fueron modificadas para que fueran capaces de encontrar y almacenar excepciones. Todos los recorridos implementados fueron revisados para pulir pequeños fallos. Debido al tamaño que el código estaba comenzando a tener, se reorganizó toda la estructura interna del proyecto.

- **Sexto hito**: Realizado durante la segunda semana de abril. Se implementó el último estado disponible para el usuario, el estado *desconocido*. Incluir este estado supuso una revisión completa de todos los recorridos definidos hasta la fecha ya que el comportamiento especial de este estado no estaba siendo bien controlado. Se llevaron a cabo pequeñas pruebas con el marco de trabajo por defecto de Python para crear interfaces gráficas, pero fueron desechadas debido a la dificultad que suponía trabajar con dicho módulo.

- **Séptimo hito**: Último iteración de proyecto realizado a finales del mes de abril. En este hito fue presentada una interfaz gráfica funcional y se añadieron los algoritmos *mergesort* y *quicksort* como ejemplos para ser depurados por la herramienta. También se implementó la opción de que el depurador pudiera ser iniciado en modo consola o interfaz gráfica, junto a la opción de fusionar o no los nodos iguales del árbol.

Algo que puede llamar la atención al revisar la lista de hitos es el porqué se comenzó en enero el proyecto en lugar de comenzar con el inicio del curso. El motivo de esto fue porque todos los miembros del grupo decidieron pasar desde octubre hasta finales de ese año practicando con Python y asimilando los conceptos básicos de la depuración declarativa antes de meterse de lleno en el diseño y construcción del depurador ya que era un campo completamente nuevo para todos.

Durante el transcurso del mes de mayo, el equipo decidió seguir realizando pequeñas pruebas con el proyecto ya finalizado para verificar el correcto funcionamiento del mismo. Debido a unos pequeños problemas encontrados durante las pruebas de la interfaz gráfica, el equipo tuvo que volver a revisarla. Este proceso de corrección no fue tomado como un hito oficial del proyecto ya que no estuvieron implicados los tutores ni se produjo ninguna reunión con ellos.

Al mismo tiempo que se estaban llevando a cabo estas pruebas y correcciones de errores, el contenido de la memoria fue dividido y asignado a cada uno de los miembros del equipo. Una vez todas fueron completadas, el equipo se reunió de forma presencial y unió todas las partes para articular la memoria y mejorarla hasta llegar al contenido final.

Estructura de la memoria

A la hora de presentar todo el trabajo realizado en este documento, se ha decidido dividir toda la información en las siguientes secciones.

- Preliminares: En esta parte se ofrece al lector el contexto e información necesaria para poder entender el trabajo realizado.

- Diseño del depurador: Aquí, se cuenta de manera técnica y en detalle cada uno de los procesos llevados a cabo en el proyecto acompañados con fragmentos de código e imágenes explicativas.

- Contribuciones: Cada uno de los miembros del grupo ha explicado cuál ha sido su trabajo tanto en el diseño del depurador como en la propia memoria.

- Instalación y dependencias: Explica que bibliotecas de terceros son necesarias para poder ejecutar el proyecto y cómo instalarlas.

- Conclusiones y trabajo futuro: Aquí se explican cuáles han sido las sensaciones finales obtenidas al terminar el proyecto y qué mejoras han sido pensadas para ser implementadas en un futuro.

Introduction

In this section we present the motivations that have driven us to make this project and the needs we expect to satisfy with it.

A declarative debugger, unlike the traditional debuggers that we can find in well-known IDEs, abandons the idea of placing breakpoints in the sections of code to debug. Instead of this, the idea is to generate a tree from the relevant information obtained through several parts of the execution of the program to debug. Once the tree has been built, the debugger traverses it while asking the programmer about the correction of the obtained results.

This alternative when analyzing and correcting a program greatly eases the task of the programmer, because it is much easier to answer simple questions about the results of a specific function than check a lot of lines of source code. This was the first and most important motivation to implement this debugger.

In addition, the fact that there was no other declarative debugger for Python, unlike other languages like Java or Haskell that already has a declarative debugger, also represents a great need for Python developers. Additionally, these debuggers do not only exist for the aforementioned languages, but also have a graphical interface (GUI) that greatly facilitates their use. This is the second motivation of our debugger.

Finally, Python is a powerful language that is easy to understand. It has very efficient data structures and a simple approach to object-oriented programming. Its elegant syntax and his duck typing make Python a great language for different usages like scripting, application development or, the best known, Big Data analysis.[2]

The goal to achieve was to obtain a declarative debugger for Python that should satisfy these points:

- Get the essential information of a program through the capture of specific data of the functions while its execution. This data corresponds to the return values of the function, the parameters that the function receive, and the side effects generated by the function execution.

- Save that information in a tree way on a class that satisfies all the needs we have.

- Navigate the nodes following several navigation strategies.

- Show different kinds of answers for the debugging questions

- Achieve a reduction in the size of the tree through a transformation strategy.

- Create a graphic user interface as an alternative to the console terminal.

These goals have been achieved successfully and, furthermore, the features of this debugger have been extended and improved with more types of answers.

The project code is available for download on GitHub with a **GNU General Public License v3.0** since all the members of the team consider ourselves lovers and defenders of free software. The link to the repository is the following: https://github.com/Wizsmiles/TFG_UCM/

Workplan

In this section we explain the guidelines that we have followed and how they have been carried out when organizing and working on the project.

Agile Methodologies - Extreme Programming

During the course of the project the characteristics that define the agile development of software have been put into practice.

Every two or three weeks the development team had face-to-face meetings with the client, the director and co-director of the project in this case, whose duration has never exceeded one hour. During these meetings, the team presented the functional progress of the project to the client and several small tests to verify the integrity of the developed code until that date.

After verifying the progress, it was time to think and analyze the new requirements and to begin to briefly organize, dividing all the development that had to be carried out for the next biweekly milestone into small independent tasks. Finally, each member of the development team expressed their personal doubts that arose during the course of the meeting and, after resolving them, the gathering was concluded.

Out of the reunions, we always tried to work in pairs of programmers, rotating their members, although, due to the different schedules of the team, this had to be done through the use of tools like Skype or Discord. This practice has not only been used at the time of developing the software, but also at the time of carrying out a series of different tests of the program logic prior to the face-to-face meeting with the tutors.

The practice described in the previous paragraphs had the favorable effect that all the members of the development team have been able to get to know the code entirely since they were almost always present during its construction and subsequent test.

Work milestones

The Degree Thesis has been divided into a series of biweekly milestones where its functionalities have been presented and its bugs have been debugged in order to meet the goals described in the introduction. Below we present these milestones and their main characteristics:

- **First milestone**: Carried out during the second week of January. In this milestone it was presented a first functional version of the function in charge of capturing the necessary information of the program that is being analyzed and that in Python is known as a tracer method. This method was presented with a basic version of the class, called Node, responsible for storing these traces and organizing them following a tree structure.

- **Second milestone**: Carried out during the first week of February. In this iteration of the project, the system of states that could be assigned to the nodes of the tree was implemented, as well as a system in charge of assigning the weight that each node has and an auxiliary method to verify the integrity of the stored data in a visual and simple way. A series of small examples were also implemented to verify the correct construction and traversal of the data captured by the traces.

- **Third milestone**: Carried out during the third week of February. The three types of available routes were implemented so that the users could debug their program. A first version of the terminal interface was also implemented so that the user could interact with the debugger. This milestone meant that the first version of the debugger was quite limited but fully functional.

- **Fourth milestone**: Carried out during the first week of March. The state system was expanded with two new options. The terminal interface was debugged so that it could show each node with different colors according to the state indicated by the user. As for the class responsible for creating the tree, the tree transformation option was added to reduce the number of questions.

- **Fifth milestone**: Carried out at the end of March. Both the tracer function and the class responsible for generating the tree were modified so that they were able to find and store exceptions. All the routes implemented were revised to polish small faults. Due to the size that the code was reaching, the internal structure of the project was reorganized.

- **Sixth milestone**: Carried out during the second week of April. The last state available to the user, the unknown state, was implemented. Including this status meant a complete revision of all the routes defined to date since the special behavior of this state was not being appropriately controlled. Small tests were carried out with Python's default framework to create a graphical interface, but they were discarded due to the working difficulty with that module.

- **Seventh milestone**: Last iteration of the project carried out at the end of April. In this milestone a functional graphical interface was presented and the mergesort and quicksort algorithms were added as examples. We also implemented the option to start the debugger in console mode or with the graphical interface, together with the option to merge or not the equal nodes of the tree.

Something that can draw attention when reviewing the list of milestones is why the project was started in January instead of at the beginning of the course. The reason for this was that all the members of the group decided to go from October to the end of that year practicing with Python and assimilating the basic concepts of declarative debugging before getting into the design and construction of the debugger since it was a whole new field for us all.

During the month of May, the team decided to continue to carry out small tests with the finished project to verify its correct operation. Due to some minor problems encountered during the graphical interface tests, the team had to revise it. This correction process was not taken as an official milestone since neither the tutors were involved did any meeting take place with them.

At the same time that these tests and bug fixes were being carried out, the content of this memory was divided and assigned to each member of the group. Once all parts were completed, the team met in person and joined all the parts together to articulate the memory and improve it until the final content.

Memory structure

When presenting all the work done in this document, it has been decided to divide all the information in the following sections:

- Preliminaries: This part offers the reader the context and the necessary information to understand the work done.

- Design of the debugger: Here, each of the processes carried out in the project is explained, accompanied by fragments of code and explanatory images.

- Contributions: Each member of the group has explained what their work has been, both in the design of the debugger and in the memory itself.

- Installation and dependencies: Explains what third-party libraries are necessary to be able to run the project and how to install them.

- Conclusions and future work: Here are explained what have been the final feelings obtained at the end of the project and what improvements have been designed to be implemented in the future.

Preliminares

En esta sección de la memoria se explica en qué consiste la depuración declarativa, así como también hablaremos de cuáles son las características principales del lenguaje de programación en el que vamos a trabajar. Una vez se hayan asimilado estos conceptos, el lector estará preparado para adentrarse en los apartados posteriores de la memoria donde se hablará de manera técnica y en detalle sobre todo el proceso de diseño del depurador.

Python

Es un lenguaje de programación interpretado de alto nivel, diseñado para ser utilizado en un gran dominio de aplicaciones informáticas. Creado y sacado a la luz en 1991 por Guido van Rossum, tiene una filosofía que pretende reforzar la legibilidad del código utilizando, por ejemplo, sangría para delimitar los bloques de código.[3]

Tiene un sistema de tipado dinámico y una administración de memoria automática. Soporta varios paradigmas de programación (es decir, es multiparadigma), incluyendo programación orientada a objetos, programación imperativa, funcional y procedural, y tiene una amplia biblioteca estándar.

Los intérpretes de Python están disponibles para muchos sistemas operativos. CPython, el referente de implementación de Python, es software de código abierto y tiene un modelo de desarrollo basado en la comunidad, como hace prácticamente todo su elenco de implementaciones. CPython es gestionado por la organización sin ánimo de lucro Python Software Foundation.

Eric Matthes, un programador de Python de cierto renombre y escritor de varios libros sobre dicho lenguaje, alabó las grandes cualidades de Python con la siguiente frase:

> *"Todos los años me pregunto si continuar usando Python o si tal vez sea mejor aprender un nuevo lenguaje… Uno que sea más reciente en el mundo de la programación. Pero sigo concentrándome en Python por muchas razones. Python es un lenguaje increíblemente eficiente: tus programas harán mucho más con menos líneas que en otros lenguajes. Su sintaxis también te ayudará a escribir código 'limpio'. Tu código será fácil de leer, fácil de depurar y fácil de extender comparado con otros lenguajes."* [4]

Sintaxis y semántica

Python está diseñado para ser muy legible. Un gran número de veces utiliza palabras en inglés donde otros lenguajes de programación utilizan símbolos de puntuación (por ejemplo **not** en lugar de **!**). A diferencia de otros lenguajes, no usa llaves para diferenciar bloques entre sí, y los puntos y coma al final de cada línea de código son totalmente opcionales. Tiene pocas excepciones semánticas y casos especiales.

Sangría

Python utiliza sangría en lugar de corchetes o palabras clave para delimitar bloques. Un aumento en la sangría comienza después de ciertas declaraciones, como por ejemplo **if**, **elif** o **else**; un decremento en la misma significa el final del bloque correspondiente. Por lo tanto, la representación visual del código se integra perfectamente con la estructura semántica del mismo.

Tipado

Python usa *duck typing* (tipado dinámico) y tiene objetos tipados pero los nombres de variables no están tipados. Las restricciones de tipado no se comprueban en tiempo de compilación; es decir, las operaciones pueden fallar, significando esto que el objeto dado puede no ser de un tipo adecuado. A pesar de utilizar tipado dinámico, Python es un lenguaje fuertemente tipado, prohibiendo operaciones que no están bien definidas (ej. añadir un número a un string) en lugar de intentar hacer que cobren sentido silenciosamente.

Python permite al programador definir sus propios tipos de datos utilizando clases, que son lo más utilizado para la programación orientada a objetos. Las nuevas instancias de las clases son construidas llamando a la clase (ej. ClaseEjemplo), y las clases son instancias de la metaclase *type*, permitiendo la metaprogramación y la reflexión.

Depuración declarativa

En este apartado de la memoria se va a explicar en qué consiste la depuración declarativa, cómo funciona y cuáles son sus principales características que la diferencian de la depuración tradicional.

¿Qué es la depuración declarativa?

Es un tipo de depuración que se define de la siguiente forma:

"La depuración declarativa es una técnica de depuración semiautomática que permite al programador depurar un programa sin la necesidad de ver el código fuente del mismo. El depurador genera preguntas sobre los resultados obtenidos en diferentes ejecuciones y el programador solo tiene que responderlas para encontrar el error." [5]

Con esta definición en mente fue con la que iniciamos nuestro trabajo, como bien dice en el artículo, "depurar es una de las tareas más costosas y menos automáticas de la ingeniería del software", el objetivo era desarrollar una herramienta que permitiese la depuración en Python de manera sencilla y automática, facilitando la tarea al programador y dándole un lugar concreto (una función) dónde comenzar a depurar errores.

¿Cómo funciona?

El depurador comienza ejecutando una expresión a elección del programador y a partir de esta se construye un **árbol de ejecución** en el que se almacenan todas las funciones invocadas en la ejecución de la primera, a continuación se comienza la depuración haciendo

preguntas al programador sobre cada función y en base a las respuestas se va recorriendo el árbol siguiendo unas **estrategias de navegación** hasta encontrar un ***nodo buggy*.**

¿Qué es un árbol de ejecución?
Para empezar un árbol es el nombre por el que se le conoce a la estructura de datos no lineal y acíclica, que consiste en una conjunción de nodos, vértices y aristas enlazados. Un árbol sin nodos se denomina **vacío** o **nulo.** Un árbol no vacío dispone de múltiples niveles de nodos ordenados de forma jerárquica. Definimos los siguientes conceptos:

- **Raíz:** El nodo superior de un árbol.
- **Hijo:** Un nodo conectado directamente con otro cuando se aleja de la raíz.
- **Padre:** Un nodo conectado directamente con otro cuando se acerca a la raíz.
- **Descendiente:** Un nodo accesible por descenso repetido de padre a hijo.
- **Hoja:** Un nodo sin hijos.
- **Profundidad:** La profundidad de un nodo es el número de aristas desde la raíz del árbol hasta un nodo.

Para más información[6]

En nuestro caso, cada nodo almacena información relacionada con cada función ejecutada, sus valores de entrada y salida, los resultados y los efectos laterales generados. En la sección *Construcción del árbol* podrán verse dos ejemplos de cómo es el resultado final.

¿Qué es un efecto lateral?
Cuando una función recibe como parámetros de entrada objetos o estructuras de datos tales como listas o diccionarios, estas pueden ser modificadas durante la ejecución. Al ocurrir esto, cuando finaliza la función, estos datos tienen distinto valor que cuándo comenzó la ejecución y, al ser mutables, conservan dicho valor.

Este suceso es denominado efecto lateral y debe ser tenido en cuenta a la hora de crear y almacenar los nodos ya que pueden ser una de las posibles causas de la aparición del ***nodo buggy*.**

¿Qué respuestas puede dar el programado?
Hay múltiples tipos de respuesta. Estas van desde lo más básico (correcto e incorrecto), hasta respuestas mucho más elaboradas y complejas. En nuestro caso se dispone de 5 respuestas que el programador puede asignar a cada función.

- **Válido:** significa que el resultado para esos parámetros en concreto es correcto.

- **Incorrecto:** la llamada contiene un error, el resultado no es el esperado.

- **Confiar:** la llamada que estamos depurando es una función que estamos seguros de que no contiene errores independientemente de los parámetros.

- **Inaceptable:** la llamada en si no es incorrecta, sino que debe su mal funcionamiento a los parámetros que recibe(p.ej. una función de ordenación recibe como lista a ordenar un entero).

- **Desconocido:** el programador no es capaz de determinar si el resultado es el esperado o no.

Cada una de estas respuestas asigna al nodo un estado que definiremos en la sección del recorrido.

¿Cuándo hay un error? ¿Qué es un nodo buggy?

Para identificar un error debemos tener en cuenta lo siguiente, un nodo es erróneo cuando la función no sigue el comportamiento esperado. En este caso hay varios escenarios: por un lado puede que el resultado de la función difiera de lo esperado, por otro lado también es posible que el resultado sea el esperado para la entrada, pero los efectos laterales no son los esperados.
Cuando el programador encuentra un nodo erróneo tenemos que decidir si es lo que denominamos un ***nodo buggy***, es decir, la función que corresponde al nodo es la que se debe depurar. Para tomar esta decisión se tiene en cuenta que un nodo es *buggy* si él mismo es incorrecto pero todos sus hijos son correctos o no tiene hijos.

¿Qué es una estrategia? ¿Cuáles utilizamos?

Primero, consideramos estrategia de recorrido al método o algoritmo utilizado para evaluar el árbol de ejecución mediante la asignación de diferentes estados a cada uno de sus nodos. La estrategia es la encargada de decidir en cada caso cuál será el nodo que se evaluará a continuación. El uso de estrategias es variable y el programador puede elegir en cada momento cual de todas ellas desea utilizar.

Nuestro depurador consta actualmente de varias estrategias de recorrido, basadas en diferentes algoritmos para poder abordar el problema de maneras distintas. Actualmente hay tres definidas ***Top Down, Heaviest First y Half Down***. Estas serán explicadas con más detalle en su correspondiente sección de la memoria.

Uno de los principales problemas que tiene la depuración declarativa es el tamaño del árbol resultante tras analizar todo el código del programa. Un árbol compuesto por una elevada cantidad de nodos puede resultar muy pesado de revisar para el programador de forma manual independientemente de la estrategia de recorrido que haya sido elegida para ello.

Para tratar de subsanar este problema, surge lo que es denominado como “transformación del árbol”. Esta técnica consiste en analizar toda la estructura de datos una vez el proceso de construcción ha finalizado y fusionar los nodos cuyos hijos hacen una llamada a la misma función que el padre, es decir, cuando se tienen métodos recursivos.

Un factor importante a tener en cuenta es que no funciona en todos los tipos de recursión, sino que únicamente este método es eficaz cuando se trata de una recursión de tipo final. Esto se debe a que en una recursión de tipo final, el último nodo generado en lo que

podemos denominar “cadena recursiva” es el que tiene el valor útil y que es devuelto a su padre, este al abuelo del hijo y así sucesivamente hasta llegar al inicio de la cadena.

Trazador personalizado

Al contrario que otros lenguajes de programación de alto nivel como pueden ser Java o C++, Python no se caracteriza por poseer unas herramientas de depuración de código tan simples, intuitivas y visuales para el desarrollador como las que podemos encontrar al usar cualquier IDE estándar.

Para poder llevar a cabo las tareas de depuración de cualquier código, Python Software Foundation implementó y puso al servicio del diseñador **sys. sys**, cuyo nombre es una abreviatura de la palabra *system*, es el módulo de Python encargado de proveer las funcionalidades y variables relacionadas de forma directa con el intérprete del lenguaje. Para el caso que nos atañe, este módulo nos permite interactuar y modificar lo que Python denomina **tracefunc** o función trazadora.

Una función trazadora es la encargada de recopilar los datos de un programa. Al ser implementada por el programador, esta puede recibir cualquier nombre ya que no requiere realizar una sobrecarga de una función definida previamente por el sistema. Sin embargo, este método siempre debe recibir tres parámetros específicos para su correcto funcionamiento a la hora de llevar a cabo la captura de información.

- **Frame**: Contiene toda la información acerca la función del código que se está depurando en este momento. De entre todos los datos que nos aporta esta variable algunos de los más interesantes a la hora de llevar a cabo una depuración declarativa son el nombre de la función, los parámetros de entrada que recibe al ser ejecutada y los parámetros o variables que se han visto modificados tras finalizar la ejecución de esta.

- **Event**: Determina el tipo de evento que ha hecho ejecutar la función trazadora. Más adelante serán nombrados y explicados todos los tipos de eventos encargados de llevar a cabo dicha función.

- **Arg**: Almacena los datos retornados por la función del código que la ha invocado. Por defecto este parámetro almacena el valor **None**, lo cual es ideal en caso de que el método encargado de llamar a la función trazadora no devuelva ningún tipo de dato.

Esta función se caracteriza por trabajar siempre en un segundo plano mientras el resto del programa se está ejecutando. Esta solo se ejecutará cuando ocurra una serie de eventos específicos en nuestro programa. Al contrario que la función en sí estos eventos no pueden ampliarse, lo que supone que el programador debe ceñirse a ellos.

- **‘call’**: Ocurre antes de que un método de nuestro código sea ejecutado. El parámetro **Arg** recibe un valor None.

- **‘line’**: Ocurre antes de que una línea de código sea ejecutada. Es el equivalente a poner un *breakpoint* e ir depurando un programa línea a línea en cualquier IDE de cualquier lenguaje. Al igual que el evento anterior, el parámetro **Arg** se envía con valor None.

- **‘return’**: Sucede justo antes de que un método devuelva un valor. El parámetro **Arg** almacena el valor que ha sido retornado. Este evento también es lanzado por el sistema incluso cuando un método no tiene retorno explícito.

- **‘exception’**: Detecta cuando hay una excepción en el programa. El parámetro **Arg** recoge el tipo de excepción producida, como puede ser un **pop()** en una lista vacía o una división entre cero, como si se tratase de un retorno de función.

- **‘c_call’**: Ocurre antes de que una función en C sea ejecutada. Al contrario que su homónimo ‘call’, el parámetro **Arg** no recibe un valor None, sino la función C encapsulada como un objeto.

- **‘c_return’**: Sucede después de que una función en C retorne un valor. Al contrario que su homónimo ‘return’, el parámetro **Arg** recibe un valor None en vez del valor retornado.

- **‘c_exception’**: Detecta cuando hay una excepción en una función en C. Al contrario que su homónimo ‘exception’, el parámetro **Arg** recibe un valor None en vez del tipo de excepción producida.

Una vez se tiene diseñada una función trazadora personalizada, esta debe sustituir a la que el módulo sys almacena por defecto. Para poder manipular el uso de una o varias de estos métodos de depuración, este módulo nos ofrece dos sencillas funciones para poder trabajar fácilmente.

- **gettrace()**: Devuelve el método trazador actual. Esto nos permite poder obtener el trazador por defecto de Python antes de usar el nuestro. Gracias a esto podremos salir de nuestro modo de depuración personalizado y volver a un modo de ejecución estándar.

- **settrace(func)**: Asigna el método trazador a ejecutar. Una vez se haya definido este, todo el código posterior a esta función será analizado. Si se quiere depurar más de un hilo de ejecución, se tendrá que llamar a este método tantas veces como hilos tengamos.

Como puede apreciarse el proceso de depuración en Python, si bien no es tan rápido e intuitivo como en los lenguajes citados al inicio de esta sección de la memoria no es muy difícil obtener todos los datos de nuestro programa al momento de su ejecución.

Lo que sin duda no es trivial es el poder definir un método trazador óptimo de propósito general junto a una estructura encargada de almacenar todos los datos recopilados para su posterior análisis individual.

Kivy

Kivy[7,8] es una biblioteca de código libre de Python para desarrollar aplicaciones móviles y otros tipos de aplicaciones multitáctiles con una interfaz natural de usuario. Puede ejecutarse en Android, iOS, Linux, OS X y Windows. Distribuida bajo los términos de la licencia MIT, Kivy es software libre y gratuito.

Kivy es el principal framework desarrollado por la organización Kivy, junto con Python for Android, Kivy iOS, y varias otras bibliotecas pensadas para ser usadas en todas las plataformas. Kivy también soporta Raspberry Pi.

Este framework contiene todos los elementos para construir una aplicación como:

- Extenso soporte de entrada para ratón, teclado, TUIO (*Tangible User Interface*) y eventos multitáctiles específicos del sistema operativo.

- Una biblioteca gráfica usando solamente OpenGL ES 2 y basada en Vertex Buffer Object y *shaders* (sombreadores).

- Un vasto elenco de *widgets* multitáctiles.

- Un lenguaje intermedio (Kv) usado para diseñar fácilmente widgets personalizados.

Kivy es la evolución natural de PyMT project, y es recomendado para nuevos proyectos.

Ejemplo de código

Este código representa la implementación de una aplicación de dibujo muy simple:

```
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.graphics import Color, Ellipse
class MyPaintWidget(Widget):
    def on_touch_down(self, touch):
        with self.canvas:
            Color(1, 1, 0)
            d = 30.
            Ellipse(pos=(touch.x - d / 2, touch.y - d / 2), size=(d, d))

class MyPaintApp(App):
    def build(self):
        return MyPaintWidget()

if __name__ == '__main__':
    MyPaintApp().run()
```

Este código, genera un canvas a modo de lienzo. Cuando se hace click en una posición de este, se captura la posición en X e Y y dibuja una caja de selección de grosor 30x30 de color amarilla.

Lenguaje Kv

El lenguaje Kivy (Kv) es un lenguaje dedicado a describir la interfaz de usuario y sus interacciones. Como con otros lenguajes de marcado, es posible crear fácilmente una interfaz de usuario entera y añadir interacción. Por ejemplo, para crear un diálogo de carga que incluye un explorador de archivos y un botón de Cargar/Cancelar, uno podría crear el widget base en Python y luego construir la interfaz en Kv.

Ejemplo

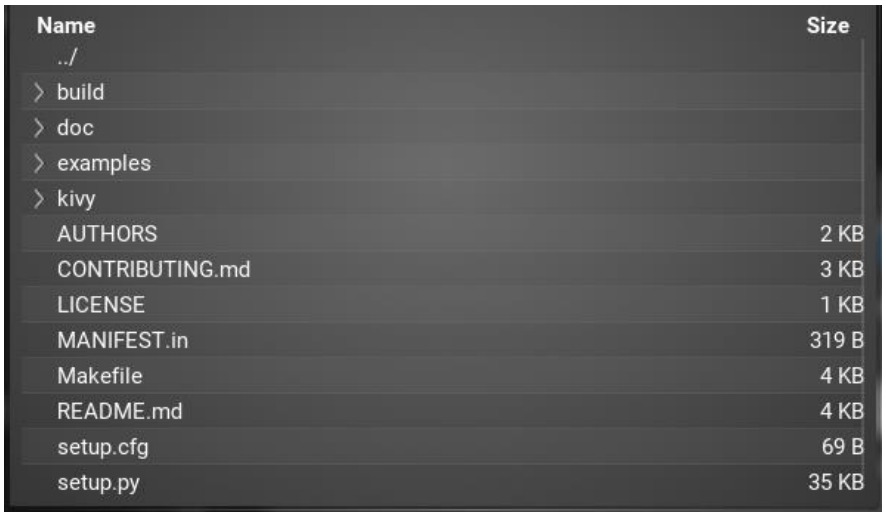
En main.py:

```
class LoadDialog(FloatLayout):
    def load(self, filename): pass
    def cancel(self): pass
```

Y en el archivo Kv asociado:

```
#:kivy 1.4.0
<LoadDialog>:
    BoxLayout:
        size: root.size
        pos: root.pos
        orientation: "vertical"
        FileChooserListView:
            id: filechooser
        BoxLayout:
            size_hint_y: None
            height: 30
            Button:
                text: "Cancel"
                on_release: root.cancel()
            Button:
                text: "Load"
       on_release: root.load(filechooser.path, filechooser.selection)
```

Este código, genera un selector de ficheros tras recibir una ruta junto al peso de cada uno de los ficheros generados.



Diseño del depurador

En esta sección de la memoria se va a narrar paso a paso cada uno de los procesos, de forma ordenada, que fueron necesarios a la hora construir el depurador.

Construcción del árbol

Tal y como se definió en el plan de trabajo, el primer objetivo que debíamos cumplir en la misión de diseñar un depurador declarativo era el de obtener la información de un programa mediante la traza de este y almacenar estos resultados en una clase. En esta sección de la memoria se va a presentar el método trazador, la clase **Nodo** encargada de crear el árbol de depuración y dos sencillos ejemplos para explicar cómo funcionan ambas partes en conjunto.

El primero de estos es una construcción al uso, mientras que la segunda se usa cuando hay una excepción que hace finalizar el programa de forma anticipada. Ambos ejemplos, junto a varios más, están incluidos en el depurador para ser ejecutados y ver los resultados de primera mano.

Método trazador

Como ya se ha explicado previamente en el apartado Trazador Personalizado, necesitábamos crear una función que recibiese una serie de parámetros específicos definidos en la documentación oficial de Python y a partir de estos construir el árbol de ejecución. En nuestro caso, el resultado final tras varias iteraciones del proyecto probando varios ejemplos es el siguiente.

```
def trace_calls(frame, event, arg):
    global wait, cont, contWait, arbol

    co = frame.f_code
    f_name = co.co_name

    if not wait:
        if event == "return" or event == "exception":
            if f_name != "_ag" and f_name != "encode":
                if event == "exception":
                    arg = arg[1]
                    wait = True

                params = frame.f_locals
                arbol.insertarValor(arg, cont)
                arbol.insertarParamsMods(params, cont)
                cont -= 1
```

```
    if event == "call":
        if f_name != "_ag" and f_name != "encode":
            params = frame.f_locals
            cont += 1

            if cont == 1:
                arbol.setNombre(f_name)
                arbol.setParamsEntrada(params)
            else:
                hijo = Nodo.Nodo()
                hijo.setNombre(f_name)
                hijo.setParamsEntrada(params)
                arbol.insertar(hijo, cont)
  else:
    if contWait < 3:
        contWait += 1
    else:
        wait = False
        contWait = 0
return trace_calls
```

Tal y como se puede apreciar, lo primero que llevamos a cabo es almacenar en la variable f_name el nombre de la función que ha provocado la ejecución del método trazador. Este nombre posteriormente será utilizado para comprobar si no se trata de una función propia de Python que actúa oculta para el programador. Si es una función del programa a analizar, el nombre será almacenado en el nodo correspondiente del árbol de depuración.

De todos los eventos disponibles, a la hora de realizar una depuración declarativa solo nos interesan los relacionados con llamar una función (**call**), devolver un valor en una función (**return**) o cuando se ha producido una excepción inesperada (**exception**). Volviendo al código anterior, vemos que aunque son tres los estados con los que trabajamos, a nivel de lógica tan solo distinguimos dos opciones.

- Cuando el evento recibido es el resultado de una llamada a una función crearemos un nodo en el que almacenaremos el nombre de dicha función y los parámetros que recibe para después almacenarlo en el árbol como hijo de un nodo previo, utilizando hijo.setNombre(f_name) para asignar el nombre, hijo.setParamsEntrada(params) para almacenar los parámetros que recibe la función y por último insertando el hijo en el árbol mediante arbol.insertar(hijo,cont). Por el contrario, si el árbol está vacío se almacena como raíz mediante arbol.setParamsEntrada(params).

- Cuando el evento recibido es el resultado de devolver un valor al finalizar una función o se ha producido una excepción, almacenaremos en el último nodo insertado en el árbol el valor retornado o excepción, así como los parámetros de la función que han sido modificados a causa de los efectos laterales de esta. Esto se realiza en arbol.insertarValor(arg,cont) **y** arbol.insertarParamsMods(params,cont)**.**

En las explicaciones anteriores puede ver el uso del objeto **arbol** y el parámetro **cont**. El primero hace referencia a un objeto de la clase **Nodo** y constituye el nodo raíz sin datos que como hemos explicado debe rellenarse con la información del primero método analizado. El segundo parámetro es, como se puede intuir por su nombre, un contador que ayuda a gestionar en que nivel del árbol debe insertarse un nuevo nodo. Esto es necesario ya que según el número de funciones anidadas de un método, una rama del árbol puede crecer en varios niveles y al finalizar este método y se llame a otra función, el nuevo nodo a insertar deba ir varios niveles más abajo en otra rama de la estructura arbórea.

Una característica importante del evento de captura de excepción es que tras producirse y almacenarse en un nodo, genera tres retornos con valores basura que deben ignorarse. Para mantener la integridad del árbol, implementamos una comprobación (**wait**) para controlar si el último evento capturado ha sido una excepción. En caso de que efectivamente se ha producido una excepción, se procede a filtrar los datos basura provenientes de las tres siguientes llamadas al trazador llevando un conteo de las mismas para poder desmarcar nuestra variable de control.

Árbol de depuración

Para almacenar todos los datos que se van recopilando con el método trazador, el equipo llegó a la conclusión de que era necesario implementar una estructura con forma de árbol.

El principal motivo por el que nos decantamos a elegir un árbol es que este nos permite dividir fácilmente cualquier código en una cantidad indefinida de nodos entre los cuales hay una relación de parentesco. Este parentesco bien puede ser padre e hijo entre una función que tiene anidadas otras funciones en su interior, o bien varias funciones hermanas que dependen de un mismo padre. Nuestra clase **Nodo** no dista demasiado de cualquier nodo visto durante el transcurso de la carrera. Los atributos que componen a un nodo son los siguientes.

- **nFuncion**: Nombre del método que ha generado el nodo.

- **Valor**: Valor devuelto por la función o el tipo de excepción generada. El valor almacenado puede ser un tipo simple, un objeto o una estructura de datos.

- **Padre**: Referencia al nodo padre. Es de gran utilidad a la hora de realizar los recorridos de depuración.

- **Hijos**: Lista con todos los nodos hijos. Es interesante hacer notar que usamos árboles generales, en el que cada nodo puede tener n hijos.

- **nNodos**: Número de nodos que cumplen ciertos requisitos a la hora de realizar los diferentes recorridos de depuración. Aunque de inicio se le otorga un valor por defecto, este se recalcula una vez ha terminado de crearse el árbol.

- **Estado**: Almacena el estado actual del nodo. Cuando un nodo se crea, este recibe el estado indefinido ya que todavía no ha llegado a ser evaluado por el usuario. Los tipos de estado y sus características serán explicados en la sección de la memoria donde se explican los recorridos en detalle.

- **paramsEntrada**: Parámetros que recibe la función. Pueden ser de cualquier tipo.

- **paramsModificados**: Parámetros que han sido modificados a causa de los efectos laterales en el transcurso de la ejecución de una función. Pueden ser de cualquier tipo.

Teniendo en cuenta esta información, el constructor queda como sigue:

```
  def __init__(self):
        global contador
        self.id = contador
        contador += 1
        self.nFuncion = None
        self.valor = None
        self.padre = None
        self.hijos = []
        self.nNodos = 1
        self.estado = Estado.INDEFINIDO
        self.paramsEntrada = []
        self.paramsModificados = []
```

Además del constructor, la clase **Nodo** dispone de otra serie de métodos privados encargados de asignar bien las relaciones padre e hijo entre nodos así como otros de utilidad a la hora de llevar a cabo los recorridos cuando se está depurando código. Los métodos del primer tipo no se han explicado debido a su simplicidad, aunque si se desean ver están disponibles en el repositorio del proyecto. Los del segundo tipo, sin embargo, se explicarán en el apartado Recorridos de la memoria y en la transformación del árbol

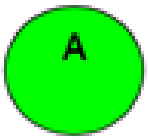
Ejemplo de construcción

Supongamos un programa formado por las siguientes funciones.

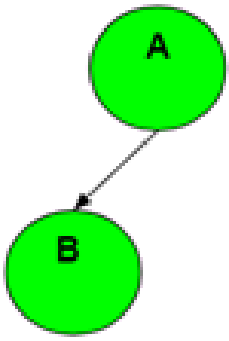
```
def A():
        return B() + C()
def B():
       return 5
def C():
       return 7
```

Como se puede observar, este código se encarga de llevar una sencilla suma donde los números son devueltos por otras dos funciones. La simpleza de este código es perfecta para poder abstraer los tipos de datos y el contenido de los métodos involucrados para que el lector pueda centrarse en comprender cómo funciona el sistema de captura de trazas para su posterior inserción en el árbol.

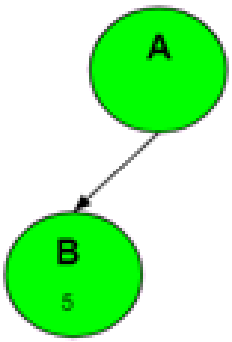
1. Cuando se ejecuta el método A, se inserta un primer nodo en nuestro árbol con el mismo nombre de la función.



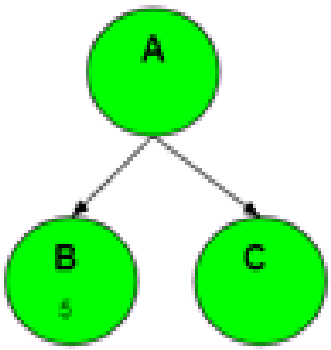
2. El flujo del programa entra en A, y esta hace una llamada al método *B*. Este nodo se almacena como un hijo del nodo A.



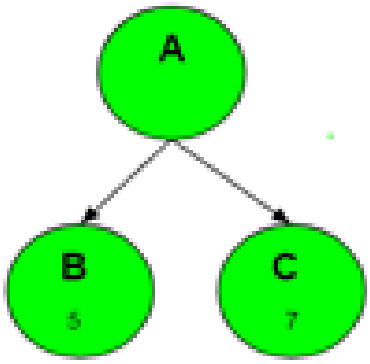
3. El método B se ejecuta y procede a retornar el valor 5, que se almacena en el nodo *B* creado en el punto anterior.



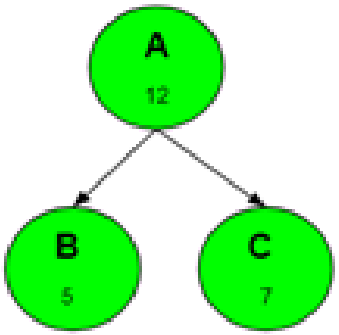
4. Una vez *B* ha terminado de ejecutarse, se llama al método C. Este crea otro nodo y se almacena como otro hijo de A, ya que como este evento se ha llamado posteriormente al evento de retorno del método B, el sistema considera B como un nodo hoja y por tanto solo puede ser almacenado en el padre de este.



5. El método C se ejecuta y retorna otro valor. Este valor se almacena en el nodo creado en el punto anterior.



6. Por último, A realiza su retorno correspondiente y lo almacena en su nodo.



Ejemplo de construcción con excepciones

Supongamos un programa compuesto por las siguientes funciones.

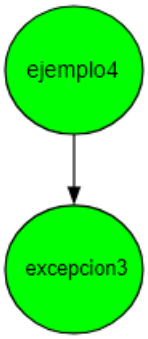
```
def excepcion1():
    x = 1/0

def excepcion2():
    lista = []
    lista.pop()
def excepcion3():
    excepcion1()
def ejemplo4():
    excepcion3()
    excepcion2()
```

Como se puede observar, en este programa se han introducido dos tipos de excepciones distintas, una producida por dividir un número entre cero y otra producida al intentar eliminar un elemento en una lista vacía, que serán ejecutadas en el método ejemplo4.

Al igual que en el ejemplo anterior, la simpleza de este código es perfecta para poder abstraer los tipos de datos y el contenido de los métodos involucrados para que el lector pueda centrarse en comprender cómo funciona el sistema de captura de trazas para su posterior inserción en el árbol cuando el programa debe finalizar su ejecución de forma abrupta.

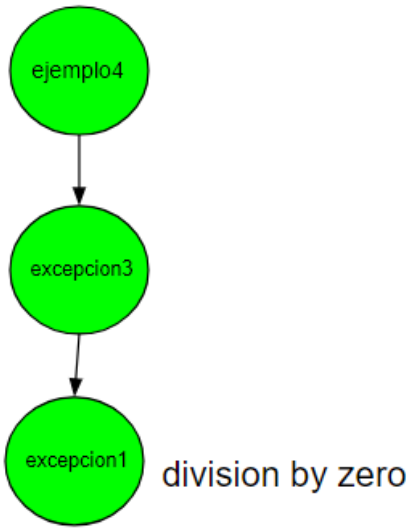
1. Cuando se llama al método ejemplo4, se inserta un primer nodo en el árbol con el mismo nombre de la función.

2. El flujo del programa entra en ejemplo4, y esta hace una llamada al método *excepcion3*. Este nodo se almacena como un hijo del nodo previamente creado.



3. Esta función a su vez llama al método excepcion1, creando un nuevo nodo y almacenándose como hijo del nodo que previamente ha sido insertado.

4. Cuando excepcion1 se ejecuta, el programa detecta la excepción y se captura la traza del fallo devuelto por este. En este caso se trata de una división entre cero que se almacena en el atributo **Valor** del **Nodo**.



Una vez se ha almacenado la excepción en el nodo correspondiente finaliza la ejecución del programa que estaba siendo depurado, dejando fuera del análisis el método excepcion2 de cara a la fase de recorrer el árbol generado por parte del usuario para evaluar todos sus nodos manualmente.

Transformación del árbol

En esta parte de la memoria se va a explicar una técnica centrada en reducir el tamaño del árbol de depuración una vez ha sido construido así como un ejemplo real de uso explicado paso a paso.

Fusión de nodos

La fusión de nodos es un arma de doble filo en el proceso de depuración. Eliminar nodos es útil de cara a ahorrar tiempo al programador durante el proceso recorrer todo el árbol, pero esto no siempre asegura que se reduzca el número de preguntas a contestar al obtener un árbol más plano. Esto se debe principalmente a que dicho nodo puede tener muchos más hijos a evaluar que su padre previamente fusionado si este último no hubiera sido marcado con el estado **error**.

Sabiendo los pros y los contras de este proceso, se llegó a la conclusión de que este debía ser de carácter opcional, por lo que puede ser activado o no por el usuario al iniciar el depurador. La forma en la que esta opción puede ser activada será explicada en el apéndice de la memoria “instalación y uso”.

La clase encargada de realizar este proceso es la propia clase Nodo.

```
def fusionNodos(self):
        if len(self.hijos) != 0:
            for i in self.hijos:
                if i.getNombre() == self.getNombre():
                    self.cambiarNodo(i)
                    self.fusionNodos()
                else:
                    i.fusionNodos()

def cambiarNodo(self, nodo):
      self.padre = nodo.padre
      self.nFuncion = deepcopy(nodo.nFuncion)
      self.valor = deepcopy(nodo.valor)
      self.nNodos = deepcopy(nodo.nNodos)
      self.estado = deepcopy(nodo.estado)
      self.paramsEntrada = deepcopy(nodo.paramsEntrada)
      self.paramsModificados = deepcopy(nodo.paramsModificados)
      self.hijos = deepcopy(nodo.hijos)
```

Tal y como se puede apreciar, el proceso de fusión de nodos consta de dos partes, implementada cada una en una función.

- **fusionNodos(self)**: Es llamada una sola vez por el árbol que ha sido generado previamente antes de comenzar el recorrido de este para su depuración. Se trata de una función recursiva que recorre los hijos de cada nodo buscando alguno que tenga el mismo nombre de función que el padre. En caso de que un nodo tenga el mismo nombre que su padre, este será quien invoque nuevamente a la función de forma recursiva. Si por el contrario el hijo se llama igual que el padre, este vuelca todos sus datos en el nodo padre y realizar otra llamada recursiva a la función.

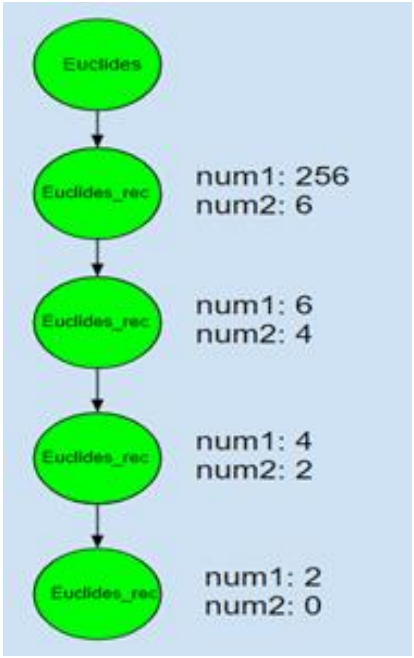
- **cambiarNodo(self, nodo)**: Es el encargado de recoger todos los datos de un nodo hijo y almacenarlos en su padre. De esta manera, su padre desaparece del árbol y se elimina un nivel de este.

Ejemplo de uso

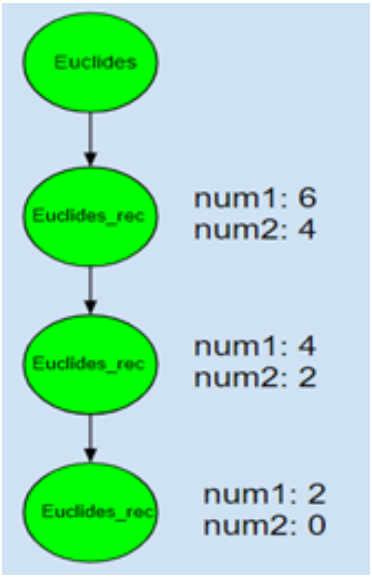
Supongamos el siguiente código de ejemplo, el cual se corresponde al algoritmo de Euclides para comprobar el máximo común divisor entre dos números.

```
def euclides():
    mcd = euclides_rec(256,6)
    return mcd
def euclides_rec(num1,num2):
    if num2 == 0:
        return num1
    return euclides_rec(num2, num1 % num2)
```

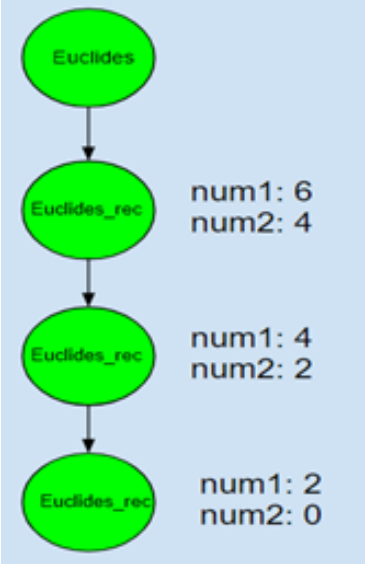
Una vez este código ha pasado por el proceso de construcción, el árbol resultante es el siguiente.



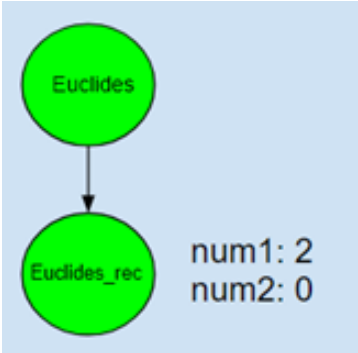
El proceso de fusión se inicia. El nodo *Euclides* evalúa a su único hijo. Como este no se llama igual que el padre, se realiza la llamada al proceso de fusión por parte del hijo. El nodo *Euclides_rec* evalúa a su hijo y detecta que es igual que él, por lo que realiza el cambio.



Una vez el cambio se ha producido, el nodo que ahora ha pasado a ser el padre realiza la recursión. Este detecta que su hijo también es igual que él y se procede otra vez realizar el cambio de nodos. El estado del árbol en este momento es el siguiente.



El proceso vuelve a repetirse, y el árbol acaba por tener la siguiente forma.



Como se puede apreciar, el resultado final de cara al usuario para la etapa de recorrido es bastante más agradable al haber eliminado todos los nodos intermedios. En este caso, la transformación del árbol se ha realizado de manera satisfactoria.

Recorrido del árbol de ejecución

Antes de hablar del recorrido vamos a ampliar un poco la información que tenemos sobre los nodos. Uno de los parámetros fundamentales de un nodo, tal y como se explicó en la sección correspondiente de la memoria, se trata del estado, que puede tomar los valores:

- **Indefinido:** Estado predeterminado de un nodo cuando es insertado en el árbol en su etapa de construcción. Significa que el nodo todavía no ha sido evaluado por el programador.

- **Válido:** El nodo es correcto con los parámetros de entrada y salida que han sido almacenados en su interior.

- **Error:** El resultado devuelto por la función del nodo no es el que debería, es decir, los valores almacenados en el nodo no son los que el programador esperaba.

- **Confiar:** Cuando el programador sabe que una función es correcta independientemente de los parámetros de la misma. Es una evolución del estado válido, ya que además de provocar el efecto producido por este, se propagara por todo el árbol para marcar todos los nodos con esta función a dicho estado.

- **Inaceptable:** la función no es un error *per se*, sino que el resultado es incorrecto debido a una situación que no deberá haber ocurrido (por ejemplo llamar a una función de ordenación sobre un array vacío).

- **Desconocido:** por diversos motivos es posible que el programador no esté seguro si una función está retornando los valores esperados. Para estos casos se utiliza el estado desconocido, ya que el programador no es capaz de dar una respuesta sobre el resultado de la función.

Hemos implementado estos estados definiendo una clase enumerado con la siguiente estructura:

```
class Estado(Enum):
    INDEFINIDO = 1
    VALIDO = 2
    ERROR = 3
    CONFIAR = 4
    INACEPTABLE = 5
    DESCONOCIDO = 6
```

Recorrido

Una vez generado el árbol el siguiente paso es recorrerlo, pero aquí nos encontrábamos con que teníamos múltiples estrategias y parámetros de dicho recorrido que podían variar en una misma sesión de depuración. Por lo tanto después de plantear varias posibilidades apostamos por la estructura que vamos a presentar a continuación.

Estructura del Recorrido

A la hora de gestionar el recorrido del árbol implementamos una clase de nombre **Recorrido**, cuyos atributos son los siguientes.

- **arbol:** El árbol de ejecución que el depurador va a recorrer.

- **estrategia:** La estrategia actual del recorrido, que puede variar después de cada pregunta, en nuestro caso inicializamos a *Top Down*, unos de los tipos de estrategia que definimos más adelante en la sección de estrategias.

- **arbol.estado:** Se inicializa a **error** ya que entendemos que al ejecutar el depurador sobre una función es porque se ha detectado un error en ella, por tanto el nodo padre sería el primer nodo erróneo encontrado.

- **desconocidos:** Es un array de nodos donde guardamos aquellos a los que el programador responde con “desconocido” para su posterior revisión en caso de no encontrar un error al finalizar el recorrido.

- **ended:** Marcador de comprobación que indica si el recorrido ha finalizado.

- **buggy:** Marcador de comprobación que indica si se ha encontrado un error durante el recorrido.

- **nodoBuggy:** Nodo que el depurador ha detectado como erróneo y que se le notificará al programador.

Hemos implementado la clase **Recorrido** utilizando los siguientes parámetros:

```
def __init__(self, tree):
        self.arbol = tree
        self.estrategia = Estrategia.TOPDOWN
        self.arbol.estado = Nodo.Estado.ERROR
        self.desconocidos = []
        self.ended = False
        self.buggy = False
        self.buggyDN = False
        self.nodoBuggy = Nodo.Nodo()
```

Cada vez que se ejecuta el depurador este genera una instancia de la clase Recorrido, la cual avanza a través del árbol de ejecución en función de la estrategia establecida siguiendo el siguiente esquema:

1. *Busca el siguiente nodo(según la estrategia)*
2. *Pregunta al programador por el nodo*
3. *Analiza la respuesta del programador y actúa en consecuencia*
4. *Busca el siguiente nodo.*

Este esquema se ejecuta en bucle hasta que, o bien encuentra un error, o bien termina de recorrer el árbol de ejecución. En el primer caso, notifica al programador y se detiene, en el segundo plantea al programador dos escenarios, puede detener el depurador o intentar reevaluar los nodos con estado desconocido e intentar encontrar un error entre ellos.

¿Qué nodos son correctos y qué nodos incorrectos?

Un nodo incorrecto es todo aquel que tenga como estado **error**, los nodos correctos son todos aquellos que tienen como estado Válido, Confiar o Inaceptable. Inaceptable se toma como correcto ya que la función en sí no es un error, sino la manera en la que se invoca.

¿Y el estado Desconocido?

El estado desconocido es un caso especial, como no sabemos si es correcto o incorrecto los pasos a seguir pueden ser muy diversos. Ya que no podemos definir con exactitud lo que son no podemos afirmar si son o no son *buggy*, por lo que se decidió aplicar el siguiente esquema.

Cuando encontramos un nodo con estado Desconocido lo almacenamos, en lugar de eliminar del recorrido los hijos de ese nodo como haríamos con un estado correcto, se avanza al siguiente nodo según la estrategia definida. Si una vez finalizado el recorrido no se ha encontrado un nodo *buggy*, se le da la opción al programador de recorrer estos nodos para volver a evaluarlos como se ha mencionado anteriormente.

Estrategias de Recorrido

Una vez explicado en secciones previas qué es un **estado**, qué tipos hay y cuáles son sus características, en este punto definiremos con detalle las estrategias que hemos utilizado aportando fragmentos de código para ver su funcionamiento.

TOP DOWN

La estrategia *Top Down* consiste en recorrer el árbol avanzando del nodo padre a los nodos hijos en orden descendente (de arriba a abajo).

Consta de un bucle que recorre los hijos sumando los correctos, profundizando en los incorrectos, ignorando los desconocidos y preguntando y analizando los que aún no han sido contestados.

```
validos = 0
for i in nodo.hijos:
      if i.estado == Nodo.Estado.VALIDO or i.estado == Nodo.Estado.CONFIAR
      or i.estado == Nodo.Estado.INACEPTABLE:
            validos = validos+1
      elif i.estado == Nodo.Estado.ERROR:
            self.topDown(i)
            break
      elif i.estado == Nodo.Estado.INDEFINIDO:
            self.ask(i)
            if i.estado == Nodo.Estado.DESCONOCIDO:
                  self.desconocidos.append(i)
            elif(i.estado == Nodo.Estado.ERROR):
                  break
            elif i.estado == Nodo.Estado.VALIDO
            or i.estado == Nodo.Estado.CONFIAR
            or i.estado == Nodo.Estado.INACEPTABLE:
                        validos = validos+1
```

El siguiente paso es analizar los resultados del bucle en función de si el nodo que estamos recorriendo es desconocido o no y si la suma de sus correctos son igual a la suma de sus hijos.

```
if len(nodo.hijos) == validos and nodo.estado != Nodo.Estado.DESCONOCIDO:
      self.nodoBuggy = nodo
      self.buggy = True
      self.buggyMsj()
elif len(nodo.hijos) == validos and nodo.estado==Nodo.Estado.DESCONOCIDO:
      if self.arbol.nNodos == 1:
            self.nodoBuggy = nodo
            self.buggy = True
            self.buggyMsj()
      else:
            for i in nodo.hijos:
                  if i.estado == Nodo.Estado.DESCONOCIDO:
                        self.topDown(i)
                  self.buggy = True
```

HEAVIEST FIRST

La estrategia *Heaviest First* (los más pesados primero) consiste en recorrer los hijos empezando siempre por el más pesado de ellos (el que más nodos indefinidos posee), lo que implica que tanto si encontramos un error como si no, una parte importante del árbol es ignorada y no será necesario recorrerla.

Consta de un bucle que recorre los hijos con estado indefinido y almacena su peso en un array que se utilizará en la lógica para escoger el siguiente hijo a analizar.

```
descendientes = []
validos = 0
for i in nodo.hijos:
      if i.estado == Nodo.Estado.INDEFINIDO:
            descendientes.append(i.nNodos)

      elif i.estado == Nodo.Estado.VALIDO
      or i.estado == Nodo.Estado.CONFIAR
      or i.estado == Nodo.Estado.INACEPTABLE:
            validos = validos+1
```

A continuación se analizan los resultados, en caso de contener hijos indefinidos, se ejecuta un bucle que recorre nuevamente los hijos buscando el de mayor peso, y pregunta por este. Cuando vuelve, invoca de nuevo a la función sobre el padre para seguir recorriendo los hijos que pudieran quedar. En caso de no haber hijos indefinidos se analiza si nos encontramos ante un *buggy* o si podemos seguir profundizando.
Si el nodo tiene todos sus hijos correctos y no es desconocido la única posibilidad es que sea un error y por lo tanto esto lo convierte en un *nodo buggy.*

```
if len(nodo.hijos) == validos and nodo.estado != Nodo.Estado.DESCONOCIDO:
      self.nodoBuggy = nodo
       self.buggy = True
       self.buggyMsj()
```

Por otro lado si el estado es desconocido, como no podemos asegurar si es un *nodo buggy*, comprobamos primero que no queden nodos con estado indefinido y a continuación lo asignamos como posible *nodo buggy* pero sin confirmación explicita.

```
elif len(nodo.hijos) == validos and nodo.estado==Nodo.Estado.DESCONOCIDO:
      if self.arbol.nNodos == 1:
            self.nodoBuggy = nodo
            self.buggy = True
            self.buggyMsj()
```

Si el array descendientes no está vacío significa que todavía quedan hijos sin preguntar por lo que se vuelve a ejecutar el bucle y en caso de no encontrar error, se vuelve a llamar recursivamente al padre.

```
elif len(descendientes)!=0:
      found = False
      j=0
      while(found==False):
            if nodo.hijos[j].nNodos == max(descendientes)
            and nodo.hijos[j].estado == Nodo.Estado.INDEFINIDO:
                  found = True
                  self.ask(nodo.hijos[j])
                  if nodo.hijos[j].estado == Nodo.Estado.DESCONOCIDO:
                        self.desconocidos.append(nodo.hijos[j])

                  if(nodo.hijos[j].estado == Nodo.Estado.VALIDO
                  or nodo.hijos[j].estado == Nodo.Estado.CONFIAR
                  or nodo.hijos[j].estado == Nodo.Estado.INACEPTABLE
                  or nodo.hijos[j].estado == Nodo.Estado.DESCONOCIDO):
                        self.heaviestFirst(nodo)
                  else:
                        self.nodoBuggy = nodo
                        self.buggy = True
                        self.buggyMsj()
            j=j+1;
```

La última posibilidad es que todos los hijos hayan sido preguntados pero aún así no se haya encontrado un buggy, esto significa que uno o varios de los hijos tienen estado desconocido, por tanto el siguiente paso es profundizar en estos siguiendo el algoritmo correspondiente de esta estrategia.

```
else:
    descendientes = []
    for i in nodo.hijos:
      if i.estado == Nodo.Estado.DESCONOCIDO:
            descendientes.append(i.nNodos)
       found = False
       j=0
       while(found==False):
            if nodo.hijos[j].nNodos == max(descendientes)
            and nodo.hijos[j].estado == Nodo.Estado.DESCONOCIDO:
                  found = True
                  self.heaviestFirst(nodo.hijos[j])
                  if self.arbol.nNodos == 1:
                        self.nodoBuggy = nodo.hijos[j]
                        self.buggy = True
                        self.buggyMsj()
                    else:
                         print(self.arbol.nNodos)
                         wself.heaviestFirst(self.arbol)
              j=j+1;
```

HALF DOWN

La estrategia *Half Down* es similar a la anterior pero cambiando la condición, en lugar de buscar el hijo más pesado busca el hijo cuyo número de nodos indefinidos sea aproximadamente la mitad de los nodos indefinidos del árbol (en caso de que ninguno tenga exactamente la mitad busca el más aproximado), de esta manera se ignora rápidamente la mitad del árbol. Esta búsqueda es muy efectiva si el árbol está equilibrado.

Esta estrategia es muy similar al *Heaviest First* solo que la condición es ligeramente diferente, por ello la estructura del bucle, de recorrido y de análisis es muy similar.

Como el código también es muy similar al expuesto en el punto anterior, en este no lo expondremos pero está disponible en el repositorio dónde se almacena el código, para más información el código completo está disponible en el repositorio.

Funciones adicionales

Por último en esta sección destacaremos una serie de funciones esenciales para nuestro depurador y que están directamente relacionadas con esta sección.

La primera función a destacar es el ask, encargada de hacer las preguntas al programador en el modo de consola, una de las partes fundamentales de este depurador.

Primero imprimimos la información del nodo que estamos preguntando.

```
def ask(self, nodo):
      nb = ""
      while(nb!="s" and nb!="n" and nb!="c"  and nb!="i"  and nb!="d"):
            View.TreeView.show(self.arbol)
            print("Nombre de función:", nodo.getNombre())
            print("ID de la función:", nodo.id)
            print("El número de hijos de '"+nodo.getNombre()+"' es:",
                  nodo.nNodos)
            print("El estado de '"+nodo.getNombre()+"' es:", nodo.estado)
            print("Valor retornado por '"+nodo.getNombre()+"' es:",
                  nodo.getValor(), '\n')
            nb = input("(Pulsa c/i/d --- c: confiar /
                  i: inaceptable / d: desconocido)\n
                  (Puedes cambiar de estrategia pulsando e)\n
                  ¿Es correcto?(s/n): ")
```

Si la respuesta es “e” significa que el usuario quiere cambiar de estrategia, por lo que ponemos a su disposición las estrategias de las que disponemos y recogemos a cuál quiere cambiar.

```
            if(nb == "e"):
                while(nb!="td" and nb!="hf" and nb!="dh"):
                    print("\nEstas son las estrategias disponibles:")
                    nb = input(" - td (TOPDOWN)\n
                   - hf (HEAVIESTFIRST)\n - dh (HALFDOWN)\n
                   Selecciona tu estrategia:")

                    if(nb == "td"):
                        self.estrategia = Estrategia.TOPDOWN
                        self.topDown(nodo.padre)

                    elif(nb == "hf"):
                        self.estrategia = Estrategia.HEAVIESTFIRST
                        self.heaviestFirst(nodo.padre)

                    elif(nb == "dh"):
                        self.estrategia = Estrategia.HALFDOWN
                        self.halfDown(nodo.padre)

                    else:
                        print("\nERROR: '" + nb +
                        "' no se corresponde con ninguna de las
                  estrategias.")
```

Una vez el programador ha escogido su estrategia y ha decidido que responder a la pregunta se recoge su respuesta y en función de esta se recorren los nodos asignando su correspondiente estado.

```
            if(nb == "s"):
                 nodo.estado = Nodo.Estado.VALIDO
                 self.arbol.recorrerNodos(nodo);
             elif(nb == "c"):
                 nodo.estado = Nodo.Estado.CONFIAR
                 self.arbol.recorrerNodos(nodo);
             elif(nb == "i"):
                 nodo.estado = Nodo.Estado.INACEPTABLE
                 self.arbol.recorrerNodos(nodo);
             elif(nb == "d"):
                 nodo.estado = Nodo.Estado.DESCONOCIDO
                 self.arbol.recorrerNodos(nodo);
```

Además si el estado es erróneo se llama recursivamente al nodo con la estrategia activa.

```
             elif(nb == "n"):
                 nodo.estado = Nodo.Estado.ERROR
                 self.arbol.recorrerNodos(nodo);
                 if(self.estrategia == Estrategia.TOPDOWN):
                     self.topDown(nodo)
                 if(self.estrategia == Estrategia.HEAVIESTFIRST):
                     self.heaviestFirst(nodo)
                 if(self.estrategia == Estrategia.HALFDOWN):
                     self.halfDown(nodo)
             else:
                 print("\nERROR: el comando '" + nb
                    + "' no se corresponde con ninguna de las opciones
             disponibles.")
```

Otra de las funciones principales es la función de **recorrerNodos**, encargada de recorrer el árbol después de cada pregunta actualizando la información tanto del recorrido como del estado de los nodos.

```
def recorrerNodos(self, nodoBusqueda):
        ret = 0
```

Si la respuesta a la función ask fue confiar, entonces solo comprobamos el nombre de la función y actualizamos el estado.

```
       if nodoBusqueda.estado == Estado.CONFIAR
        and nodoBusqueda.getNombre() == self.getNombre():
             self.estado = Estado.CONFIAR
```

Si por el contrario la respuesta fue otra analizamos todos los parámetros, tanto el nombre como la entrada y salida, efectos laterales incluidos.

```
elif len(nodoBusqueda.paramsEntrada) == len(self.paramsEntrada)
and nodoBusqueda.getValor() == self.getValor()
and nodoBusqueda.getNombre() == self.getNombre():
      ok = True
      if len(nodoBusqueda.paramsEntrada) > 0 :
          for clave in nodoBusqueda.paramsEntrada:
            self.comprobarParams(ok, clave, self.paramsEntrada,
                  nodoBusqueda.paramsEntrada)
            if ok:
                  self.comprobarParams(ok, clave,
                   self.paramsModificados,
                   nodoBusqueda.paramsModificados)
            if ok:
                  self.estado = nodoBusqueda.estado
            if self.estado == Estado.ERROR
            or self.estado == Estado.INDEFINIDO:
                  ret = 1
```

Si el nodo no es una hoja continuamos recursivamente la búsqueda hasta finalizar todos los nodos. Esto también actualiza la profundidad del árbol.

```
  if len(self.hijos) != 0
      and self.estado != Estado.VALIDO
      and self.estado != Estado.CONFIAR
      and self.estado != Estado.INACEPTABLE:
      for i in self.hijos:
          ret += i.recorrerNodos(nodoBusqueda)

self.nNodos = ret
return ret
```

Otra de las funciones utilizadas para interactuar con el usuario es la función **buggyMsj**, esta función es invocada cuando el recorrido finaliza, también es la encargada de invocar a la función de revisar los desconocidos en caso de tener nodos con este estado.

```
def buggyMsj(self):
```

Si el nodo que encuentra como buggy tiene estado desconocido significa que no ha podido encontrar un *buggy* de forma totalmente segura por lo que indica una alta probabilidad y pregunta al programador si quiere revisar los nodos desconocidos.

```
if self.nodoBuggy.estado == Nodo.Estado.DESCONOCIDO:
      print("Hay una alta posibilidad de que la función buggy sea:",
      self.nodoBuggy.getNombre())
      print("Su valor retornado es:", self.nodoBuggy.getValor())
      View.TreeView.show(self.arbol)
      nb=""
```

```
while(nb!="y" and nb!="n"):
      nb = input("¿Quieres revisar los nodos desconocidos?(s/n)")
      print(nb)
      if(nb == "s"):
            while(len(self.desconocidos) > 0):
                  self.revisarDK()
      elif(nb == "n"):
            sys.exit()
```

En cualquier otro caso significa que ha encontrado un buggy y por tanto indica al programador dónde se encuentra para que pueda iniciar la depuración.

```
 else:
      print("Buggy en la función:", self.nodoBuggy.getNombre())
      print("Valor retornado es:", self.nodoBuggy.getValor())
      View.TreeView.show(self.arbol)
      sys.exit()
```

Para terminar, la última función que vamos a explicar en este apartado, y también la última del flujo de ejecución del programa es la función **revisarDK,** función encargada de recorrer los nodos con estado desconocido, volviendo a hacer preguntas al usuario para intentar encontrar un *buggy* que pasase desapercibido en el primer recorrido.

```
def revisarDK(self):
      self.buggy = False
       j = 0
```

Debido a que en la lista de desconocidos se almacenan los nodos según se van encontrando esta comprobación sirve para asegurarse de que no hemos resuelto un nodo previo como correcto (lo que implica que este es correcto también), se cambia el estado y se elimina el nodo del array.

```
  for i in self.desconocidos:
      if i.estado == Nodo.Estado.DESCONOCIDO:
          if i.padre.estado == Nodo.Estado.VALIDO
            or i.padre.estado == Nodo.Estado.CONFIAR
            or i.padre.estado == Nodo.Estado.INACEPTABLE:
              i.estado = i.padre.estado
              self.desconocidos.pop(j)
```

En caso de que el nodo padre sea error o desconocido se vuelve a preguntar por este nodo y se elimina del array, a continuación se ejecuta sobre el árbol la función de recorrido para comprobar si gracias a esta respuesta se puede encontrar un *buggy.*

```
else :
      self.ask(i)
      if i.estado ==Nodo.Estado.VALIDO
      or i.estado==Nodo.Estado.CONFIAR
      or i.estado == Nodo.Estado.INACEPTABLE:
            self.desconocidos.pop(j)
            if(self.estrategia == Estrategia.TOPDOWN):
            self.topDown(self.arbol)
      if(self.estrategia == Estrategia.HEAVIESTFIRST):
            self.heaviestFirst(self.arbol)
      if(self.estrategia == Estrategia.HALFDOWN):
            self.halfDown(self.arbol)
      j = j+1
```

Interfaces

En este apartado de la memoria se presenta cómo se implementó tanto la interfaz de consola como la posterior interfaz gráfica.

Interfaz de consola

La interfaz de consola consta de dos partes, el Módulo *View* y la función **ask**:

Módulo View

El módulo *View* es el encargado de imprimir todos y cada uno de los nodos del árbol formado por las llamadas a funciones del programa. Es una clase que cuenta únicamente con métodos estáticos, lo que facilita la reutilización de los mismos sin tener que instanciar un objeto *View* que guarde una copia o referencia del árbol en cuestión.

Cuenta con dos métodos estáticos:

- **show(**node**):** Este método tiene la función de ser un lanzador de lo que es el método que de verdad se encarga de hacer la impresión por pantalla del árbol.

```
class TreeView():

    @staticmethod
    def show(nodo):
        colorInit()
        TreeView.recursiveShow(nodo, 0)
```

Llama al siguiente método:

- **recursiveShow(**node, nivel**):** Este método hace uso de una biblioteca de Python llamada Colorama, que permite un manejo mucho más sencillo e intuitivo de los colores la consola, en comparación con la forma nativa del lenguaje.

El código de colores que hemos implementado es el siguiente:

- **Blanco:** color para los nodos con estado INDEFINIDO, los cuales no han sido establecidos todavía por el usuario
- **Rojo:** color para los nodos con estado ERROR, los cuales han sido marcados como incorrectos según el criterio del usuario.
- **Verde:** color para los nodos con estado VALIDO, no tienen ningún error.
- **Azul:** color para los nodos con estado CONFIAR, los cuales se han marcado así porque se tiene la total certeza de que son correctos siempre.
- **Amarillo:** color para los nodos con estado INACEPTABLE, los cuales se han marcado así porque el usuario considera que los parámetros son erróneos, o que su valor de retorno es inadmisible.
- **Magenta:** color para los nodos con estado DESCONOCIDO, de los cuales el usuario desconoce su corrección o incorrección.

recursiveShow es un método recursivo que utiliza el argumento *nivel* para llevar constancia de cuál es el nivel de profundidad de *node* con respecto a la raíz del árbol. Este argumento también se utiliza para aplicar sangría a los hijos con respecto a su padre

La información que se muestra mediante este método es la siguiente: el nombre de la función, los parámetros de entrada y los de salida y, por último, el valor de retorno de la función, que, en caso de haberse lanzado una excepción, ése será su valor de retorno.

Ejemplo consola

```
ejemplo1 Entrada: ( {} ) Salida: ( {} ) --> 30
        h Entrada: ( {'param2': 'hola', 'param1': 'hello'} ) Salida: ( {'param2': 'hola', 'param1': 'hello'} ) --> hello
        c Entrada: ( {} ) Salida: ( {} ) --> 25
                b Entrada: ( {} ) Salida: ( {} ) --> 8
                        a Entrada: ( {} ) Salida: ( {} ) --> 6
                                f Entrada: ( {} ) Salida: ( {} ) --> None
                b Entrada: ( {} ) Salida: ( {} ) --> 8
                        a Entrada: ( {} ) Salida: ( {} ) --> 6
                                f Entrada: ( {} ) Salida: ( {} ) --> None
                a Entrada: ( {} ) Salida: ( {} ) --> 6
                        f Entrada: ( {} ) Salida: ( {} ) --> None
```

Este ejemplo hace referencia a uno de los múltiples códigos de ejemplo disponibles en el depurador para comprobar su uso y que se puede encontrar en el módulo *Utils* del proyecto para su ejecución.

Función ask

La función *ask* se encarga de manejar las respuestas del usuario respecto al nodo dado.

Lo primero que realiza esta función es la llamada al método *show* del módulo *View* para mostrar el árbol antes de realizar las preguntas de modo que estas sean más sencillas de responder y, sobre todo, más visuales.

A continuación, se expone la información necesaria para el usuario para poder elegir una respuesta y establecer un estado en el nodo actual. La información que se le indica es la siguiente: el nombre de la función, el ID, el número de hijos, el estado actual del nodo y el valor de retorno de la función. Después de esto, se le indica las opciones que posee sobre el nodo o sobre la estrategia a seguir para recorrer el árbol de llamadas.

En el caso de que se decida cambiar la estrategia entre las que estén disponibles, entraría en un nuevo bucle que no cesa hasta que la nueva estrategia es escogida. Estas estrategias están definidas con un código de dos caracteres que son lo suficientemente intuitivos para saber siempre cuál es la estrategia escogida.

- ***td:*** *Top-Down.*
- ***hf:*** *Heaviest First.*
- ***hd:*** *Half Down*

Una vez el usuario tiene decidida cuál va a ser su respuesta puede introducir un solo carácter, que coincidirá con la inicial de la respuesta en español.

Por último, tendríamos que hablar de la finalización del recorrido, cuando un nodo *buggy* es encontrado, implementado por la función buggyMsj del módulo Recorridos. Una función que se encarga de mostrar al usuario cuál es el nodo *buggy* en caso de haberlo encontrado o, en su defecto el nodo con la probabilidad más alta de serlo.

Ejemplo consola

Interfaz gráfica

Aquí trataremos principalmente cuatro partes distintas: Controller, RecorridosGUI, Módulo principal y módulos secundarios.

Controller

Controller es el módulo que se encarga de hacer de mediador entre el modelo, la lógica, la función *main* y la interfaz. Es una clase que se encarga de gestionar muchos de los eventos que tienen lugar en tiempo de ejecución.

Posee varios atributos entre los que se encuentran:

- ***tree***: atributo que se encarga de mantener una referencia al árbol de llamadas a funciones. En el constructor se le asigna el estado ERROR al nodo raíz.

- ***graphics***: booleano que se obtiene de los parámetros del *main*, indica si es necesario utilizar la interfaz de consola o la gráfica.

- ***estrategia***: El controlador guarda en todo momento una instancia de la estrategia que está siendo utilizada.

- ***revisarDK***: Es un booleano utilizado exclusivamente en la interfaz gráfica para llevar a cabo la rutina de tratamiento de nodos DESCONOCIDOS.

run

Encargado de ejecutar una de las dos interfaces basándose en el atributo *graphics.*

answerGUI(node, answer)

Se ejecuta al pulsar sobre alguno de los botones de respuesta de los nodos y es llamado por el controlador de acción de estos mismos botones. Se encarga de establecer el estado del nodo *node* en función de la respuesta obtenida en *answer*, actualizar tanto el árbol como la interfaz con el cambio que ha producido esa respuesta, volver a llamar al recorrido de la estrategia escogida y gestionar los nodos DESCONOCIDOS.

handleDK(boolean)

Es el método encargado de hacer la gestión de estados desconocidos ya mencionada en la función answerGUI. En el caso de que *boolean* sea true, se pasará a revisar los desconocidos existentes y, en caso contrario se obtendrán los nodos buggy (error y desconocido) y se mostrarán en la interfaz.

swapStrategy(strategy)

Método setter, encargado de establecer la estrategia del controlador a la dada por el parámetro strategy. Este método es llamado por el controlador de acción del Spinner de selección de estrategia.

RecorridosGUI

Es un módulo que simula el comportamiento de las estrategias para recorrer el árbol existentes en el módulo Recorridos. La lógica a alto nivel es prácticamente la misma, pero los recorridos están adaptados a las exigencias de la interfaz, ya que no se puede pausar la ejecución de un recorrido para esperar una respuesta de la interfaz, si no que se debe devolver un valor para que la interfaz se actualice en relación a ese valor.

Sus atributos son:

- **dk**: es un atributo booleano cuyo valor se corresponde con la existencia de un posible nodo buggy en alguno de los nodos con estado desconocido.

- **buggy**: atributo booleano que, en caso de ser true, significa que un nodo buggy con estado error ha sido encontrado.

- **buggyNode**: es una referencia al último nodo con estado error que ha sido encontrado o, en caso de haber sido modificado por la función `getDeepestError`, al nodo que ha provocado el fallo en el programa.

- **desconocidos**: es una lista de nodos cuyo estado es desconocido, cualquier nodo que en la interfaz se marque como desconocido será añadido a esta lista para futura gestión con la función `revisarDK`.

- **buggyDKNode**: es una referencia al nodo con estado desconocido que, posiblemente, ha provocado el fallo en el programa. Su valor es actualizado mediante la función `getDeepestDK`.

Como se ha explicado anteriormente, el manejo de nodos desconocidos en la interfaz gráfica es distinto al de la interfaz de consola. En lugar de pasar al siguiente hermano indefinido en caso de marcar un nodo como desconocido, se profundizará en la rama correspondiente a este nodo, por alguno de sus hijos si es que los posee.

topDown(node)

Función que simula el comportamiento de la estrategia `Recorridos.topDown(node)`, el funcionamiento es prácticamente el mismo, pero en lugar de realizar la llamada a la función `ask`, realiza el retorno del siguiente nodo que necesita una actualización de su estado.

heaviestFirst(node)

Simula el comportamiento del recorrido `Recorridos.heaviestFirst(node)`, el funcionamiento es prácticamente el mismo, con excepción de la llamada a la función `ask`, que se sustituye por el retorno del siguiente nodo que requiere una actualización de estado.

halfDown(node)

Como seguramente ya se haya explicado antes, la estrategia Half Down es muy parecida a Heaviest First, la única diferencia es el criterio que se usa para escoger un hijo del nodo en lugar de otro. Su comportamiento es similar al recorrido `Recorridos.halfDown(node)`, con la excepción de la llamada al método `ask`.

revisarDK

Es el método encargado del tratamiento de errores después de recorrer el árbol por primera vez. Revisa todos los nodos desconocidos incluidos en el atributo de clase *desconocidos* comprobando si su padre tiene un nodo con el que se pueda profundizar o no, en caso negativo, hereda su estado para asegurar que, si tiene hijos con estado desconocido, tampoco se recorran.

getDeepestError(node)

Obtiene el error más profundo (si existe) a partir de un nodo dado, y se lo asigna al atributo de clase buggyNode.

getDeepestDK(node)

Obtiene el nodo con estado desconocido más profundo (si existe) a partir de un nodo dado, y se lo asigna al atributo de clase buggyDKNode.

Módulo principal (Interface)

Este módulo está formado por dos archivos, Interface.py e Interface.kv, que son los encargados de construir la interfaz a partir de la información obtenida por el controlador y utilizando los elementos de los módulos secundarios (explicados más adelante).

Interface.py

Este archivo es el contenedor de toda la construcción de la interfaz, es donde todos los widgets son creados a partir de otras clases y añadidos a la aplicación. Aquí ha sido necesario utilizar distintas variables globales, ya que, intentar integrarlas en una de las clases existentes en este archivo sería complicado y contraproducente.

- **tree**: es una variable global que almacena el árbol formado por las llamadas a funciones del programa, se obtiene a partir de la función *initGUI*, que se llama desde *Controller*, pasándole una referencia del mismo árbol que posee.

- **controller**: es una variable global que almacena la referencia al *Controller* usado para comunicar la interfaz con la lógica, también se obtiene en la función *initGUI*.

- **tv**: contiene un widget de tipo *TreeView*, utilizado para contener el árbol de la interfaz, que a su vez es construído a partir de *tree*.

- **menu**: contiene un widget personalizado, necesario para permitir al usuario comenzar la depuración, salir del depurador, cambiar la estrategia o visualizar el error y/o posible error.

- **popup**: contiene un widget popup personalizado, necesario para solicitar al usuario confirmación sobre su deseo de volver a repasar los nodos con estado desconocido.

populate_tree_view(parent, node)

Permite rellenar el *TreeView widget tv* a partir de nodos CustomTreeNode explicados más adelante. Es una función recursiva que crea una representación visual del árbol contenido en *tree*.

Clase MainFrame

Es una clase que hereda del módulo *BoxLayout* de Kivy. La instancia creada de esta clase es la encargada de contener todos los widgets principales que forman la interfaz (*tv* y *menu*). Se encarga también de añadir el widget *tv* a un ScrollView para permitir al usuario desplazarse libremente por el árbol, y de llamar a populate_tree_view para rellenar *tv* de nodos.

Clase InterfaceApp

Es la clase principal, que hereda de App, módulo que utiliza Kivy para definir la instancia de esta clase como la aplicación principal desde la que se construirá el resto de la interfaz. Su función *build* llamada desde Kivy, se encarga de construir una instancia de la clase *MainFrame*.

Set selected/unselected

Estas dos funciones, permiten a la interfaz resaltar un nodo del *TreeView tv* para ayudar al usuario a ver, con más claridad, el nodo al que se le tiene que asignar un estado. *setSelected(node)* recibe una referencia al nodo del árbol *tree*, que contiene el nodo actual a seleccionar, resaltando la representación visual del mismo y activando los botones de respuesta de este nodo. setUnselected(node) hace precisamente lo contrario, deselecciona el nodo y desactiva los botones del mismo.

Ambas funciones se ayudan de searchNode(node), función que busca en tv la representación visual correspondiente al nodo pasado por parámetro, que referencia el nodo actual de tree en el que se encuentra la ejecución.

updateNodes

Es una función que es llamada desde el controlador que permite actualizar el estado de todos y cada uno de los nodos de *tv*. Llama a su vez a una función propia de *CustomTreeNode* que se encarga de actualizar de forma individual el nodo que indica esta función. Además, en el caso de que el estado que se actualice, sea un estado con el que no se tiene que profundizar en la rama, se encarga de colapsar este nodo para no ensuciar la vista general del usuario sobre el árbol.

Popup

Nuestro popup personalizado se gestiona mediante dos funciones diferentes, llamadas askDK, y dismissDK. La primera de ellas se encarga de crear una instancia de nuestro popup personalizado DKPopup para luego añadirlo como widget a la variable global de tipo *Popup*, establecer la información que se mostrará en el popup y por último abrirlo, bloqueando la interacción del usuario con el resto de la interfaz hasta que se responda a la pregunta con los botones “Aceptar” o “Cancelar”.

dismissDK tiene una función sencilla, únicamente se encarga de cerrar este popup cuando la respuesta ha sido dada por el usuario. Es una función necesaria ya que se llama desde el controlador, que es quien posee la información sobre la respuesta.

showBuggyFunction(name, params, namedk, paramsdk)

Esta función se encarga de actualizar la etiqueta que informa al usuario de cuál es el nodo "buggy" y/o cuál es el nodo que tenga más probabilidades de serlo. Esta etiqueta se encuentra dentro del *widget menu*.

initGUI(arbol, controller)

Es la función inicializadora de la interfaz, asigna a las variables globales *tree* y *controller* los valores pasados por parámetro *arbol* y *controller,* respectivamente.

Inteface.kv

Este archivo está escrito en el lenguaje propio de Kivy, Kv. Lenguaje de marcado explicado anteriormente. Es utilizado para crear fácilmente nuestros *widgets* personalizados como CustomTreeNode, DKPopup y Menu. Todos estos *widgets* contienen ids en cada una de sus partes para facilitar la posterior modificación a petición de *Interface.py*.

CustomTreeNode

Es el *widget* personalizado que representa cada uno de los nodos del árbol de manera visual. La información que muestra es la siguiente: nombre de la función, parámetros de entrada, parámetros de salida, valor retornado, botones para seleccionar la respuesta y, por último, el estado del nodo.

```
<CustomTreeNode>:
  size: self.width, self.ids.labelName.height+self.ids.labelReturn.height
  BoxLayout:
    orientation: 'vertical'
    Label:
      size: self.texture_size[0], self.texture_size[1]
      text_size: self.width, None
      id: labelName
      text: 'Nombre funcion'
    Label:
      size: self.texture_size[0], self.texture_size[1]
      haling: 'left'
      text_size: self.width, None
      id: labelReturn
      text: 'Valor retornado: '
  BoxLayout:
    BoxLayout:
      orientation: 'vertical'
      Button:
        id: buttonYes
        text: 'V\u00e1lido'
```

```
        disabled: True
      Button:
        id: buttonNo
        text: 'Error'
        disabled: True
    BoxLayout:
      orientation: 'vertical'
      Button:
        id: buttonTrust
        text: "Confiar"
        disabled: True
      Button:
        id: buttonUnacceptable
        text: "Inaceptable"
        disabled: True
      Button:
        id: buttonDontKnow
        text: "Desconocido"
        disabled: True
  Label:
    id: labelState
    text: 'estado'
```

Menu

Se encarga de manejar las funciones generales del depurador, como comenzar la depuración, salir del depurador, cambiar de estrategia y, además, ofrece información como el nodo “buggy” o el que tiene más probabilidades de serlo.

```
<Menu>:
  canvas:
    Color:
      rgb: 0.4,0.4,0.4
    Rectangle:
      size: self.size
      pos: self.pos
  size_hint: 1,0.2
  GridLayout:
    size_hint: .25,1
    rows: 2
    cols: 1
    AnchorLayout:
      anchor_x: 'right'
      anchor_y: 'bottom'
      GridLayout:
        size_hint: .9,.75
        cols: 2
```

```
        AnchorLayout:
          anchor_x: 'left'
          anchor_y: 'top'
          Button:
            size_hint: .98,.95
            id: buttonGo
            text: 'Debug'
        AnchorLayout:
          anchor_x: 'right'
          anchor_y: 'top'
          Button:
            size_hint: .98,.95
            id: buttonExit
            text: 'Salir'
    AnchorLayout:
      anchor_x: 'right'
      anchor_y: 'top'
      Spinner:
        id: spinnerEstrategia
        size_hint: .9,.75
  BoxLayout:
    Label:
      id: labelBuggyFunction
      text: 'La funcion buggy es: '
      color: 1,1,1,1
```

Popup

Muestra un mensaje sobre los nodos desconocidos, y pregunta al usuario si quiere volver a recorrer los nodos con estado desconocido para encontrar definitivamente el error.

```
<DKPopup>:
  cols:1
  Label:
    id: labelInfoDK
  GridLayout:
    cols: 2
    size_hint_y: None
    height: '44sp'
    Button:
      id: buttonAccept
      text: 'Aceptar'
    Button:
      id: buttonCancel
      text: 'Cancelar'
```

Módulos secundarios

En esta sección explicaremos los módulos o clases sobre las que se apoya el módulo principal de la interfaz para conseguir recrear una representación visual de la aplicación con éxito

CustomTreeNode

Esta es una clase encargada de manejar los eventos relacionados con el *widget* del mismo nombre. Contiene una referencia al controlador, para poder comunicarse de forma rápida con el mismo; una referencia al nodo del árbol de llamadas que representa visualmente, y una referencia al id de ese mismo nodo.

paramsToString()

Es una función dedicada a convertir el diccionario de parámetros de entrada y salida del nodo a un string que sea fácilmente legible por el usuario.

updateNodes y updateColor

Estas funciones se encargan de actualizar la etiqueta que representa el estado del nodo en la interfaz. La primera modifica el texto de la etiqueta, y, la segunda, modifica el color del mismo además del resto de las etiquetas.

Menu

Esta clase maneja los eventos provocados por las acciones de los usuarios en el *widget* del mismo nombre. Contiene una referencia al controlador para facilitar la comunicación con los métodos del mismo.

DKPopup

Esta clase maneja los dos eventos provocados por las acciones de los usuarios en el *widget* del mismo nombre, anclado a la ventana emergente de la interfaz.

Capturas de pantalla de la interfaz

Esta interfaz está en una primera fase de desarrollo, que planeamos mejorar a lo largo del tiempo. La intención de esta interfaz es que fuera completamente funcional y fuese más fácil e intuitiva de usar que su homóloga de terminal.

En la figura 1, vemos la interfaz principal. La parte inferior es el menú del que hemos hablado antes que contiene las funciones básicas del depurador. La parte superior es la representación gráfica del árbol mediante el TreeView, cuyos nodos son CustomTreeNode.

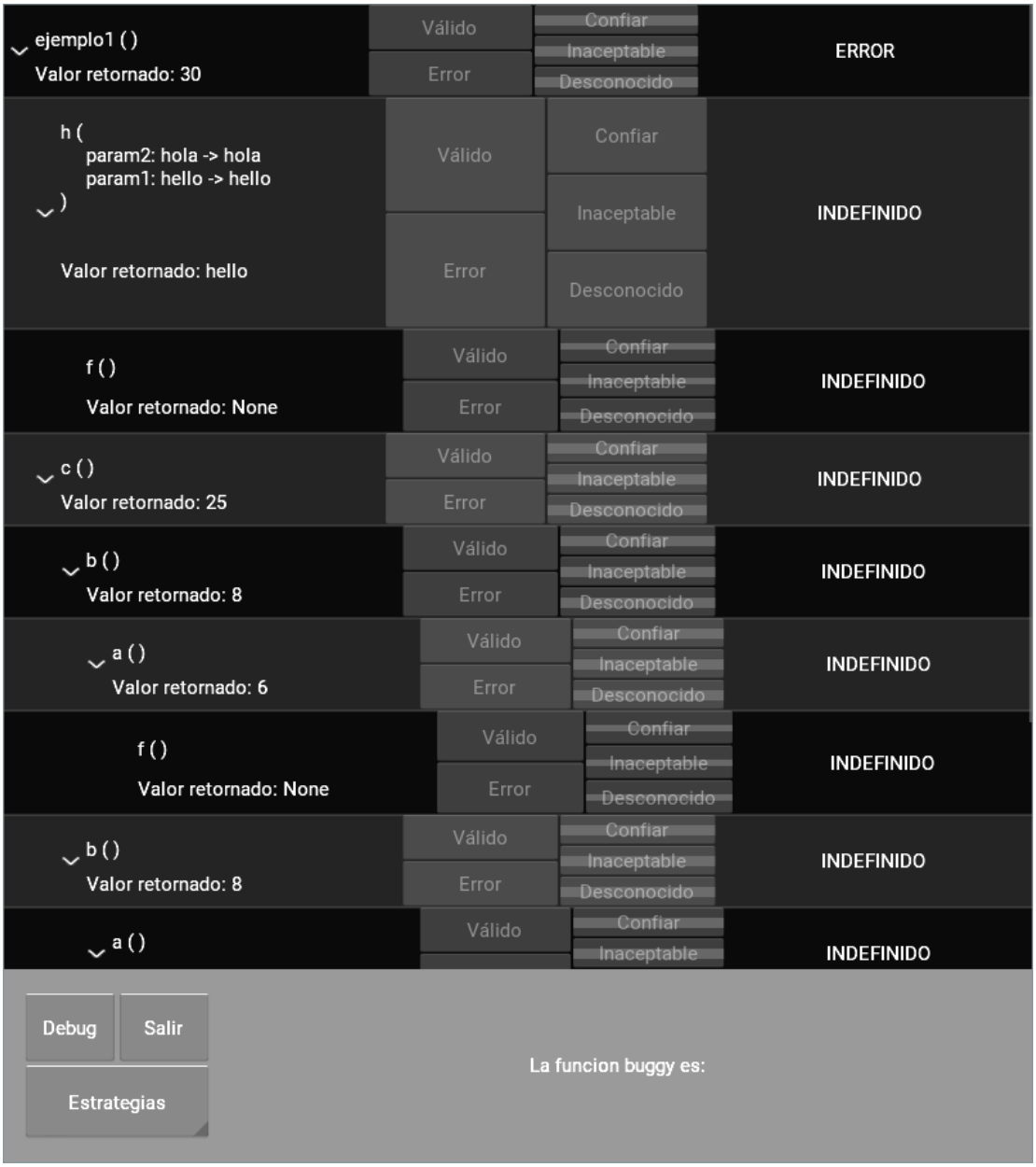


Figura 1. Estado Inicial de un programa a depurar

En la figura 2, lo que podemos observar es el inicio de la depuración, el primer nodo se marca con estado error y, el siguiente nodo escogido por la estrategia, es resaltado por la interfaz y, a su vez, son habilitados los botones de este mismo nodo.

También, en la parte inferior, podemos ver cómo se realiza el cambio de estrategia mediante un desplegable, que siempre estará disponible

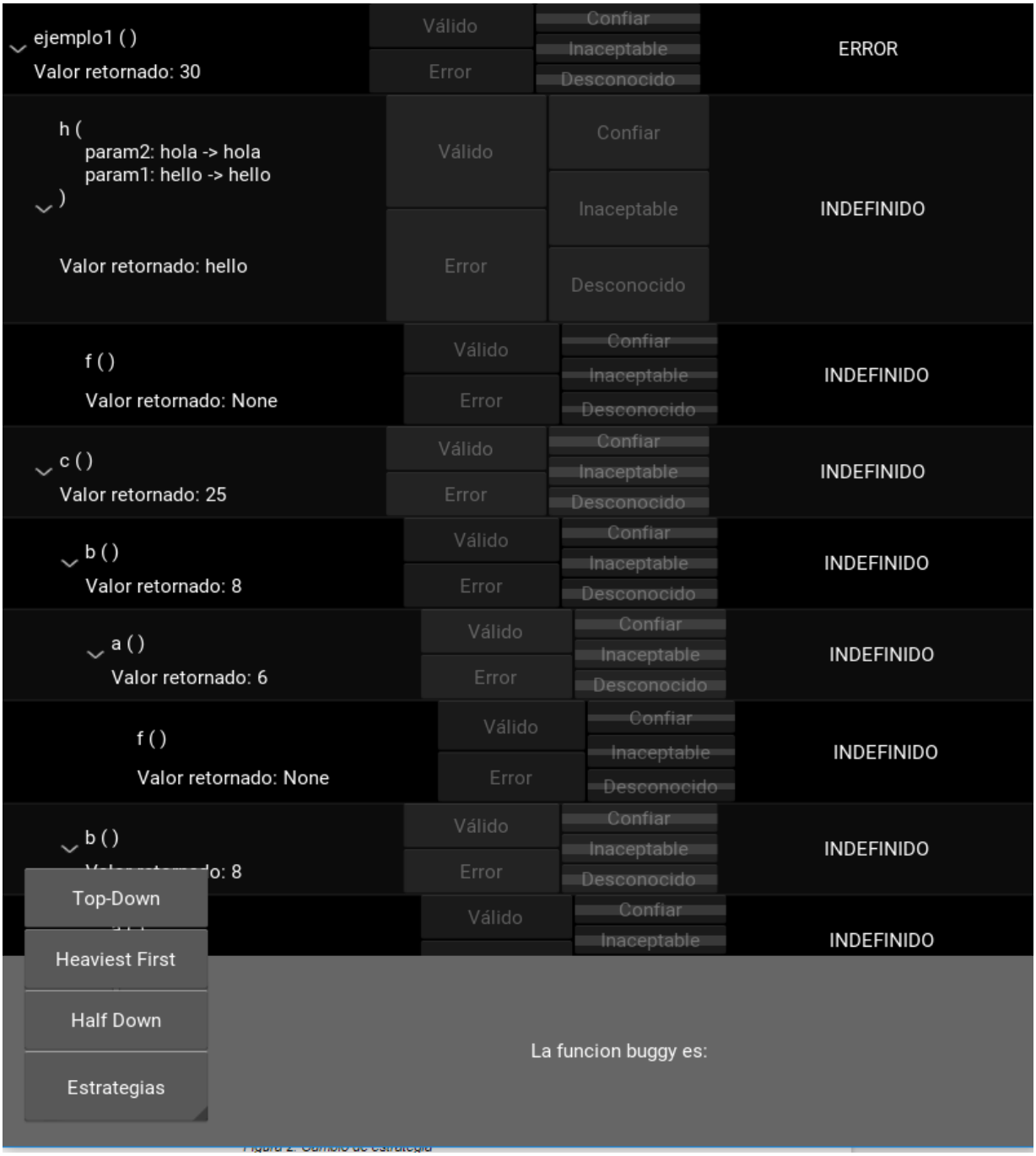


Figura 2. Selección de estrategias

En la figura 3, observamos el *popup* del que hemos hablado anteriormente. Una vez terminada la ejecución del recorrido principal, si existen nodos que tienen estado desconocido, pero podrían contener el error, se muestra este mensaje emergente.

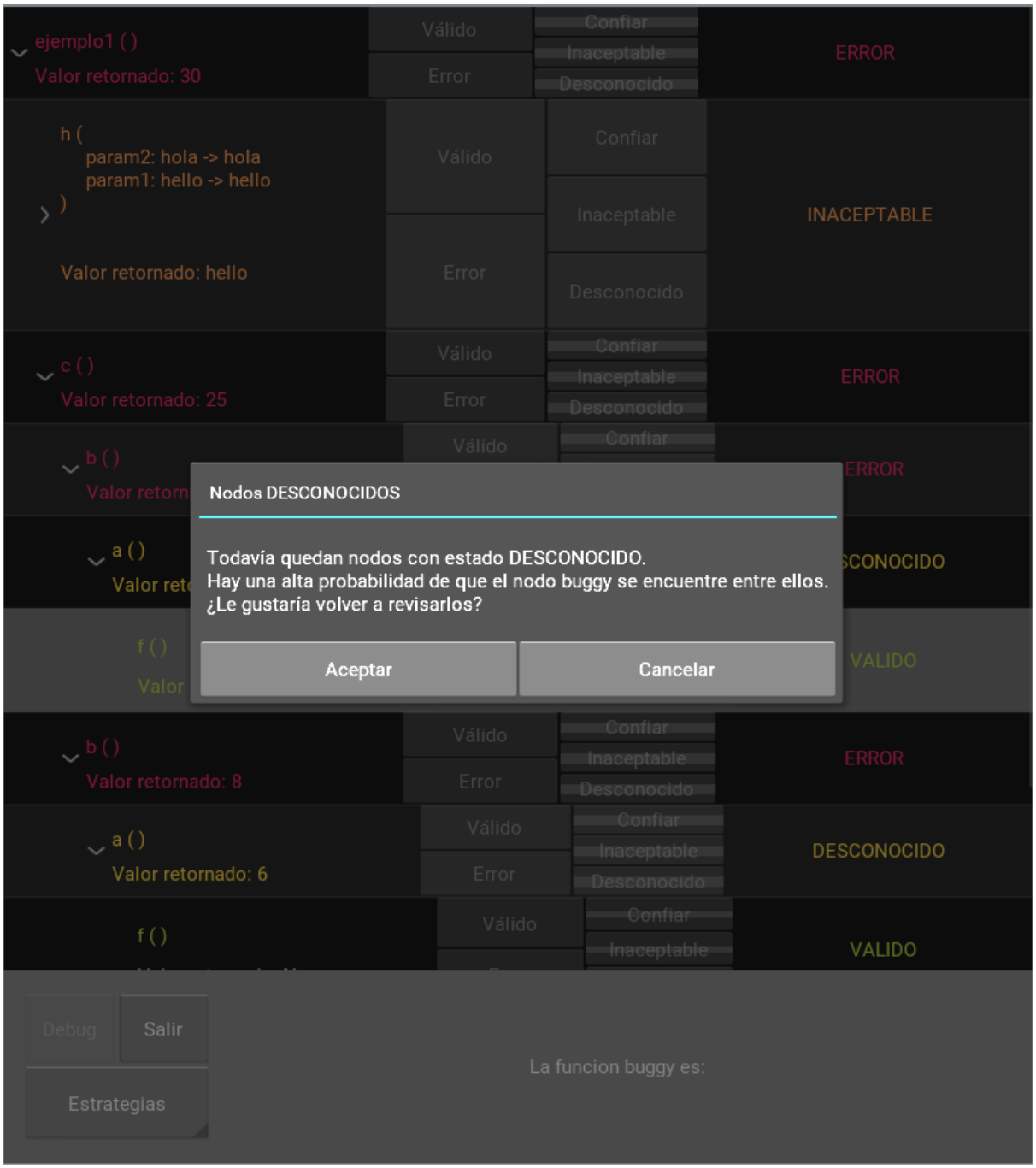


Figura 3. Análisis de nodos DESCONOCIDOS al finalizar la ejecución normal

En la figura 4, se muestra la finalización del programa. En la parte inferior podemos observar cómo se informa al usuario de cuál es el nodo *buggy* y cuál tiene altas probabilidades de serlo.

Figura 4. Posibles mensajes ofrecidos al usuario en la búsqueda del nodo buggy

Importancia del *testing*

Uno de los aspectos más importantes de nuestro trabajo con este depurador fue el *testing*. Al crear un programa el número de problemas y errores que pueden surgir es inabarcable, sin embargo a la hora de depurarlo debes abarcar todos esos casos, ya sean muy comunes o muy remotos.

Esta tarea era una de las que mayor tiempo consumía ya que se realizaba de manera completamente manual. Especialmente en las fases medias del desarrollo, cada vez que hacíamos un ligero cambio en la lógica del depurador (para incluir un nuevo caso, subsanar algún error…) se debían hacer pruebas de una gran cantidad de casos conocidos de errores para comprobar que eran detectados y que nuestro cambio no había afectado al buen funcionamiento del depurador.

Así mismo, cuando encontrábamos un nuevo caso de error o intentábamos ampliar las funcionalidades del depurador (por ejemplo, el estado desconocido o las excepciones), debíamos realizar nuevamente pruebas para asegurar la robustez del depurador.

¿Por qué debíamos hacer tantas pruebas?

El *testing* es una parte fundamental del desarrollo software, por un lado permite comprobar el funcionamiento del *software* a medida que se va generando, por otro lado ayuda a detectar errores en fases tempranas impidiendo que proliferen a fases posteriores y puedan generar fallos más graves. Con un esfuerzo previo en pruebas se evitan retrasos innecesarios posteriormente y facilita la ampliación y mejora del mismo.

¿Qué pruebas se realizaron?

En las primeras fases se realizaban pruebas con un pequeño ejemplo introducido en el propio código del depurador, el cual se ejecutaba al iniciar el depurador, esto útil en las primeras fases cuando se probaban las funciones por primera vez y se necesitaba un código reducido y no muy complejo para ver rápidamente si había algún error grave.

Una vez familiarizados con el sistema de trazas de Python y con varios recorridos desarrollados, empezamos a crear pequeños ejemplos de código que eran ejecutados a través de un comando del depurador que los invocaba. Había varios tipos de ejemplos según las pruebas que necesitábamos realizar.

En las fases finales de desarrollo comenzamos a utilizar ejemplos reales de algoritmos o funciones recursivas en las que introducimos intencionadamente un error y ejecutábamos el depurador para comprobar si detectábamos el error introducido.

Instalación y uso

En esta sección se van a nombrar las librerías de terceros utilizadas en el proyecto y cómo deben instalarse en Windows para poder ejecutar correctamente el depurador, mientras que las instalaciones para Linux [9] y Mac OS [10] tan solo serán citadas en la sección de referencias a las direcciones web oficiales. También se va a hablar de cómo debe instalarse **Debuggy** para su correcto funcionamiento.

Dependencias

Como ya se ha hablado de estas bibliotecas previamente, tan solo se van a nombrar antes de explicar su instalación.

- Instalación de Kivy (Interfaces gráficas)

    1. Instalar la última versión de Pip y Wheel.

```
python -m pip install --upgrade pip wheel setuptools
```

    2. Instalar las dependencias propias de Kivy.

```
python -m pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew
python -m pip install kivy.deps.gstreamer
```

    3. Instalar Kivy

```
python -m pip install kivy
```

- Colorama (Añadir colores a la terminal)

    1. Instalar Colorama

```
pip install colorama
```

Al contrario que Kivy, esta no requiere dependencias adicionales al propio Python para poder trabajar con ella.

Descarga e instalación de Debuggy

Para poder hacer uso del depurador debemos seguir los siguientes pasos:

1. Instalar (si no se tiene instalado) Python 3.x o superior.
2. Descargarse o clonar el proyecto, se encuentra disponible en el repositorio. https://github.com/Wizsmiles/TFG_UCM/

3. Instalar las dependencias especificadas previamente.
4. Ejecutar el comando “python Main.py <inputfile> <function> <-g|-f>”
    - **inputfile:** ruta al archivo donde se encuentra la función que se va a depurar.
    - **function:** nombre del método a analizar.
    - **-g:** esta opción activa la interfaz gráfica.
    - **-f:** esta opción activa la transformación del árbol.

Contribuciones

En esta sección de la memoria cada miembro del equipo explicará cuales han sido sus labores durante todo el proceso de desarrollo y cuáles han sido sus sensaciones al trabajar en equipo.

José Javier Escudero Gómez

Nada más comenzar con el Trabajo de Fin de Grado al inicio de curso me dí cuenta que tenía dos limitaciones a la hora de poder abordar todo el trabajo que tendría por delante junto a mis dos compañeros: mi conocimiento tanto de Python como en el ámbito de la depuración declarativa no era el que se podía desear para al llevar a cabo un proyecto como el que se había acordado con los tutores.

Para solventar este problema, estuve varias semanas estudiando y realizando pequeños proyectos en la versión 3 de Python así como pude comprender en amplio detalle en qué consistía la depuración declarativa gracias a la bibliografía que nos recomendaron nuestros tutores y que aparece en la sección bibliografía de esta memoria.

Solventado este escollo inicial, tomé la iniciativa y con la aprobación de mis dos compañeros me convertí en el puente de comunicación entre los tutores y el grupo, para así comenzar con el sistema de hitos quincenal y realizar el reparto de tareas correspondientes para cumplir cada uno de estos objetivos.

Durante los primeros hitos del proyecto, mi labor consistió en crear la estructura básica que se encargaría de obtener las trazas del sistema necesarias para la creación del árbol de depuración así como una primera versión de la clase encargada de crear el propio árbol de depuración. Esta tarea no fue trivial, ya que la documentación oficial de Python no se caracteriza por ser demasiado clara, y en el caso concreto de buscar información acerca de su módulo de sistema **sys** y su forma de obtener trazas del sistema al ejecutar código supuso un verdadero reto al que tener que enfrentarse.

En los hitos intermedios del proyecto, me encargué de añadir complejidad tanto a la estructura que llevaba a cabo la captura de trazas del sistema para que fuera capaz de capturar excepciones, así como me encargué de llevar a cabo el proceso de transformación del árbol para reducir el número de preguntas.

Al contrario que mi labor en los primeros hitos, estas tareas no fueron demasiado complejas gracias a toda la experiencia adquirida en el proceso y conocer de primera mano todo nuestro modelo de datos para poder modificar o ampliar lo que fuese necesario. Además, también me encargué de ayudar a probar todos los recorridos del árbol que mis compañeros habían implementado.

Mi labor durante las etapas finales del proyecto pasó a ser un mero soporte en las tareas que mantenían atascados a mis compañeros, como fue todo lo relacionado con el recorrido del árbol al implementar el **estado desconocido**. Añadir este estado a la lista de estados

posibles al evaluar un nodo supuso un cambio importante en toda la lógica encargada de llevar a cabo los distintos recorridos del árbol y llevó bastantes horas de trabajo el subsanar todos los problemas e inconsistencias que aparecieron.

Aparte de la tarea anterior, también me encargué de buscar una biblioteca alternativa a la estándar de Python para realizar interfaces gráficas (**TKinter**), ya que es muy poco intuitiva y se hace bastante difícil el tener que trabajar con ella si se desea realizar algo medianamente complejo sin tener que invertir una gran cantidad de tiempo.

En tan solo una tarde de búsqueda intensa encontré **Kivy**, una biblioteca que nos gustó a todos al ver varios ejemplos de interfaces y lo fácil e intuitivo que era trabajar con ella. Otro punto bastante importante que nos hizo decantarnos por este y no otro *framework* de los que estuvimos barajando fue la gran cantidad de ejemplos disponibles en la propia web junto a una extensa API pública que rellenaba con creces todos los huecos de conocimiento más avanzados que dejaban los ejemplos.

En cuanto a la memoria, mis contribuciones personales han sido:

- Realizar el Plan de Trabajo. Para ello, me basé en las actas que fuimos recogiendo en cada reunión con los tutores.

- Trazador personalizado. Basado en mi experiencia personal tras investigar la documentación oficial de Python y la posterior codificación de una solución que satisficiera las necesidades del proyecto.

- Construcción del árbol. Basado en mi experiencia personal al trabajar en esta parte tras haber estudiado previamente todo lo relacionado con trazar el programa a depurar.

- Transformación del árbol. Basado en mi experiencia personal al trabajar en ello. Haber trabajo previamente en la construcción del árbol fue de gran ayuda al realizar esta tarea.

Considero que tanto el trato recibido como el realizado con mis compañeros ha sido muy bueno y hemos sabido compenetrarnos a la perfección. Hemos tenido las cosas claras desde el primer momento y creo que todos hemos aportado ideas y opiniones clave para poder resolver todos los retos y problemas que fueron apareciendo en el camino hasta llegar al resultado final que teníamos en mente.

Aunque nuestros horarios hayan sido bastante dispares debido a estar los tres trabajando con diferentes horarios y estar matriculados en distintas asignaturas, siempre hemos sido capaces de sacar tiempo cada semana para hablar y planificar cada paso del proyecto de forma conjunta tanto de forma presencial como con herramientas digitales.

Respecto a los tutores, solo puedo decir que ha sido un placer trabajar con ellos. Siempre han estado ahí cuando hemos necesitado información y consejo ya fuera por correo electrónico o quedando de forma presencial cuando lo hemos necesitado adaptándose a nuestros horarios sin problema.

Sergio Freire Fernández

Como mis compañeros, mi primera tarea fue profundizar en mis conocimientos de Python, ya que aunque todos conocíamos el lenguaje nuestra experiencia y conocimiento sobre el mismo era reducido. Así pues después de las primeras reuniones con los profesores y antes de trazar nuestro plan de trabajo y fijar reuniones, nos dedicamos a profundizar en nuestros conocimientos sobre este lenguaje y sobre la programación declarativa. Para esto último fue de gran ayuda la bibliografía facilitada por los profesores.

Una vez trazado el plan y fijadas reuniones con los profesores empezamos a desarrollar nuestro trabajo, aunque al principio hacía alguna tarea de revisión del código que desarrollaba José, mi primera tarea principal fue el desarrollo de la clase **Recorrido**. Los primeros pasos a seguir fueron la estructuración de la clase y de la función “ask” encargada de hacer las preguntas del depurador al usuario. El siguiente paso fue la definición y desarrollo de las funciones de estrategia de recorrido, siendo la primera **Top Down** y la segunda **Heaviest First**.

Llegados a este punto nuestros recorridos eran capaces de recorrer los árboles asignándoles hasta 3 estados (válido, error y confiar), fue entonces cuando mi compañero Sergio Ulloa comenzó a desarrollar la estrategia **Half Down** apoyándose en el código que había desarrollado. Mientras tanto José y yo nos encargamos de investigar el módulo **Tkinter** de Python para iniciar el desarrollo de la interfaz.

Tras un tiempo de investigación y pruebas, y varios intentos poco fructuosos de crear un prototipo de interfaz decidimos reunirnos con los profesores para analizar alternativas a Tkinter ya que su complejidad y estética poco vistosa no se adaptaba a lo que nosotros queríamos. Fue así como finalmente decidimos que la interfaz se desarrollaría utilizando el módulo **Kivy**.

Después de esto, mis tareas volvieron a girar en torno a los recorridos, había que introducir nuevos estados y perfeccionar los recorridos. Lo primero fue introducir el concepto de *buggy* y que el depurador se detuviese al encontrar este estado, al principio parecía una tarea sencilla pero al introducir nuevos estados.

Fue en esta fase cuando se incluyeron los nuevos estados de **inadmisible** y **desconocido**.
El primero fue sencillo, en parte porque no profundizamos demasiado en todas las implicaciones que tiene, de las que hablaremos en el apartado de “Conclusiones y trabajo futuro”.

El segundo en cambio fue una de las tareas más arduas del depurador. Debido a que no encajaba ni como un estado correcto ni como un estado incorrecto, sui inclusión en la lógica era compleja ya que había que tratarlo como un caso especial, lo que abría nuevos caminos de depuración que no habíamos contemplado.

El estado desconocido era un concepto que resultaba ligeramente abstracto, ya que para recorrido la lógica del mismo variaba, la complejidad no radicaba en el propio estado en sí,

sino en cómo continuar el recorrido después de detectar dicho estado y sobre todo en cómo analizar si se había encontrado el **nodo buggy**.

Para esta tarea tuve la ayuda de José, quien me daba soporte y me ayudaba a hacer pruebas del nuevo estado. El procedimiento era el siguiente, proponía una lógica para el "desconocido", analizábamos su viabilidad, la implementábamos y la probábamos,. Ssi satisfacía los casos de prueba pasábamos a la siguiente estrategia, si no, volvíamos al primer paso.

El estado "desconocido" fue lo que consumió la mayor parte de nuestros esfuerzos hacia las fases finales. Una vez probado y tras asegurarnos de su funcionamiento, tuvimos una reunión con los profesores para mostrarselo. En esa misma reunión establecimos los puntos que debíamos abordar en la memoria así como las últimas tareas que debíamos completar, que en mi caso han sido:

Memoria

- Depuración declarativa: una breve explicación de lo que es la depuración declarativa basándome en la bibliografía de la que disponíamos.
- Importancia de *Testing*: explicación sobre la importancia de las pruebas para nuestro proyecto y la ingeniería del software en general.
- Clase Recorrido: descripción detallada de los algoritmos y estructuras utilizados para implementar dicha clase.
- Conclusiones y trabajo futuro: detalle sobre las conclusiones obtenidas de nuestro trabajo y breve análisis sobre posibles ampliaciones o aplicaciones de nuestro depurador.
- Traducciones de introducción y conclusiones.

Depurador

- Apoyo a las tareas de mis compañeros.
- Mantenimiento y revisión de depurador (en especial para revisar que los cambios finales no afecten al funcionamiento, como por ejemplo la inclusión de la interfaz gráfica).
- Pruebas y *testing*.

Por último agradecer a mis compañeros su esfuerzo y dedicación y a los tutores su implicación con nosotros, ha sido un placer trabajar con ellos y compartir con ellos este proyecto.

Sergio Ulloa López

Mi contribución al proyecto comienza con retomar los escasos conocimientos que tenía sobre el lenguaje con el que realizaríamos el depurador. Estos conocimientos en Python los adquirí durante la asignatura de Minería de Datos y Big Data junto con algo de curiosidad personal. Al haberlos obtenido de dicha asignatura, no estaban para nada orientadas al futuro trabajo que nos esperaba, ya que me hubiesen sido de utilidad en el caso de haber realizado un proyecto sobre minería de datos o tal vez machine learning. Sin embargo, la base era la misma, y no fue especialmente complicado volver a adquirir soltura en el lenguaje para programar de forma eficaz y rápida.

Otro obstáculo que encontré en los primeros compases del desarrollo fue el absoluto desconocimiento de lo que era un depurador declarativo. Un problema que fue subsanado gracias a la documentación aportada por nuestros dos tutores y a varias reuniones con los mismos, que nos sirvieron en cierto modo de explicación sobre la materia.

Después de lo que fue la primera reunión para establecer una serie de hitos en el tiempo y programar una reunión cada dos semanas para así tener un control constante de qué era lo que habíamos hecho y qué es lo que debíamos hacer a continuación, mi primera tarea consistía en el desarrollo de la clase **View**. Esta clase se encargaría posteriormente de imprimir el árbol formado por las llamadas a las funciones del programa a depurar. Consistía, básicamente, en una clase que cuenta únicamente con métodos estáticos para facilitar la reutilización en diferentes partes del código sin tener que guardar ni una referencia, ni una copia al árbol que se imprimiría.

View cuenta con dos métodos estáticos:

- **show(nodo):** es un método que simplemente sirve como inicializador de la recursión. Llama al siguiente método.

- **recursiveShow(nodo, nivel):** es un método recursivo que se encarga de cambiar el color de la consola en función del estado del nodo. Esto se consigue mediante el módulo Colorama, el cual encontré tras un poco de investigación, tras darme cuenta de que la forma nativa de Python de hacer el cambio de colores era demasiado compleja.

Una de mis funciones dentro del equipo fue la de recoger información de las reuniones con nuestros tutores para luego no olvidarnos nada en posteriores reuniones y llamadas entre Sergio, José y yo. Estos documentos recogían ayuda e instrucciones para llevar a cabo los hitos más cercanos en el tiempo.

En una fase posterior, mientras mis compañeros investigaban módulos gráficos e interfaces, yo implementé el algoritmo Half Down para recorrer el árbol. Esto lo realicé con la ayuda proporcionada en las explicaciones de nuestros tutores, la ayuda de mi compañero Sergio que ya había hecho otros dos estrategias y diversa información en Internet que me ayudó a comprender el criterio que se utilizaba en el algoritmo para escoger una u otra rama del árbol.

A partir de la finalización de esta última tarea y mientras mis compañeros se dedicaban a implementar nuevas respuestas para el depurador, yo pasé a realizar tareas de mantenimiento y refactorización del código para dejarlo relativamente más comprensible, modularizado y limpio.

Una de las dificultades que encontré al realizar esta modularización fue el intento de crear una clase Main, cosa que, desafortunadamente, nos resultó imposible, ya que era necesario que el trazador que creaba el árbol a partir de la ejecución del programa se encontrase fuera de cualquier clase o función, es decir, en modo imperativo. Por suerte, sí nos fue posible crear un **Controlador** que conectase el modelo y la lógica del depurador con la parte gráfica del mismo. Yo fui el encargado de esta tarea.

Una vez terminado el controlador, me dediqué a la implementación de un método lanzador que permitiese al futuro usuario de nuestro depurador un uso más sencillo y rápido sin necesidad de realizar cambios en el código fuente. Aunque este método podría parecer algo sencillo, hay que tener en cuenta que necesitábamos llamar a una función de un archivo Python externo del cual solo conoceríamos su ruta. Es decir, debíamos importar el módulo para después llamar a una función de este mismo. Esto fue posible gracias a que Python permite ejecutar comandos a través de la función `exec` que acepta un `string` como argumento.

Una vez terminadas todas las funcionalidades del depurador, fue mi turno para comenzar el desarrollo de la interfaz gráfica, la cual no hubiese sido posible sin la ayuda de mi compañero José, quién investigó diversos frameworks gráficos de Python, entre ellos el integrado en el propio lenguaje, **TkInter**, que resultó ser la peor solución de las que había considerado. Poco después se topó con **Kivy,** un *framework* que no conocíamos pero que era uno de los más extendidos, y uno de los pocos con una documentación sorprendentemente útil.

Como la mayoría de desarrollos de interfaces gráficas, fue un proceso largo y pesado, ya que se tuvieron que volver a pensar los algoritmos de recorrido de árbol para adaptarlos a las exigencias del framework, además la forma de tratar los nodos desconocidos no es idéntica a la de la interfaz de consola.

Para terminar, añadiré mi contribución a la fase final del proyecto:

- Por una parte, mi rol ha consistido en asegurarme de que la parte de la lógica de la interfaz se correspondía con la lógica de consola, volver a redactar la información que aparece por pantalla en la terminal, para que sea todo coherente y comprobar que la implementación de la GUI no afecta en absoluto al funcionamiento ya existente de la interfaz de consola.

- Por otra parte, en la redacción de la memoria, me he encargado de los siguientes apartados:
    - **Objetivos:** en los que recojo las motivaciones y necesidades que nos han hecho esforzarnos en este proyecto.

    - **Python:** en el que explico en qué consiste el lenguaje que hemos utilizado.

- **Kivy:** en el que explico el framework Kivy, su lenguaje propio en archivos .kv.

- **Interfaces gráficas:** una explicación más exhaustiva sobre nuestra implementación de la interfaz de consola y la GUI mediante Kivy.

Por último, agradecer a nuestros tutores su gran trabajo y su gran empatía con el nuestro. Han puesto todo de su parte para apoyarnos en el desarrollo de este depurador. También agradecer a mis compañeros, José y Sergio, su esfuerzo e implicación, sin ellos hubiese sido imposible llevar esto a cabo.

Conclusiones y trabajo futuro

En esta sección se han recopilado todas las conclusiones obtenidas por el equipo de desarrollo en el proceso así como algunas mejoras en las que nos gustaría trabajar de cara al futuro.

Conclusiones

Nuestro trabajo ha consistido en desarrollar un depurador declarativo para Python que soporta cualquier código de Python 3 en adelante, consta de dos interfaces una por terminal de consola y otra gráfica para el usuario.

Extraemos principalmente dos conclusiones tras desarrollar este trabajo, por un lado el hecho de que disponer de un depurador declarativo facilita mucho la tarea al programador a la hora de depurar su código, especialmente cuando este todavía no está muy familiarizado con el mismo.

Por otro lado también hemos detectado ciertos problemas a la hora de realizar esta tarea. Lo más destacable es la documentación de Python, que consideramos poco completa y muy compleja de analizar, en muchos casos hay una total falta de ejemplos que retrasa mucho la tarea de aprendizaje del mismo. Este factor ha obligado tener que complementar la paupérrima documentación oficial con foros y comunidades de usuarios desde donde hemos podido rellenar las lagunas dejadas por la propia documentación.

Por otra parte también destaca lo poco orientado que está Python para el desarrollo de interfaces gráficas ya que los módulos que trae predeterminados quedan obsoletos en cuanto a diseño y experiencia de usuario, en cuanto a módulos externos, el principal problema es que requieren instalaciones adicionales o falta de versatilidad en los mismos.

Trabajo futuro

En segundo lugar, hay una variada lista de posibles ampliaciones y mejoras del proyecto que se podrían implementar, que detallamos a continuación:

- **Ampliar el número de estrategias:** actualmente consta de 3 estrategias, se pueden definir tantas estrategias como algoritmos de recorrido de árboles, pudiendo así adaptar la estrategia al tipo de árbol que genera nuestra función en el depurador.

- **Estrategia de navegación libre:** en consonancia con el punto anterior, también se puede definir una estrategia en la que el usuario sea el que decida cuál será el siguiente nodo que comprobará. Esta ampliación es específica de la interfaz gráfica.

- **Ampliación del estado Inaceptable:** actualmente el estado inaceptable se aplica como una caso especial de estado correcto, pero debido a sus implicaciones de error, se podría crear un módulo que, una vez detectado un inaceptable, inspeccionase de nuevo el código buscando dónde se invoca a esta función y por

que no se hace correctamente, para aislar un posible problema ya que probablemente el buggy se genere ahí.

- **Generar plugins para IDEs:** generar un plugin para IDEs como Eclipse o Jupyter, que permitan ejecutar el depurador directamente sin necesidad de utilizarlo como un programa externo.

- **Crear un paquete para su instalación:** empaquetar el código para que se pueda instalar a través del comando **pip install**.

Conclusions and Future Work

In this section we have collected all the conclusions obtained by the developers in the process, as well some improvements we would like to work on in the future.

Conclusions

Our goal was to develop a declarative debugger for Python that supports any code for Python 3 onwards, it has two interfaces one for console terminal and a graphical user interface.

After working on this project we reached two main conclusions, the first one is the fact that to have a declarative debugger greatly simplifies the task of the programmer when debugging the code, especially when the programmer is not very familiar with the code.

On the other hand, we have detected some issues when doing this task. The most remarkable thing is Python documentation; we consider it isa bit incomplete and hard to analyze, in many cases the absolute lack of examples greatly delays the learning task. This fact has forced us to complement the official documentation with forums or user communities where we could fill the gaps of the documentation.

Moreover, it is highly remarkable the difficulty of implementing graphical user interfaces in Python, because the default modules are outdated at design and user experience. The external modules are quite better but they need additional installations or they have low versatility.

Future work

Secondly, we present a list of possible extensions and improvements of this project that can be implemented, we detail them below:

- **Extend the number of strategies:** currently it has 3 strategies, it can be defined as many strategies as tree navigation algorithms exist, adapting the strategy to the type of tree that generates our function in the debugger.

- **Free navigation strategy:** in line with the previous point, you can also define a strategy where the user decides which will be the next node to check. This extension is specific to the graphical interface.

- **Expansion of the unacceptable status:** currently the unacceptable status is applied as a special case of correct status, but due to its error nature, but a fix can be created that, once detected an unacceptable, checked the code again looking for where this function is called and why it is not done correctly, to isolate a possible problem, since the buggy is probably generated there.

- **Plugins for IDEs:** generate a plugin for IDEs like Eclipse or Jupyter, which allows users to execute the debugger directly avoiding the need of using it as an external program.

- **Create an installation package:** pack the code so it can be installed through a tool like **pip install**.

Referencias

1. Rafael Caballero, Adrián Riesco, Josep Silva:
   A Survey of Algorithmic Debugging -
   http://users.dsic.upv.es/~jsilva/papers/CSUR2017.pdf (10-01-2018)
2. The Python Tutorial - https://docs.python.org/3/tutorial/index.html (21-05-2018)
3. Artículo de Python en la Wikipedia - https://es.wikipedia.org/wiki/Python (21-05-2018)
4. Python Crash Course: A Hands-On, Project-Based Introduction to Programming
5. David Insa, Josep Silva, and César Tomás:
   Enhancing Declarative Debugging with Loop Expansion and Tree Compression -
   http://users.dsic.upv.es/~jsilva/papers/LOPSTR2012.pdf(22-05-2018)
6. Artículo sobre árboles en la Wikipedia -
   https://es.wikipedia.org/wiki/%C3%81rbol_(inform%C3%A1tica)(22-05-2018)
7. Sitio web de Kivy - https://kivy.org/#home (22-05-2018)
8. Artículo de Kivy en la Wikipedia - https://en.wikipedia.org/wiki/Kivy_(framework) (22-05-2018)
9. Instalación Kivy en Linux - https://kivy.org/docs/installation/installation-linux.html
10. Instalación Kivy en Mac OS - https://kivy.org/docs/installation/installation-osx.html

Bibliografía utilizada

- Modern Python Cookbook - Steven F. Lott - Packt Publishing (2016)
- Mastering Python - Rick van Hattem - Packt Publishing (2016)
- Python Data Structures and Algorithms - Benjamin Baka - Packt Publishing (2017)
- Python. Paso a Paso - Angel Pablo Hinojosa Gutiérrez - Editorial RA-MA (2016)
- Documentación oficial de Python - https://www.python.org/doc/
- Rosetta Code - https://rosettacode.org/wiki/Rosetta_Code