Manual Técnico

Juan Alejandro Bernal

7 de julio de 2020

Índice

1. Introducción

La aplicación fue realizada en el lenguaje de programación Python, específicamente Python 3.8, un lenguaje de programación que es comparativamente lento, pero es fácil de leer y mantener. La aplicación fue realizada con múltiples tecnologías, por lo que se separo en módulos, los módulos principales son: **src, frontend y backend**, y es donde la aplicación tiene todo el código. A parte de estos 3 módulos hay 2 archivos, adicionales, el primero se llama **requirements.txt**, en el estan todas las tecnologías utilizadas en la realización de esta aplicación, esta puesto explicitamente como convención de Python, para poder instalar todas las dependencias haciendo uso del gestor de paquetes "pip". El ultimo archivo se llama README.md, en el cual están las instrucciones de instalación de dependencias para poner en funcionamiento la aplicación. La aplicación tiene la siguiente estructura:

```
chmaquina
├── src
├── frontend
├── backend
└── requirements.txt
├── README.md
```

Cada modulo tiene su propia sección en la cual explica el funcionamiento del mismo. Cada modulo tiene su propio trabajo, y fueron diseñados con la intención de reducir el acoplamiento entre ellas lo mas posible. Tener un sistema con poco acoplamiento fue un objetivo ya que se puede programar funcionalidades sin tener los demás módulos en un

estado funcional [1] , en ese momento, un ejemplo de esto es el modulo “frontend”, este modulo fue el primer modulo que se realizo, y se trata del modulo encargado de generar una pagina web con estilos, y de mandar y recibir solicitudes a las demás partes de la aplicación. Al programar este modulo, solo se tenían ideas básica, como lo es el tipo de datos que se debían mostrar. El tipo de solicitudes que se necesitaban mandar, el procesamiento que se le debía realizar a cada estructura de datos, el formateo de texto necesario para transformar algunos datos en mensajes. Por ahora vamos a empezar a explicar este sistema desde el nivel mas alto de abstracción, mostrando la estructura en cada nivel.

2. Src

El modulo “src” es el modulo mas importante del proyecto, ya que contiene toda la lógica que permite cargar, compilar y ejecutar todos los programas, por tanto, es también el modulo mas complejo y extenso del proyecto. Se divide en 2 carpetas principales, la primera se llama “scheduler” y tiene 2 tareas globales, la primera es la de proveer una API global para que los demás módulos puedan consumir la funcionalidad de este modulo, y la segunda es la de hacer la planeación de ejecución de programas. La segunda carpeta llamada “unittests” es una carpeta cuyo unico proposito es realizar pruebas de funcionamiento al sistema, estas pruebas son de caracter general, y levantaran una bandera si alguna funcionalidad genero un resultado inesperado.

```
src
├── scheduler
└── unittests
```

2.1. Algorithm

La clase algorithms es una clase abstracta, la cual va a ser heredada por cada algoritmo, esta clase da una interface común que todos los algoritmos deben implementar, cada clase que herede a la clase “algorithms” debe de implementar 5 funcionalidades, las cuales son:

- Setup: prepara los procesos para ser ejecutados.
- Run: Ejecuta los procesos.
- RunLine: Ejecuta solo una linea de un proceso.
- getOrder: Muestra el orden de ejecución de los procesos.
- getTable: Muestra una tabla de tiempos.

Adicionalmente de estos métodos abstractos, la clase Algorithm tiene un atributo llamado Time.

2.1.1. Time

La clase Time esta encargada de gestionar todos los factores relacionados con tiempo en tiempo de ejecución, se encarga de calcular los tiempos de ráfaga de cpu de cada proceso, de gestionar los tiempos de llegada y en caso del algoritmo RoundRobin, se encarga de gestionar el quantum, dado que Algorithm es padre de todos los algoritmos, todos los algoritmos tendrán acceso a este atributo.

2.1.2. Multihilos

Se implemento multihilos haciendo uso de las librerías "concurrent.future" de Python, al aplicar multihilos en tiempo de ejecución, dado el diseño de esta aplicación y la que la complejidad mas alta en todo el sistema es de O(n), resulta en un bajón de eficiencia aplicar multihilos con muy pocos procesos, dicho bajón solo puede ser visto en milisegundos. También se aplico de multihilos en el backend, para cargar todos los archivos asincrónicamente, aumentando la eficiencia del compilador.

2.2. Scheduler

La clase scheduler esta compuesta por 2 clases, la clase dispatcher y la clase algorithms, solo se preocupa por dirigir las funcionalidades de las demás clases, seleccionar un algoritmo de ejecución y de poner en disposición la API global de la aplicación.

2.3. Dispatcher

La clase dispatcher solo sabe como crear procesos, crear memoria para dichos procesos y mandar dichos procesos al "scheduler.º a la clase encargada de ejecutar dichos procesos llamada "instruction runner", la cual se encuentra en el submódulo "instructionHandling".

Su estructura es la siguiente.

```
dispatcher
├── dispatcher.py
├── declaration
├── errorHandling
├── factory
├── instructionHandling
└── memory
```

2.3.1. Declaration

Una instancia de la clase Declaración tiene 2 atributos, ambos atributos son instancias de una misma clase llamada DeclarableItem, esta subclase es trivial en implementación, se trata de una subclase que tiene un atributo llamado all_data, el cual es un diccionario de Python, el cual es en esencia un Hashmap, los Hashmap tienen una complejidad O(1), lo cual quiere decir que no importa cuantos valores se ingresen a este diccionario, el tiempo de búsqueda sera siempre constante, a pesar que Python es de los lenguajes mas lentos que hay en la actualidad, esta implementación es implementada en C, por lo que se garantiza velocidad a la hora de buscar variables. Cada proceso tiene asignado una instancia única de Declaration, por lo que cada proceso tiene sus etiquetas y variables completamente aislados de otros procesos.

2.3.2. ErrorHandling

Esta clase es estática, su única funcionalidad es añadir los mensajes de error en el arreglo de errores en memoria y mostrar los errores en el terminal.

2.3.3. Factory

Clase que surge de la refactorización del código, para poder abstraer la creación de instancias en el nivel más alto de abstracción. Esta clase solo se preocupa por la creación de instancias de las diversas clases que utiliza el dispatcher.

2.3.4. InstructionHandling

Este modulo esta encargado de cargar, procesar, guardar, y consultar todo lo referente a las instrucciones de un programa.

```
instructionHandling
├── compiler.py
├── instruction_runner.py
└── program_definitions
    ├── declaration_definitions.py
    └── operator_definitions.py
```

- **Compiler**

  La clase Compiler se encarga de procesar las instrucciones, eliminando los comentarios y errores de formateo que puedan tener las instrucciones, luego, se encarga de procesar las instrucciones buscando las palabras clave que estén definidas en su clase interna progDefs, esta subclase esta desacoplada, y sigue las reglas de la inversión de dependencias [2] para garantizar que en caso la aplicación funcioné con cualquier otro lenguaje que se quiera implementar. La creación de estas variables y etiquetas cae bajo la jurisdicción de la clase Declaration, la cual guarda toda la lógica de las mismas y las almacena en memoria. Una vez creadas las variables, las instrucciones son guardadas en memoria.

- **InstructionRunner**

  Es la clase que hace de "Wrapper" para cada proceso que se va a ejecutar, esta clase sabe todo lo referente a el "como" ejecutar un proceso, se encarga de consultar las instrucciones guardadas por Compiler, y tiene a la vez su propia instancia de progDefs, el cual es el conjunto de operaciónes que se han definido, se aplica la misma regla de la inversión de dependencias para así asegurar que se pueda intercambiar de definiciónes en el futuro, y mantener aislado el "que" del "como".

2.3.5. Memory

Este Modulo es trivial, la clase memory es utilizada por todas las partes del programa para guardar información de forma indirecta, ya que directamente solo guarda un arreglo donde estan las instrucciones de los programas, en el orden que llegaron, este arreglo es utilizado unicamente para dar información de los programas, a parte de este arreglo, memory dirige diversas clases y pone a disposición una API que es consumida por el resto de modulos en la aplicación, por lo que este es el nivel mas bajo de abstracción.

```
memory
├── memory.py
├── dataStream
├── declarationsInMemory
├── files_info
├── maquina_settings
├── queues
└── stepbystep
```

- **DataStream**

  Clase que contiene 5 arreglos que son consumidos en el frontend.

  - sdtout: Arreglo con todos los elementos que se deben mostrar en pantalla, al ejecutar un programa.
  - printer: Similar a sdout
  - stderr: Contiene todos los mensajes de error que puedan surgir
  - steps: Guarda todos los pasos que se realizaron al ejecutar cada programa
  - fileDescriptor: guarda los códigos de retorno al ejecutar un programa, 1 si hubo un error, 0 si no hubo error.

- **DeclarationsInMemory**

  Se encarga hacer un análisis de la memoria que a sido utilizada hasta el momento, teniendo en cuenta las declaraciones que se han guardado. Esta clase guarda todas las declaraciones que se han creado, cuando se guarda una declaración, también se guarda las instrucciones que corresponden a esa declaración, dado que cada proceso tiene una única declaración, entonces se hace uso de un diccionario donde la declaración es la llave y la instrucción el valor, permitiendo así una complejidad O(1) para el segundo proceso mas lento de la aplicación, optimizando la eficiencia del mismo, ademas de esto, al tener acceso a todas las declaraciónes, esta clase dispone de funcionalidades para devolver todas las variables y etiquetas creadas, junto con sus posiciones en memoria.

- **FileInfo**

  Guarda toda la información referente a los programas, su dirección en la computador, nombre, tamaño, etc.

- **MaquinaSettings**

  Sus atributos son el Kernel y la capacidad de memoria, esta clase hace cálculos referentes a capacidad disponible y utilizada.

- **Queues**

  Es una clase que guarda y pone a disposición todas las instancias creadas referentes a los procesos, los procesos pendientes, procesos terminados, instrucciones por ejecutar, etc.

- **StepByStep**

  Se encarga de la lógica asociada con formatear mensajes de paso a paso, utilizando las instancias de declaración, y las definiciónes de funciones de operaciónes y declaraciónes.

3. Frontend

El modulo “frontend” esta conformado por un archivo index, el cual tiene todo el codigo html que se utiliza para mostrar la pagina en un ambiente web, se hizo uso del framework libre ”Bootstrap“, el cual permite dar una estructura a paginas web de manera rapida.

```
frontend
├── assets
│   ├── css
│   ├── images
│   └── js
└──  index.html
```

3.1. Assets

En assests hay 3 módulos, los cuales tienen tareas diferentes.

- **css**: encargado de dar estilos a la pagina, dentro de estos estilos estan incluidas las proporciones, colores y efectos.
- **images**: es donde se guardan las imágenes que utiliza la pagina, son llamadas desde el mismo archivo html.
- **js**: es donde esta el código encargado de recibir y mandar solicitudes al modulo “Backend”, desde ahí se consumen todos los datos son producidos en el modulo “Src”.

4. Backend

Este modulo fue implementado utilizando el framework de python “Falcon”, el cual permite la rápida de una API REST, dentro de este modulo se crea una instancia de Scheduler, y se abren los puertos para recibir solicitudes y devolver datos. El frontend manda un archivo a este modulo, el modulo se lo pasa a el Scheduler, carga el archivo en memoria, y queda en espera por más solicitudes provenientes del frontend.

```
backend
├── run.py
├── buttons
│   ├── clean.py
│   ├── compiler.py
│   ├── runner.py
│   └── stepByStep.py
├── chmaquina_settings
├── lea
└── uploaded_files
```

4.1. Buttons

Este submodulo esta encargado de recibir las solicitudes mandadas por el frontend provenientes de un botón presionado, el frontend hace una solicitud HTTP POST, el backend lo recibe en el modulo correspondiente a la solicitud hecha.

- **clean**: Genera una nueva instancia de memoria en el Dispatcher, borrando todos los datos previos.
- **compiler**: Recibe archivos, los guarda en la carpeta uploaded_files, carga en memoria el archivo, y lo compila, luego deja a disposición de una solicitud HTTP GET toda la información de memoria, adicionalmente hace uso de múltiples hilos, ver 2.1.2 .
- **runner**: Corre todas las instrucciones, haciendo uso del algoritmo seleccionado.
- **stepByStep**: Corre una sola linea, devuelve el paso formateado y la información de memoria.

4.2. MaquinaSettings

Aquí se recibe la información referente a la cantidad de memoria y el valor del kernel que están marcados en el frontend, al recibir estos datos, se genera una nueva instancia de memoria en el Dispatcher, permitiendo reiniciar la aplicación cada que se cambie un atributo.

4.3. Lea

Su única tarea es la de recibir valores mandados por el frontend, en caso de haber una instrucción lea, al recibirlo, este valor es mandado directamente a la definición de operaciónes del respectivo proceso (ver sección InstructionHandling).

Referencias

[1] S. McConnell, "Code complete, second edition." `https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670`, 2004.

[2] R. C. Martin, "Clean code: a handbook of agile software craftsmanship." `https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882`, 2009.