UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de una herramienta de visualización de trazas de ejecución concurrentes

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Sun , Lishuang

Tutor/a: Vidal Oriola, Germán Francisco

CURSO ACADÉMICO: 2021/2022

Resumen

El desarrollo de las tecnologías de la información ha llegado a impactar de forma directa o indirecta todas las actividades económicas y la vida cotidiana del ser humano. La industria del *software* ha ido evolucionando hasta construir sistemas concurrentes y distribuidos que están presentes en prácticamente todas las aplicaciones y tecnologías que usamos en nuestro día a día.

Durante el desarrollo de una aplicación se realizan continuas validaciones para garantizar su correcto desempeño; además, se suele realizar una fase de depuración para encontrar las posibles raíces de los fallos detectados del programa. Si bien depurar un programa secuencial es relativamente sencillo, lo cierto es que no resulta así para los programas concurrentes debido, principalmente, al indeterminismo inherente a este paradigma. Sin embargo, disponemos de muchas alternativas a la depuración, tal es el caso del registro de las trazas de ejecución del programa en ficheros log.

En este trabajo final de grado se presenta el diseño e implementación de un visor multiplataforma (Windows y Linux) desarrollado en Java para las trazas de ejecución de los programas concurrentes basados en paso de mensajes. Para ello, analizaremos varias herramientas disponibles para crear interfaces de usuario, presentaremos los detalles del diseño e implementación de la solución propuesta y generaremos un ejecutable de la aplicación.

Palabras clave: traza, log, concurrencia, interfaz de usuario, GUI, multiplataforma.

Abstract

The development of information technologies has had a direct or indirect impact on all economic activities and people's daily life. The software industry has evolved to build concurrent and distributed systems which are present in practically all the applications and technologies that we use in a daily basis.

During the software development, continuous validations are carried out to guarantee its correct performance. Also, a debugging process is usually carried out to find the possible roots of the program's detected errors. Although debugging a sequential program is relatively easy, the truth is that it is not so when it comes to concurrent programs, mainly due to the indeterminism inherent in this paradigm. However, there are several alternatives to debugging, an example of which is recording program execution traces into log files.

In this final degree project, the design and implementation of a cross-platform viewer (Windows and Linux) developed in Java for the execution traces of concurrent programs based on message passing is presented. In order to achieve this, we will analyze several available tools to create user interfaces, present the details of the design and implementation of the proposed solution and generate an executable of the application.

Keywords : trace, log, concurrency, user interface, GUI, cross-platform.

Índice general

Índice de figuras

1. Introducción

En este primer capítulo vamos a presentar el contenido de este trabajo, incluyendo una breve descripción de la problemática de las técnicas de depuración en el contexto de la programación concurrente, los objetivos de nuestro trabajo y la estructura de la memoria.

1.1. Contexto y motivación

El diseño de sistemas concurrentes requiere técnicas fiables para coordinar su ejecución, el intercambio de información, la asignación de memoria, y una ejecución programada para minimizar el tiempo de respuesta y maximizar el rendimiento. Estos programas suelen ser complejos; por consiguiente, para comprobar su correcto desempeño, se necesita frecuentemente incluir en el código fuente instrucciones auxiliares que permitan el seguimiento de la ejecución del programa, presentando el estado de ejecución en cada instante de tiempo, como una alternativa manual a los depuradores.

Sorprendentemente, y a pesar de la gran cantidad de programas informáticos que se han creado a lo largo de la historia, son escasas las aplicaciones que muestran gráficamente el comportamiento a bajo nivel de dichos programas (1; 2).

Según un estudio realizado recientemente (3), un 49.9% del tiempo empleado para el desarrollo de un *software* se dedica a las continuas validaciones y pruebas necesarias para garantizar la calidad del producto. En este contexto, un visor de trazas de ejecución podría ayudar a reducir el tiempo que dedican los programadores a depurar, revisar y corregir los fallos detectados en sus programas.

1.2. Objetivos

El presente trabajo explora el mundo de la depuración de programas concurrentes basados en paso de mensajes (el caso de Erlang, sección 2). Y propone una solución *software* para ofrecer una interfaz amigable con el propósito de visualizar trazas de ejecución que reflejan la comunicación entre los procesos de un programa concurrente. En concreto, proponemos el desarrollo de la aplicación *Thread Logger Viewer*.

Para el correcto desarrollo y funcionamiento de la aplicación, debemos establecer explícitamente una serie de objetivos generales a nivel técnico:

- Implementar un *backend* para leer, procesar y extraer datos específicos a partir de unos ficheros log.
- Seguir una serie de buenas prácticas, tales como la programación modular, para facilitar el posterior proceso de mantenimiento.
- Definir y realizar un MVP (*Minimum Viable Product*) con el que el usuario pueda proporcionar *feedback* y sugerencias de mejora con respecto a la primera versión del producto.
- Estudiar el proceso de lanzamiento del *software* en diferentes plataformas, como Windows y Linux.

Por otra parte, se pretende comprender y profundizar en la gran variedad de herramientas disponibles para diseñar, crear y personalizar interfaces de usuario.

1.3. Estructura de la memoria

La memoria se estructura en varios capítulos. Cada capítulo se organiza en secciones, hasta cuatro niveles de profundidad. A continuación, se enumeran los capítulos junto a una breve descripción de cada uno.

1. **Introducción**

   Introduce la motivación y los objetivos principales del proyecto, así como otros objetivos de índole personal que deberán cumplirse durante el desarrollo.

2. **Erlang**

   Se introduce un caso particular de lenguaje concurrente basado en *paso de mensajes*.

3. **Trazas**

   Se proporciona una perspectiva genérica de las trazas con las que tratamos en nuestra aplicación.

4. **Especificación de requisitos**

   Se comenta en detalle el proceso de elicitación de requisitos y posteriormente se obtiene una especificación informal de los requisitos funcionales y no funcionales.

5. **Desarrollo e implementación**

   Se analizan las herramientas usadas para el trabajo, el *software* de nuestra aplicación, la metodología de desarrollo de *software* seguida y distintos manuales de usuario.

6. **Validación y pruebas**

   Se presentan los casos de prueba y los resultados para validar la calidad de nuestro *software*.

7. **Conclusiones y trabajo futuro**

   Se reflexiona sobre el trabajo realizado y se proponen mejoras a realizar en un futuro cercano.

2. Erlang

Erlang es un lenguaje de programación funcional concurrente. Fue creado para desarrollar aplicaciones distribuidas, tolerantes a fallos, y de funcionamiento ininterrumpido. Tal es el caso de aplicaciones como el chat de Facebook.

2.1. Estructura del lenguaje

Un programa en Erlang se divide en uno o más módulos. Cada módulo es un fichero que contiene una secuencia de atributos y funciones $f_1, \ldots, f_n$:

- Los atributos definen las propiedades del propio módulo.
- Una función `f` cuyo nombre es `fname` se declara construyendo una secuencia de cláusulas separadas con punto y coma. Cada cláusula consiste en una cabecera y un cuerpo:

```
fname(X1,…,Xn) [when expresión] -> expresión
```

  - La cabecera de una cláusula `fname(X1,…,Xn) [when expresión]` se forma con el nombre de la función, una lista de argumentos (patrón) y una guarda opcional, cuya expresión evalúa si los argumentos cumplen los requisitos para ejecutar el cuerpo de dicha cláusula.

  - El cuerpo de una cláusula contiene las acciones que se desean realizar. Es decir, una expresión de la forma $e_1, \ldots, e_n$, donde $e_1, \ldots, e_n$ son expresiones, $n \geq 0$.

Las cláusulas de una misma función comparten en su definición el nombre de la función. En cambio, dependiendo de los valores en la lista de argumentos en la llamada de la función, se ejecutará una cláusula u otra. Esta técnica de selección se llama *pattern matching* (4).

Para visualizarlo mejor, una función con dos cláusulas se declara de esta forma:

```
fname(X1,…,Xn) [when expresión] -> expresión;
fname(Y1,…,Yn) [when expresión] -> expresión.
```

O bien, así:

```
fname() ->
   {X1,…,Xn} [when expresión] -> expresión;
   {Y1,…,Yn} [when expresión] -> expresión.
```

Usaremos la primera alternativa sintáctica para el siguiente ejemplo de un programa escrito en Erlang que calcula el factorial de un número `N`:

```
factorial(N) when N == 0 -> 1;
factorial(N) when N > 0 -> N * factorial(N-1).
```

Queremos calcular el factorial de 1, por lo que llamamos a la función factorial con `N=1`, esto es, `factorial(1)`. Se comprobarán las cláusulas definidas de la función `factorial` de arriba a abajo:

- Paso 1: Se comprueba la guarda `N==0` de la primera cláusula; como `1` no es igual a 0, no se entra al cuerpo de dicha cláusula.
- Paso 2: `factorial(1)` hará *matching* con la cabecera de la segunda cláusula. La `N` tiene como valor `1`, la guarda `N>0` es cierta, por ende, entramos al cuerpo de la cláusula y se evalúa la expresión `1*factorial(0)`.
- Paso 3: El cuerpo de la segunda cláusula genera una nueva invocación de la función, `factorial(0)`, que hará *matching* con la primera cláusula, cuyo cuerpo devuelve un `1`.
- Paso 4: Volvemos a la acción del paso 2. Ya tenemos el valor de `factorial(0)`, por lo tanto, podemos calcular la multiplicación `1*1` que da `1`. Finalmente, sabemos que `factorial(1)` se evalúa a `1`.

Con respecto al arranque del programa, este se realiza llamando a la función inicial desde la consola: *nombre_módulo:nombre_función().*

2.2. Concurrencia en Erlang

Cuando hablamos de concurrencia, nos referimos a que distintas partes (subtareas) de un programa sean ejecutadas (potencialmente) en paralelo sin afectar el resultado final.

En Erlang, la concurrencia es explícita. Dicho de otro modo, el programador es quien crea los procesos manualmente. Además, a cada hilo de ejecución se le denomina proceso (5), ya que no comparten recursos. Un proceso no es más que una tarea (set de instrucciones) que puede ser ejecutada al mismo tiempo que otra tarea.

En muchas ocasiones, los procesos deben coordinarse y comunicarse entre sí. En Erlang, la comunicación es asíncrona y se basa en el *paso de mensajes*. A diferencia de la *memoria compartida* (el caso de *Java*), es menos propenso a fallos (6).

Cabe considerar, por otra parte, que Erlang sigue el modelo de actores: un proceso representa a un actor, esto es, puede crear más procesos (*spawn*), enviar mensajes y determinar cómo actuar con respecto a cada mensaje que reciba. Cada proceso posee un identificador único, conocido como PID (del inglés “Process Identifier”) y un buzón local (Figura 2.1) de capacidad ilimitada donde se depositan los mensajes que le llegan.

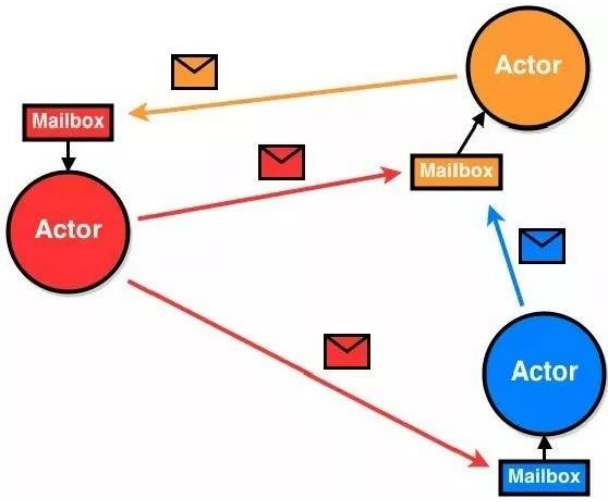

Figura 2.1. Modelo de actores

Los mensajes que se reciben son selectivos. Más exactamente, si un determinado mensaje del buzón no hace *matching* con ningún patrón contenido dentro del bloque *receive*, dicho mensaje permanece en el buzón y el proceso no lo recibe. En caso contrario, el mensaje se extrae del buzón y el proceso realiza la acción correspondiente.

Acerca de la lógica del procesamiento de mensajes, el bloque *receive* revisa el buzón de un proceso desde el primer mensaje hasta el último, secuencialmente, comparando cada mensaje con cada patrón. Cada vez que dicho proceso reciba un nuevo mensaje, se sigue la misma lógica.

2.3. Sintaxis del lenguaje

Una sentencia de Erlang no es más que una combinación de palabras y símbolos que su compilador puede reconocer. Entre estas combinaciones, nos vamos a centrar en el *paso de mensajes* entre procesos.

Para comenzar, introducimos las principales acciones concurrentes:

- *spawn*: Se crea un nuevo proceso que ejecutará una función de un módulo dado.
- *send*: Un proceso envía un mensaje a otro proceso.
- *delivery*: Un proceso recibe un mensaje en su buzón.
- *receive*: Un proceso extrae un mensaje de su buzón y lo consume mediante *pattern matching*.

Para entenderlo mejor, es esencial conocer la sentencia para crear un proceso. La función *spawn* crea el proceso y devuelve su identificador Pid. Dicho proceso empieza su ejecución en la función *nombre_función* del módulo *nombre_módulo* con los argumentos dados:

Pid = spawn(*nombre_módulo*, *nombre_función*, *lista_de_argumentos*)

La sentencia para enviar un mensaje al proceso que se identifica con Pid:

Pid ! *mensaje*

siendo el mensaje simplemente una expresión válida en Erlang.

De igual forma, cabe mencionar que aquellas expresiones que empiezan por una letra mayúscula o guion bajo se denominan variables (7). Las variables almacenan valores, así como la variable Pid almacena el identificador de un proceso.

A continuación, vamos a analizar brevemente el bloque *receive* (Figura 2.2), cuyo propósito es permitir a los procesos escuchar y seleccionar mensajes de su buzón local. En el siguiente fragmento de código podemos apreciar la posición del bloque *receive* dentro de la función *greet,* que contiene dos cláusulas:

```
greet() ->
      receive
          {hello, Sender} ->
                Sender ! {reply, self(), "Hi!"};

          {reply, Pid, String} ->
                io:put_chars(String)
      end.
```

Figura 2.2. El bloque *receive*

Donde la función *self()* es una función predefinida de Erlang, que devuelve el Pid del proceso que lo ejecuta.

Supongamos que tenemos dos procesos, P1 y P2. Nuestro objetivo es conseguir que P1 le transmita un saludo a P2 de esta forma: “Hello”, y P2 le responda con “Hi!”.

```
main() ->
    Pid1 = spawn(?MODULE, greet, []),
    Pid2 = spawn(?MODULE, greet, []),
    Pid2 ! {hello, self()},
    ok.
```

Figura 2.3. La función principal

Observando la función principal (Figura 2.3), el programa empieza creando los procesos P1 y P2 con identificadores Pid1 y Pid2, respectivamente. A su vez, P1 y P2 empiezan sus ejecuciones con la función *greet*. Es decir, cada vez que se añada un nuevo mensaje al buzón de mensajes de cada proceso, el bloque *receive* revisará cada mensaje del buzón correspondiente secuencialmente, en busca de algún mensaje que haga *matching* con el patrón de alguna de las dos cláusulas. Si no hay ningún mensaje que haga *matching* con algún patrón, el proceso se queda suspendido hasta recibir un nuevo mensaje.

Seguidamente, P1 envía a P2 el mensaje *{hello, Pid1}*, que se pone a la cola de mensajes del buzón de P2. Este mensaje en concreto hará *matching* con la primera cláusula, cuya acción correspondiente será, en este caso:

```
Pid1 ! {reply, Pid2, "Hi!"}
```

Ahora, como P1 también está ejecutando la función *greet*, el mensaje *{reply, Pid2, "Hi!"}* en su buzón será extraído, dado que hará *matching* con el patrón de la segunda cláusula. Y, finalmente, se llevará a cabo la acción correspondiente: P1 recibe el mensaje y lo escribe por consola (8).

En la Figura 2.4 se puede apreciar el comportamiento de este sencillo programa gráficamente.

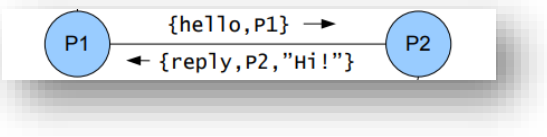

Figura 2.4. Flujo de ejecución

3. Trazas

En el área de desarrollo de *software*, la depuración consiste en la búsqueda y corrección de errores en programas, *software* o sistemas. Muchas veces nos podemos encontrar con programas que no funcionan de acuerdo con el propósito de su creación y sea necesario recurrir a la depuración. Si bien depurar un programa secuencial es relativamente sencillo, no resulta ser así en los programas concurrentes.

Sin lugar a duda, una de las principales desventajas de la programación concurrente reside en desconocer el orden en que se ejecutarán los procesos. Un programa concurrente puede exhibir un comportamiento inesperado a pesar de ejecutarse en un entorno de ejecución habitual, y raramente volver a presentar dichos fallos en las próximas ejecuciones, en virtud de la naturaleza dinámica de los procesos que lo conforman.

Pese a que existen técnicas y herramientas para la depuración reversible, esto es, seguir la traza de ejecución desde la aparición visible del problema hasta donde realmente se originó el *bug*, estas no ofrecen una forma de registrar una ejecución específica del programa, ya que no hay garantía de que la ejecución problemática vuelva a suceder en el depurador. Por ello, se han propuesto alternativas como la monitorización del estado del programa: *record and replay debugging*.

Como bien sabemos, un programa se puede ejecutar tantas veces como se desee. La propuesta anteriormente expuesta se basa en instrumentar un programa para que su ejecución genere trazas que luego se cargan en un depurador y nos permiten reproducir la ejecución que dio problemas. En otras palabras, generar una traza de una ejecución específica implica registrar en ficheros (logs) una serie de mensajes textuales que identifican el estado de las partes del programa y la línea de código que se estaba ejecutando en su momento, resolviendo así uno de los principales problemas de la depuración de programas concurrentes: la dificultad para reproducir una ejecución concreta debido al indeterminismo inherente a los programas concurrentes.

Toda reflexión se inscribe al protagonismo de las trazas en nuestra aplicación. Se plantea entonces definir la estructura del contenido de los logs válida para nuestra aplicación.

En primer lugar, conviene aclarar que nuestras trazas tienen el objetivo de registrar la comunicación que se da entre procesos a lo largo de la ejecución. Por ello usamos una traza basada en prefijos, esto es, introducimos etiquetas {*send*, *delivery*, *receive*} a los mensajes para identificar el emisor y el receptor de cada mensaje. Cabe mencionar que disponemos de tantos ficheros log como procesos haya, donde cada log registra los eventos `e` {*spawn, send*, *deliver*, *receive*} que surgen en el proceso correspondiente, en orden cronológico, de la forma {`e`, `n`}, donde `n` puede ser un PID (*spawn*) o un identificador de un mensaje (el resto). Además, se crea un fichero log adicional donde se registra el PID del primer proceso creado, llamado *trace_result*.

trace_result.log

…

{*pid*, 81}.

…

trace_81.log

{*spawn*, 82}.
{*send*, 0}.
{*deliver*, 1}.
{*receive*, 1}.
{*send*, 2}.
{*deliver*, 3}.
{*receive*, 3}.

trace_82.log

{*deliver*, 0}.
{*receive*, 0}.
{*send*, 1}.
{*deliver*, 2}.
{*send*, 3}.
{*receive*, 2}.

Figura 3.1. Tres ficheros log

Imaginemos un programa que ejecuta dos procesos con PIDs 81 y 82. El programa nos genera los tres ficheros log (versión simplificada) mostrados en la Figura 3.1, de los cuales obtendremos la traza mostrada en la Figura 3.2:

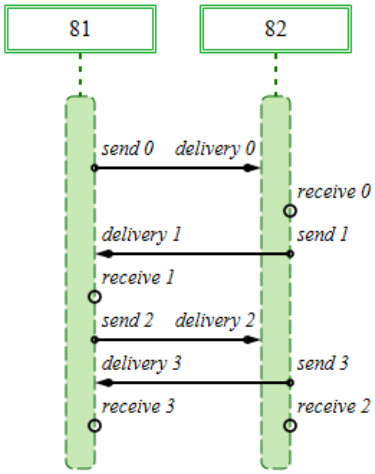

Figura 3.2. Ejemplo de traza

En este ejemplo de traza no tomamos en cuenta qué proceso crea qué proceso (*spawn*). Los procesos son representados verticalmente como un rectángulo discontinuo (el tiempo fluye de arriba a abajo). El envío y la entrega de mensajes se representan con una flecha sólida, desde el evento de envío (*send* `n`) hasta el evento de entrega (delivery `n`), `n = 0,…,3`. Los eventos de recepción se representan con un círculo y se denotan con *receive* `n`, `n = 0,…,3`. Obsérvese que no se ha fijado un orden global de los eventos, solamente se encuentran ordenados cronológicamente los eventos dentro de cada proceso, tal es el caso de la posterioridad del evento *{receive, 2}* con respecto al evento *{send, 3}*.

4. Especificación de requisitos

La Ingeniería de Requisitos (IR) es el área más importante de la Ingeniería de *Software* y posiblemente de todo el ciclo de vida del desarrollo de software. En esta fase se transforman los requisitos del *software* en especificaciones formales del aplicativo final.

4.1. Elicitación de requisitos

Para obtener los requisitos tenemos que empezar por la fase de elicitación de requisitos. Es una actividad basada en la comunicación humana entre el equipo de desarrollo y los *stakeholders* (personas interesadas por el producto y están afectados por él). Durante este proceso, queremos conocer en detalle el contexto del problema y las necesidades de los *stakeholders*.

En nuestro caso, el equipo de desarrollo está compuesto por un individuo. Y los *stakeholders* del proyecto del desarrollo son los desarrolladores de CauDEr (9).

Para nuestra aplicación en particular, los problemas a los que se están enfrentando y los objetivos fueron expuestos por videoconferencia entre el desarrollador y un representante de los *stakeholders*. En este caso, el objetivo más destacado es exponer una traza de pasos de mensajes entre procesos de un programa concurrente sobre una interfaz amigable.

4.2. Definición de requisitos

En esta sección se propone reflejar los requisitos funcionales y no funcionales de nuestra aplicación.

Con el fin de construir una buena ERS (Especificación de Requisitos *Software*) que pueda ser bien comprendida tanto por los desarrolladores como por los *stakeholders*, vamos a emplear un lenguaje natural preciso, apropiado y libre de ambigüedades para la definición de los requerimientos. Hemos escogido una técnica informal de especificación, que tiene como ventajas su sencillez, libre expresividad y prescindibilidad de entrenamiento especial. Seguiremos una serie de pautas al escribir nuestras definiciones:

1. La definición de un requisito solo debe incluir información esencial para su entendimiento.
2. La definición de un requisito debe ser clara y concisa.
3. Cada requisito tendrá un identificador único.
4. Los requisitos se clasifican mediante categorías (funcional, no funcional, diseño, bug, etc.).
5. Los requisitos son priorizables: esencial (1), condicional (2) y opcional (3).

Id	Descripción	Categoría	Prioridad
r01	El único panel de la aplicación debe ser redimensionable.	No funcional	2
r02	El botón de “Select a directory” debe abrir el explorador de archivos del dispositivo.	Funcional	1
r03	El botón de “Select a directory” no debe permitir seleccionar ficheros directamente, solo el directorio.	Funcional	2
r04	Si tras pulsar el botón de “Select a directory” se selecciona un directorio que no contiene los ficheros log requeridos, muestra una alerta de error.	Funcional	3
r05	La entrada de datos de la aplicación es un directorio que contiene todos los ficheros log relativos a los procesos de un programa.	No funcional	1
r06	Si la entrada de datos de la aplicación es correcta y completa en el panel principal debe aparecer un gráfico de trazas (Figura 3.2).	Funcional	1
r07	Todo gráfico debe cargarse en menos de 2 segundos en condiciones normales.	No funcional	3
r08	Se debe seguir y respetar los tamaños (de fuente, de figuras, etc.), tipo de tipografía y distancias entre elementos.	No funcional	2
r09	Si se ha apretado el botón de “Select a directory” y se quiere cancelar la selección, el contenido del panel no debe cambiar.	Funcional	3
r10	La aplicación debe ser multiplataforma. Como mínimo debe funcionar en entornos de Windows y Linux.	No funcional	2

5. Desarrollo e implementación

En la fase de análisis del problema hemos obtenido la especificación de los requisitos. A partir de esta, se estudian posibles opciones de implementación (fase de diseño) para el *software* que hay que construir. Seguidamente, pasamos a la fase de desarrollo e implementación, en el que se eligen las herramientas adecuadas, un entorno de desarrollo que facilite el trabajo y un lenguaje de programación.

Durante el desarrollo del producto, hemos seguido una metodología que se comentará en este capítulo. Adicionalmente, el producto *software* final se acompañará de un manual de usuario que facilitará la comprensión del modo de uso de la aplicación.

5.1. Tecnologías usadas

En esta sección hablaremos sobre las herramientas escogidas (GitHub) para el desarrollo de nuestra aplicación, así como la estructura del proyecto y los algoritmos implementados.

5.1.1. Git

El proceso de programación pasa por diversas fases, entre ellas, la fase de codificación, cuando se transforma el algoritmo previamente diseñado al ordenador a través de un lenguaje de programación, de modo que el algoritmo pasa a llamarse código fuente. Durante esa fase, los desarrolladores agregan o modifican líneas de código constantemente. Y en cuestión de segundos, el código fuente ha cambiado. O, lo que es lo mismo, surge una nueva versión del código.

En ocasiones, un cambio en el código puede conducir a errores que en la versión anterior no había. Por ello resulta conveniente disponer de alguna herramienta para gestionar y manipular los cambios de nuestro proyecto y, sobre todo, revertir (operación *rollback*) los cambios eliminando aquellas líneas de código problemáticas con un *click*.

Hoy en día, existe una herramienta ampliamente conocida y utilizada por todos los programadores: Git, un sistema distribuido de control de versiones (10). Su gran éxito se debe principalmente a que es rápido y ligero. Además, dispone de muchas herramientas, entre las cuales están las siguientes:

- *Branching*. Crea una rama en el repositorio con una copia independiente de todo el proyecto. Resulta útil para mantener muchas versiones del código en paralelo.
- *Merging*. Integra los cambios de una rama en otra rama.

Cabe destacar también algunas de las operaciones de Git más usadas:

- *Clone*. Cada programador puede descargarse una copia del código fuente desde un repositorio remoto de Git a su repositorio local.
- *Commit*. Los cambios locales del código de un programador se pueden registrar en su repositorio local, sin necesidad de tener acceso a Internet.
- *Pull*. Descarga la última versión del código en el repositorio remoto al repositorio local.
- *Push*. Carga los *commits* del repositorio local al repositorio remoto.

Por supuesto, Git también permite tener una copia del repositorio en un servidor central. Dado que Git es una herramienta de gran rendimiento, para nuestro proyecto optamos por usar el servidor central gratuito GitHub (11).

5.1.2. Organización de código fuente

La aplicación que implementamos y almacenamos en el repositorio GitHub consiste en un único proyecto organizado con carpetas separadas, cada una con un propósito:

1. Carpeta ***backend***

   Obtiene los ficheros log del dispositivo local y guarda los datos de entrada requeridos por la aplicación en estructuras de datos internos del código fuente.

2. Carpeta ***common***

   Se divide en dos archivos.
   En uno se guardan las configuraciones iniciales para la lógica del *backend*, como el nombre del primer fichero *trace_result.log*. En otro se guardan las configuraciones iniciales para dar estilo a la interfaz del *frontend*, como el radio del círculo para los eventos *receive* (Figura 3.2).

3. Carpeta ***frontend***

   En esta parte se implementan dos microservicios:

- Lee el fichero HTML en la carpeta *resources* y obtiene un nuevo HTML en texto plano a partir de este, con el contenido modificado.
- A partir de las estructuras de datos del *backend*, construye un algoritmo de cálculo de las coordenadas X e Y de cada elemento de la interfaz.

4. Carpeta ***resources***

   Contiene todos los archivos relacionados con el *frontend*, esto es, los ficheros HTML y CSS, y las imágenes (jpg, png).

5. Carpeta ***application***

   En esta carpeta reside la función principal que gestiona el *backend* y el *frontend*. Esta función también añade a la interfaz de usuario un botón de "Select a directory", que permitirá al usuario introducir los datos de entrada y ejecutar toda lógica de la aplicación.

5.1.3. *Backend* y *frontend*

Como ya mencionamos en la sección 5.1.2, nuestra aplicación se encuentra en un único proyecto. Elegimos crear un proyecto JavaFX (sección 5.1.4) con el fin de asegurar la compatibilidad con el *software* de los *stakeholders* y, además, por su simplicidad a la hora de dibujar los gráficos.

Naturalmente, toda la lógica *backend* de la aplicación se ha implementado en *Java*. Este lenguaje de programación se organiza jerárquicamente en paquetes. Cada paquete agrupa un conjunto de clases. Y en nuestro caso, cada clase es simplemente un fichero que contiene una secuencia de atributos de clase y funciones, similar al caso de Erlang (sección 2.1).

Nuestras estructuras de datos se definen como atributos de clase. Nosotros, en particular, solo usamos las siguientes estructuras de datos (12; 13):

- `Map<K, V> nombre` permite almacenar valores de tipo `V` e identificarlos con una clave de tipo `K`, donde cada elemento tiene la forma de un par (`clave`, `valor`). `K` y `V` pueden ser el mismo tipo de dato (cadena de caracteres, numérico, etc.). También tiene un `nombre` que identifica esta colección de datos, el cual nos permite acceder a sus elementos. Así pues, obtenemos un valor a través de su clave de esta forma: `nombre`[`clave`]. De ahora en adelante, la llamaremos *diccionario*.

- `ArrayList<T> nombre` es una lista genérica cuyos elementos son de tipo `T` (cadena de caracteres, numérico, etc.). Cada elemento de la lista tiene un

índice, de 0 a N, siendo N la talla de la lista. Accedemos a un elemento de esta forma: `nombre[indice]`.

Como es de suponer, cada atributo posee un `nombre` que lo identifica, es decir, un identificador.

Vamos a pasar a comentar el funcionamiento del algoritmo de nuestra aplicación paso a paso, usando el lenguaje natural. Como bien sabemos, la entrada de datos de nuestro visor de trazas es un conjunto de ficheros log que contienen los eventos de cada proceso en orden cronológico. El primer paso del algoritmo es la lectura de dichos ficheros para extraer la traza de eventos {*send*, *deliver*, *receive*} de cada proceso. Para verlo de manera sencilla, expondremos directamente los datos que se obtienen en la lógica del *backend*:

- Una lista de los PID (del inglés "Process Identifier") de los procesos, de la forma `List<String>` con identificador `pids`. Por lo tanto, también sabemos el número total de procesos, que coincide con la talla de la lista.

- Tres diccionarios de la forma `Map<Integer,String>` con identificadores `sendEvents`, `deliverEvents` y `receiveEvents`, que almacenan pares (`n`, `p`), donde `n` es el número de evento y `p` el proceso al que pertenece dicho evento, `n = 0,…,N`, siendo `N` el número total de mensajes enviados por todos los procesos.

- Un diccionario de la forma `Map<String,List<String>>` con identificador `sortedEvents`, que almacena pares (`p`, `eList`) donde `p` es el PID de un proceso y `eList` es la lista de eventos de cada proceso ordenada cronológicamente.

Por otro lado, para el *frontend* de la aplicación, usamos la herramienta WebViewer de JavaFX para cargar un HTML que se pasa en texto plano, que representa gráficamente la traza. Realmente, partimos de un fichero HTML incompleto al que le vamos añadiendo los elementos HTML mediante programación en *Java*. Todas las trazas tienen la apariencia similar a la de la Figura 3.2, esto es, tenemos el PID de cada proceso englobado en un rectángulo verde con doble borde, en disposición horizontal; un rectángulo con borde discontinuo debajo de cada PID, en disposición vertical, representando el ciclo de vida del proceso; una línea discontinua que separa los dos rectángulos de cada proceso, y el texto del evento junto a las flechas dirigidas de un proceso a otro. Todas estas figuras se dibujan en el panel principal de la aplicación siguiendo unos ejes de coordenadas, X e Y. Por lo tanto, es necesario calcular las coordenadas de cada figura de modo que muestren coherencia entre sí y no se solapen unas con otras. Si bien parece ser una cuestión complicada, lo cierto es que se puede resumir de manera sencilla de la siguiente forma:

- Como cada proceso se representa con dos rectángulos y una línea que los separa, definimos una coordenada (x, y) inicial para cada uno de estos tres elementos pertenecientes al proceso de más a la izquierda. Estos valores se guardan en un fichero del paquete *commons*.

- La anchura y la altura de cada tipo de elemento es la misma para todos los procesos.

- Los elementos que conforman los siguientes procesos se colocan a la derecha de este, en líneas paralelas, con una separación fija mediante la suma acumulativa de `x` unidades (por defecto, 136) a la coordenada X inicial. Por otro lado, las coordenadas Y permanecen igual.

- Creamos un diccionario de la forma `Map<String, Integer>` con identificador `xCoordinates`, que almacena pares (`p`, `x`) donde `p` es el PID de un proceso y `x` el valor de la coordenada X asignada a cada proceso.

En último lugar quedan las coordenadas de las flechas (eventos *send* y *delivery*) que van de un punto a otro y de los círculos (evento *receive*). Como bien sabemos, precisamos de las coordenadas (x_1, y_1) de origen y las coordenadas (x_2, y_2) de destino para cada flecha. Para los valores de x_1 y x_2 hacemos uso de las coordenadas X de cada proceso calculadas previamente. En cuanto a las coordenadas Y de cada evento, decidimos crear un algoritmo, haciendo uso de los datos obtenidos en el *backend* (`pids`, `sendEvents`, `deliverEvents`, `receiveEvents`, `sortedEvents`). Adicionalmente, utilizamos tres diccionarios auxiliares:

- `Map<String,Integer>` con identificador `lastY`, que almacena pares (`p`, `y`), donde `p` es el PID de un proceso y `y` la última coordenada Y dibujada en dicho proceso.

- `Map<String,Boolean>` con identificador `waitingProcess`, que almacena pares (`p`, `b`), donde `p` es el PID de un proceso y `b` un valor booleano (*true*, *false*). Si `b` es *true*, indica que los eventos de `p` se han dejado de revisar. En caso contrario (`b` es *false*), los eventos de `p` se siguen leyendo.

- `Map<String,Integer>` con identificador `processEventPointer`, que almacena pares (`p`, `i`), donde `p` es el PID de un proceso e `i` el índice en `sortedEvents[p]` del último evento leído de dicho proceso, inicialmente a cero.

Por simplicidad, omitiremos la mayor parte del uso del primer diccionario mencionado.

A continuación, explicaremos el algoritmo en pseudocódigo:

Para cada proceso `p ∈ pids`, (Bucle 1)

Para cada evento `e ∈ sortedEvents[p]`, (Bucle 2)

Si **`tipo`**`(e) == `*delivery* `->`

```
{
   waitingProcess[p] <- true
   eventPointer <- processEventPointer[p]
}
```

Si no, si **`tipo`**`(e) == `*send* `->`

```
{
```

Para cada proceso `p2 ∈ pids`, (Bucle 3)

Si `p == p2 ->`

```
{
   e2 <- sortedEvents[p2][eventPointer + 1]

   Si num(e) == num(e2) ->
   {
      Calcular y1 e y2 (1)
      waitingProcess[p2] <- false
      processEventPointer[p] += 1
      processEventPointer[p2] += 1
      Salir del bucle 3
   }
}
```

Si no, si `p ≠ p2 ->`

```
{
   eventPointer2 <- processEventPointer[p2]

   e2 <- sortedEvents[p2][eventPointer2]

   Si waitingProcess(p2) & num(e) == num(e2)->
   {
      Calcular y1 e y2 (2)
      waitingProcess[p2] <- false
      processEventPointer[p] += 1
      processEventPointer[p2] += 1
      Salir del bucle 3
   }
}
```

```
}
```

Si no, si **`tipo`**`(e) == `*receive* `->`

```
{
```

Calcular y (3)

```
processEventPointer[p] += 1
}
```

Nuestra coordenada inicial es `x=30` y `y=50`, la distancia horizontal entre dos figuras iguales es de `gapX = 136`, tal y como expusimos anteriormente. Por otro lado, la distancia vertical entre flechas consecutivas es de `gapY = 30`.

Guardaremos las coordenadas Y de los eventos en un diccionario de la forma `Map<String, List<Map<String,Integer>>>` con identificador `yCoordinates`, donde la clave es el PID de cada proceso y el valor es una lista cuyo único elemento es un diccionario que almacena pares de (`e`, `y`), donde `e` es el evento y `y` el valor de la coordenada Y del origen/destino de la flecha de dicho evento. Pongamos por caso que tenemos un proceso cuyo PID es 81, y tiene los eventos {*send 0*, *delivery 1*, *send 2*, *receive 1*}. Su entrada en el diccionario tendría la siguiente forma:

{ (81, [{(*send 0*, $\mathbf{y_1}$), (*delivery 1*, $\mathbf{y_2}$), (*send 2*, $\mathbf{y_3}$), (*receive 1*, $\mathbf{y_4}$)}]) }

Ahora que hemos introducido los conceptos necesarios para el algoritmo, volvemos al pseudocódigo y comentamos el cálculo de las coordenadas Y para cada caso:

El proceso emisor y el receptor de los eventos {*send* `n`, *delivery* `n`}, `n=0,…,N`, siendo `N` el número total de mensajes enviados por todos los procesos, …

(1) … son el mismo:

Si la coordenada y_1 del evento *send* es 50, la coordenada y_2 que corresponde al evento *delivery* será y_1 + `gapY`, es decir, y_2 = 80.

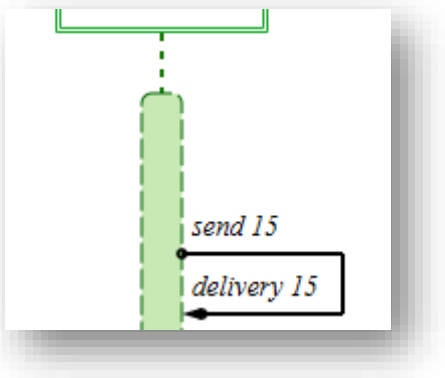

Figura 5.1. Una flecha hacia el mismo proceso

En este caso, este algoritmo no soporta un evento en medio de estos dos eventos, vemos que el {*delivery* `n`} se dibuja justo debajo del {*send* `n`}.

(2) … no son el mismo:

La coordenada y_1 del evento *send* y la coordenada y_2 del evento *delivery* son iguales.

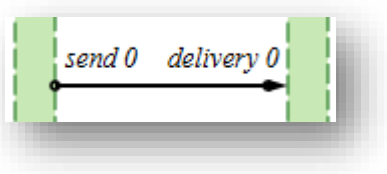

Figura 5.2. Una flecha de izquierda a derecha

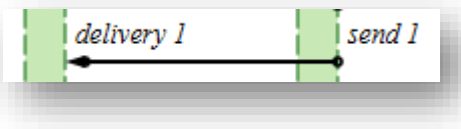

Figura 5.3. Una flecha de derecha a izquierda

Como en este trabajo los eventos no tienen marca de tiempo, este algoritmo tampoco toma en cuenta el instante temporal en que un mensaje se entrega al receptor. Se asume que se llegan a entregar de forma inmediata. Por lo tanto, en este caso, las flechas se dibujan sin ninguna inclinación.

(3) Es un evento *receive*:

La coordenada y_1 del evento *receive* se calcula con el valor de la última coordenada Y usada en el ciclo de vida del proceso, almacenado en `lastY`, sumado con la separación vertical fija entre figuras, es decir, y_1 = `lastY(p) + gapY`. La siguiente figura ilustra que dos eventos *receive* en procesos distintos pueden situarse a la misma altura.

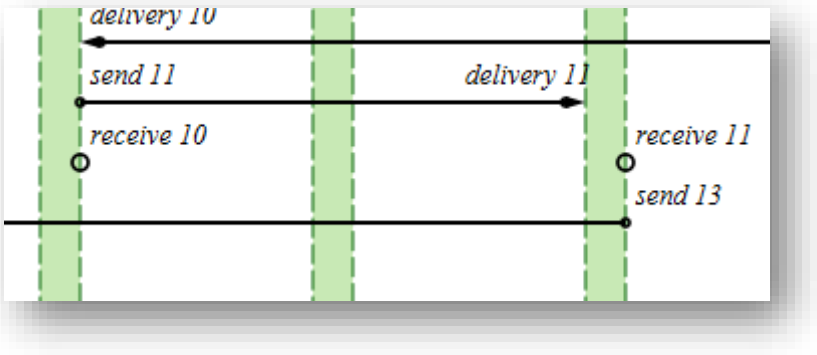

Figura 5.4. Dos círculos de eventos *receive*

Finalmente, queda por recalcar que el pseudocódigo mostrado se repite en un bucle infinito, con el fin de asegurar que todos los eventos hayan sido procesados. Asimismo, la condición de parada, que se sitúa al final del pseudocódigo, debe ser:

```
contador <- 0

Para cada proceso p ∈ pids,

  Si processEventPointer[p] ≠ talla(sortedEvents[p]) ->
    {
       Volver a empezar el bucle 1
       contador += 1
    }

Si contador == talla(pids) ->
  {
     Parar el algoritmo. Ya tenemos el yCoordinates bien rellenado.
  }
```

Sabemos que cada proceso tiene un puntero apuntando a sus eventos, desde el primer evento hasta el último. Cuando el puntero del proceso `p`, `processEventPointer[p]`, apunta al último evento, podemos concluir que todos los eventos de dicho proceso han sido procesados. Por ende, incrementamos el contador en una unidad. Si el valor del contador alcanza el número total de procesos, entonces aseguramos que todos los eventos de todos los procesos han sido procesados y el algoritmo puede terminar.

Significado de los símbolos en el pseudocódigo

Símbolo	Expresión	Significado
==	`A == B`	El valor de A es igual al valor de B.
≠	`A ≠ B`	El valor de A es no es igual al valor de B.
<-	`A <- B`	El valor de B se asigna a A.
+=	`A += B`	El valor de A se actualiza con la suma de los valores de A y B.
∈	`A ∈ B`	A pertenece al conjunto B.
&	`A & B`	Si A es cierto y B es cierto, la sentencia es verdadera (*true*). Si A y/o B es falso, la sentencia es falsa (*false*).

Significado de las funciones en el pseudocódigo

Función	Expresión	Significado
tipo	**tipo**`(e)`	Devuelve el tipo del evento `e`: *send*, *delivery* o *receive*.
num	**num**`(e)`	Devuelve el número `n` del evento. Un evento se define como: {*send* `n`}, {*delivery* `n`} o {*receive* `n`}
talla	**talla**`(C)`	Devuelve el número de elementos que contiene la colección de datos `C`.

5.1.4. JavaFX vs Swing

El deseo de integrar HTML en Java nos ha conducido a escoger una de estas dos herramientas para diseñar la interfaz de usuario.

Swing (14) es una librería de Java que proporciona una gran variedad de componentes orientados a la creación de una *Graphical User Interface* (GUI). Una *User Interface* (UI) es cualquier elemento visual que permite la interacción entre usuario y máquina. Mientras que una GUI se limita a mostrar gráficos, tales como iconos, menús, botones, etc. Similarmente, el conjunto de paquetes de gráficos de JavaFX (15) ofrece, en menor cantidad, componentes enriquecidos con una apariencia más avanzada y natural que en Swing.

Al principio, consideramos usar JEditorPane (16) de la librería Swing para cargar texto HTML en el panel principal de nuestro visor de trazas. JEditorPane no soporta HTML5 y CSS3 (17; 18) y como nuestro fichero HTML incluye un atributo de estilo avanzado imprescindible, finalmente optamos por la WebEngine de JavaFx.

Un gráfico de escena es el punto de partida para construir una aplicación JavaFX. Como se puede apreciar en la Figura 5.5, una escena forma parte de la capa superior de la arquitectura de JavaFX. Cada elemento situado en una escena se llama nodo y cada escena contiene una jerarquía de nodos.

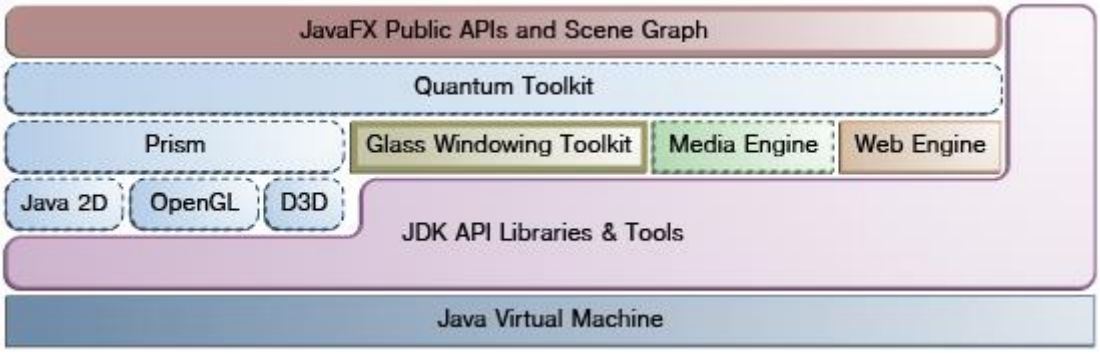

Figura 5.5. Diagrama de la arquitectura de JavaFX

El diseño de la interfaz de nuestra aplicación es simple. Está compuesto por un botón y un panel que muestra contenido HTML. Nuestra jerarquía de nodos se muestra en la Figura 5.6.

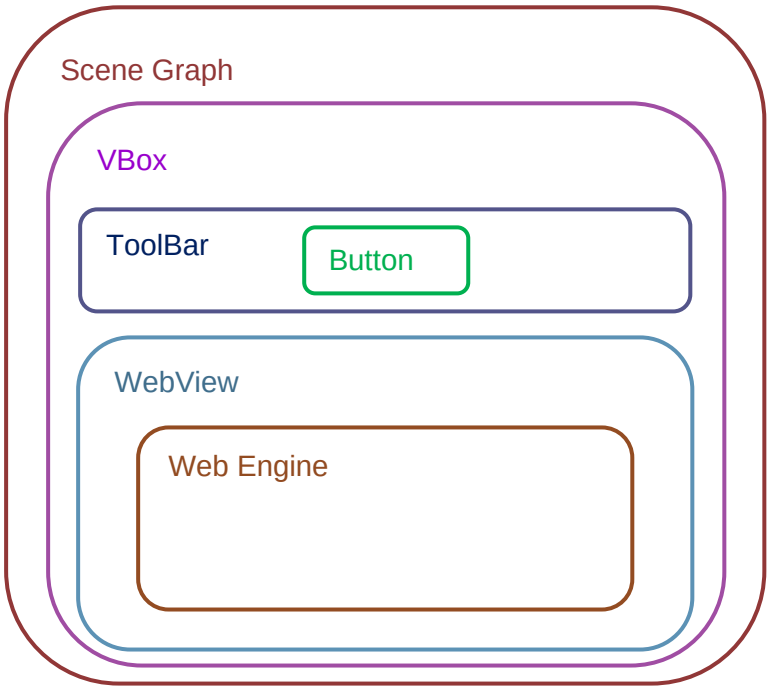

Figura 5.6. Jerarquía de nodos en nuestra escena

El componente que renderiza y presenta las trazas de ejecución que se dibujan con HTML es un componente web embebido en la escena y está compuesto por los nodos WebEngine y WebView. WebEngine provee la capacidad de navegar por una página web y es encapsulado por WebView, que incorpora el contenido HTML en la escena de nuestra aplicación, además de ofrecer campos y métodos para aplicar efectos y transformaciones.

5.2. Metodología

Una metodología de desarrollo de *software* es un marco de trabajo usado para estructurar, planificar y controlar el proceso de desarrollo del *software*. Su objetivo es la formalización de las actividades relacionadas con la elaboración de sistemas informáticos. Esto es, no especifica herramientas concretas, sino procesos, roles, tareas, *workflows*, etc. Cabe recalcar que no existe una metodología de *software* universal. Sin embargo, las actuales se pueden clasificar en tradicionales y ágiles.

Las metodologías tradicionales establecen que los proyectos se ejecutan en un ciclo secuencial, pasando por las fases de análisis, diseño, implementación, pruebas y mantenimiento. Lo que implica que los resultados del desarrollo no se ven hasta llegar a las últimas fases del ciclo de vida del proyecto (implementación y pruebas). Un ejemplo es *Rational Unified Process* (RUP) (19).

En cuanto a las metodologías ágiles, desarrollar *software* es la primera medida de progreso. Buscan satisfacer al cliente mediante entregas tempranas y continuas de *software* utilizable; asimismo, prioriza la obtención de un *software* que funciona con respecto a conseguir una buena documentación. Por mi experiencia en entornos profesionales, prácticamente todos los equipos de proyecto siguen la metodología ágil SCRUM (20).

Para nuestro proyecto se optó en primera instancia, por su simplicidad y por la falta de recursos y equipo (el equipo de proyecto está formado por una única persona), por la metodología tradicional: modelo *en cascada* (Figura 5.7), enfocado en el desarrollo de un único *mínimo producto viable* (MVP).

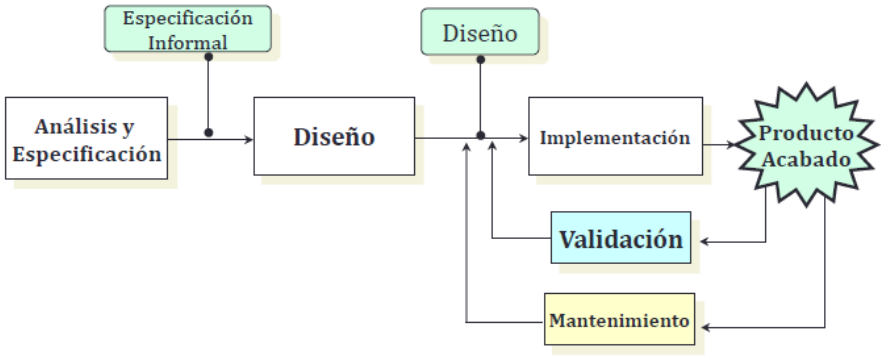

Figura 5.7. Modelo clásico o en cascada

No obstante, esta metodología limitaría el uso de prácticas y herramientas propias de las metodologías ágiles, tal es el caso de la atención continua a la calidad técnica y

al buen diseño. Por ello, tras un estudio de posibles metodologías del mundo del *software* para el desarrollo del producto, se siguieron, finalmente, las directrices de la metodología ágil *Personal Extreme Programming* (PXP) (21).

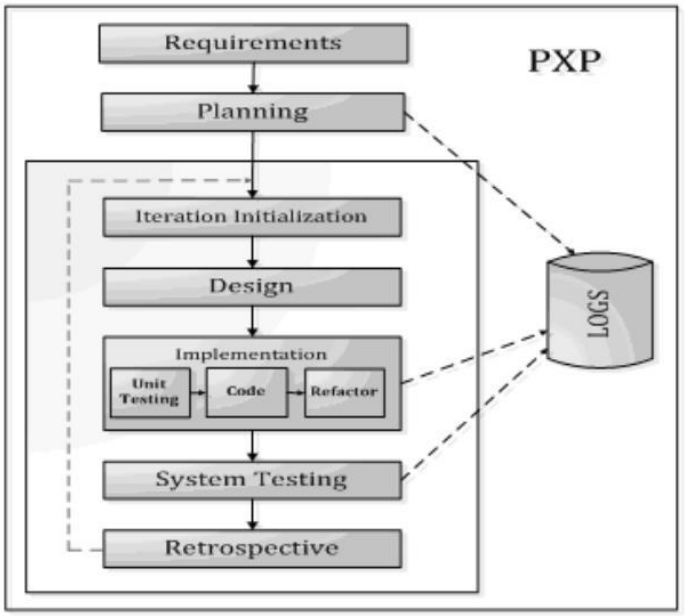

Figura 5.8. Fases de la metodología PXP

La metodología PXP se organiza con las fases que se ilustran en la Figura 5.8.

- Fase de requisitos. Es una fase opcional, en la que se genera un documento que reúne todos los requerimientos del sistema a desarrollar. En nuestro caso, obtuvimos una especificación informal de los requisitos (sección 4.2).

- Fase planificación. Se trata de identificar un conjunto de tareas a partir de los requisitos recién detectados. Cada tarea se puede dividir en subtareas y cada subtarea tiene asignada una estimación del tiempo que necesita el desarrollador para terminarla. Además, en esta fase se toman decisiones sobre las herramientas a utilizar (GitHub, JavaFX, HTML, etc.). Si posteriormente se modifican los requisitos, es posible adaptarse a una nueva planificación de tareas.

- Fase de inicio de la iteración. Una iteración empieza con la selección de una tarea. Para nuestro proyecto, una de las tareas fue programar la modificación del fichero HTML.

- Fase de diseño. Se modelan los módulos y clases del sistema, siguiendo los criterios del propio desarrollador, para su posterior implementación dentro de la iteración actual. Nosotros programamos con Java, por lo que el proyecto se

organiza en paquetes y clases. Para la tarea mencionada anteriormente, se han creado las clases nombradas UserInterface.java y Painter.java (dentro del paquete *frontend*, descrito en la sección 5.1.2), Algorithm.java e Initializer.java (dentro del paquete *algorithm*, que a su vez está dentro del paquete *frontend*).

- Fase de implementación. El desarrollador implementa todos los módulos definidos en la fase de diseño y realiza un proceso llamado *unit testing* para verificar el correcto funcionamiento de cada módulo implementado en la iteración actual. Si el código fuente compila sin errores y han pasado todos los tests unitarios (de cada función de nuestras clases), se pasa a la siguiente fase.

- Fase de validación del sistema. Se procede al *integration testing* para validar el funcionamiento correcto en conjunto de todas las características y funcionalidades implementadas hasta el momento. Se corrigen los fallos encontrados.

- Fase de retrospectiva. El desarrollador analiza los datos estadísticos recogidos de las fases anteriores, como el tiempo real empleado para finalizar la tarea de la iteración actual para evitar futuras subestimaciones o sobreestimaciones de tiempo en otras tareas. Esta fase finaliza cuando la versión actual del *software* cumple con todos los requisitos. En caso contrario, se vuelve a la fase de inicio de la iteración.

5.3. Distribución del *software*

El primer paso para empaquetar un proyecto JavaFX para su distribución consiste en la generación de un fichero .jar, esto es, un fichero especializado de Java que reúne las clases de Java con sus metadatos y recursos asociados. En nuestro caso, generamos el fichero JAR con la herramienta de Eclipse *Runnable Jar*.

Seguidamente, descargamos el JDK 17 (*Java Development Kit*, versión 17) y las librerías de JavaFX 18 para Linux y Windows, y los depositamos en el directorio donde se encuentra nuestro *app.jar*, como se puede apreciar en la Figura 5.12.

Llegado a este punto, necesitamos ejecutar *app.jar*. Para ello, se crearon scripts Bash y PowerShell para Linux y Windows respectivamente, que automatizarían el proceso de lanzamiento de *app.jar*. Adicionalmente, para Windows se usó el comando *ps2exe* para generar el ejecutable (Figura 5.11). El contenido de los *scripts* se muestra

en la Figura 5.9 y la Figura 5.10, donde se pasan la ruta del JDK y la librería de JavaFX como argumentos.

```
#!/bin/bash

chmod -R a+x *

./jdk17/bin/java

--module-path ./jfx18/lib/   --add-modules javafx.controls, javafx.fxml, javafx.web

-jar ./app.jar
```

Figura 5.9. Contenido del *script ThreadLoggerViewer.sh* para Linux

```
.\jdk17\bin\java.exe

--module-path ".\jfx18\lib\"    --add-modules javafx.controls, javafx.fxml, javafx.web

-jar .\app.jar
```

Figura 5.10. Contenido del *script launch.ps1* para Windows

```
Install-Module -Name ps2exe

ps2exe launch.ps1
```

Figura 5.11. Comandos en Windows para generar .exe

Debido a que este proceso se realiza manualmente cada vez que se crea un *tag* en GitHub, se decidió utilizar el servicio de GitHub Actions, para guardar las *releases* de Linux y Windows con sus correspondientes ejecutables listos para el usuario final.

GitHub Actions es una plataforma de integración y despliegue continuos (IC/DC) que permite automatizar un mapa de compilación, pruebas y despliegue para el *software* de un repositorio, permitiendo crear flujos de trabajo. El GitHub Actions de nuestro repositorio conserva la *action* creada con el fichero *multi_os_release.yml* (sección A.2). Este *action* automatiza la ejecución de dos tareas en dos máquinas virtuales (Ubuntu y Windows) siguiendo una secuencia de pasos para crear las *releases*:

1. Traer los ficheros del repositorio al directorio de trabajo en la máquina virtual.

2. Crear un directorio *ThreadLoggerViewer*. Copiar el *app.jar*, el directorio *resources* y el binario (solo para Windows) y pegarlos en el directorio creado.

3. Descargar de la web de Oracle el JDK comprimido versión 17 para procesadores x86 de 64 bits. Dependiendo de la máquina virtual se descarga el de Linux o Windows.

4. Descargar las librerías de JavaFX versión 18 de la misma forma que el JDK, desde la web de Gluon (22).

5. Descomprimir el JDK 17 y las librerías de JavaFX 18, y moverlos al directorio *ThreadLoggerViewer*.

6. Se crea el *script* para lanzar el *app.jar* (para Windows es una alternativa al binario) y se comprime el directorio *ThreadLoggerViewer*.

7. Se sube el fichero comprimido como *artifact* para que pueda ser descargado al finalizar la tarea.

5.4. Manual de usuario

Nuestro visor de trazas es una aplicación multiplataforma que ofrece dos versiones, una para Windows y otra para Linux.

5.4.1. Manuales específicos

5.4.1.1. Para usuarios de Windows

Paso 1: Ejecutar el .exe desde el explorador de archivos del dispositivo (Figura 5.12).

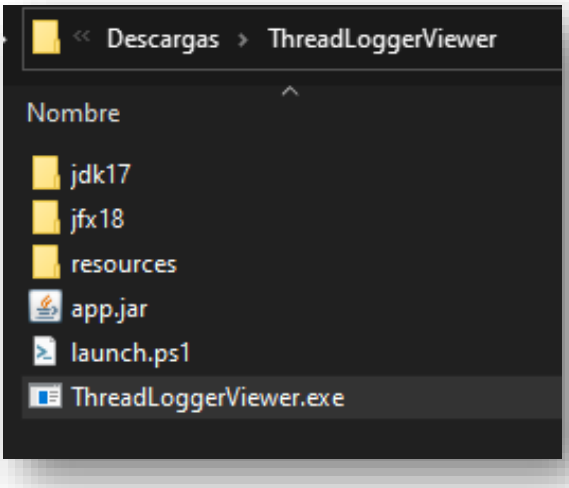

Figura 5.12. El ejecutable de la aplicación

Paso 2: Se abrirá el panel principal de la aplicación (Figura 5.13).

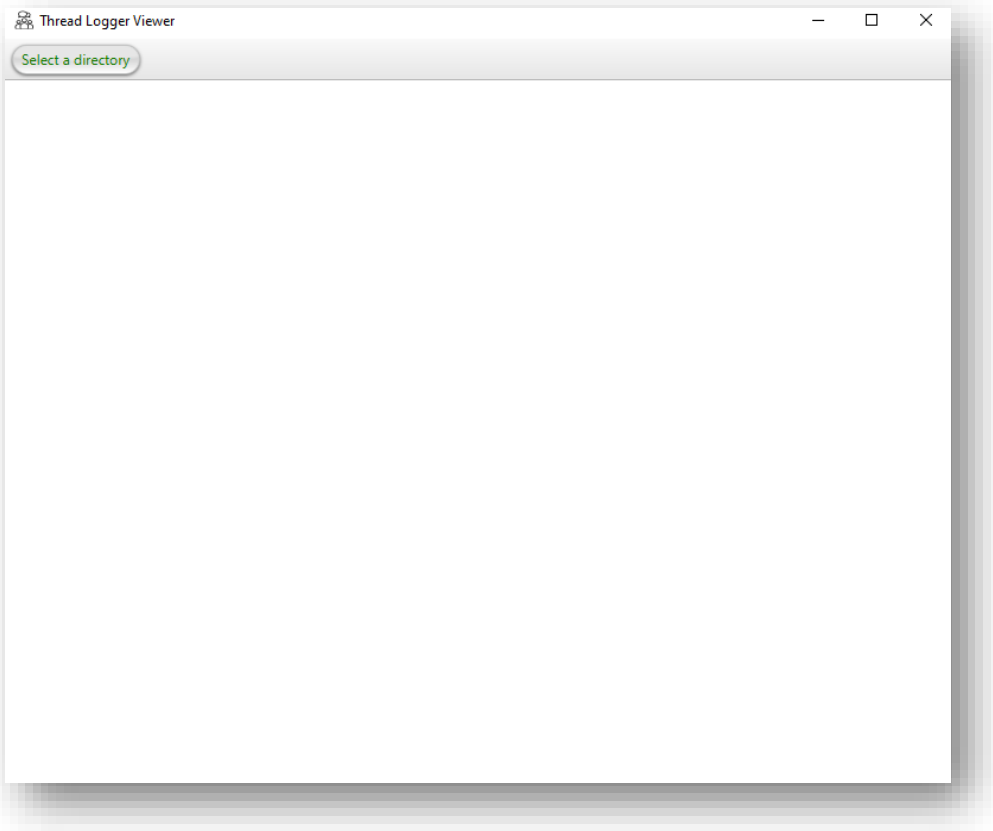

Figura 5.13. Panel principal de la aplicación

Paso 3: En la esquina superior izquierda hay un botón con el texto “Select a directory” (Figura 5.14). Tras pulsarlo, abre el explorador de archivos (Figura 5.15). Debe seleccionarse una carpeta que contenga los ficheros log del programa concurrente.

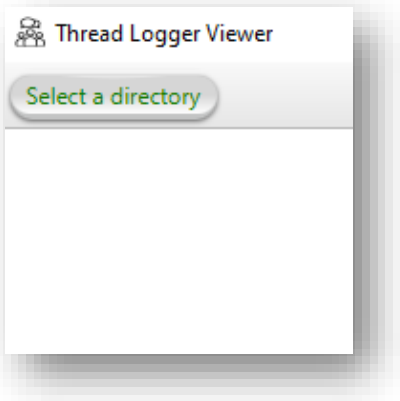

Figura 5.14. Botón pulsado

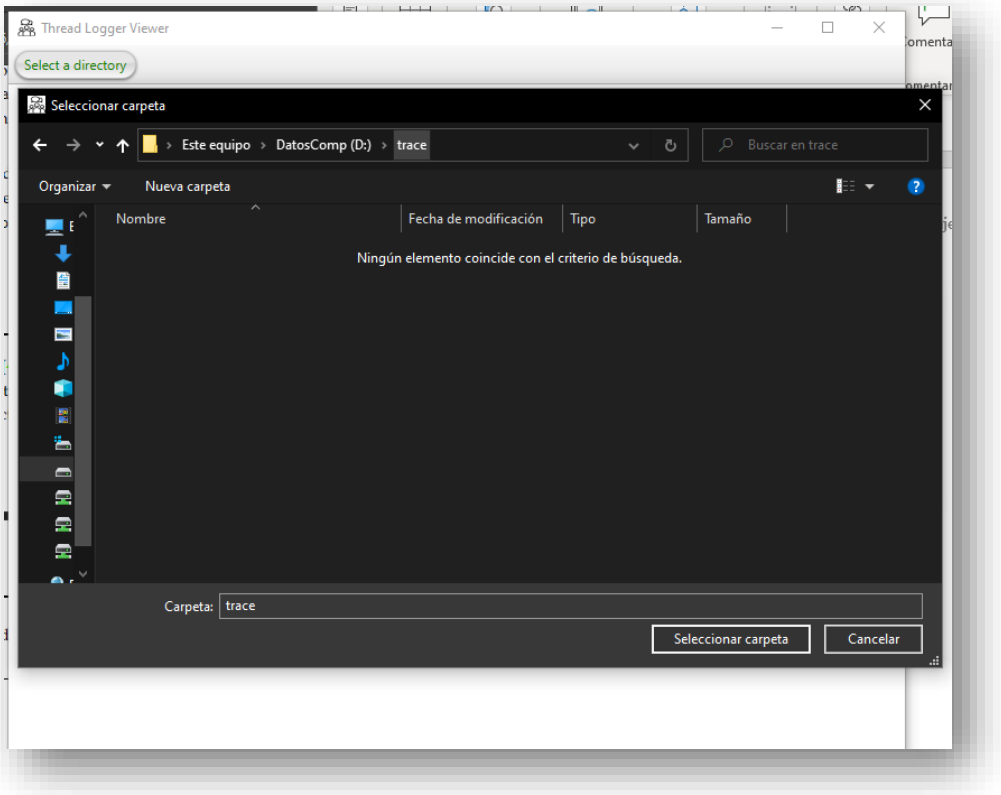

Figura 5.15. Explorador de archivos del dispositivo

Paso 4: Si la carpeta seleccionada contiene los ficheros log correctos, en el panel principal del visor debe aparecer una traza como en la Figura 5.16. El panel se puede redimensionar manualmente; si no, también se pueden usar los *scrollbars* vertical y horizontal para ver el contenido en su totalidad.

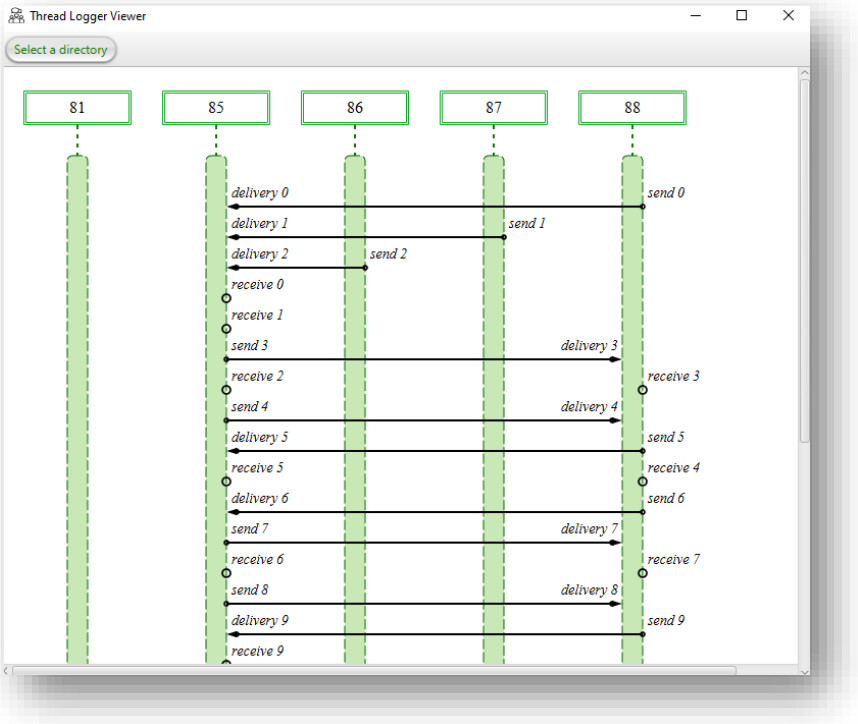

Figura 5.16. Una traza de ejemplo

5.4.1.2. Para usuarios de Linux

Paso 1: Abrir la consola de Linux, navegar hasta la carpeta (por defecto, *ThreadLoggerViewer*) que contiene el *script ThreadLoggerViewer.sh* y darle permisos de ejecución (Figura 5.17). En la Figura 5.18 se muestra el contenido de la carpeta donde se encuentra el *script* ejecutable.

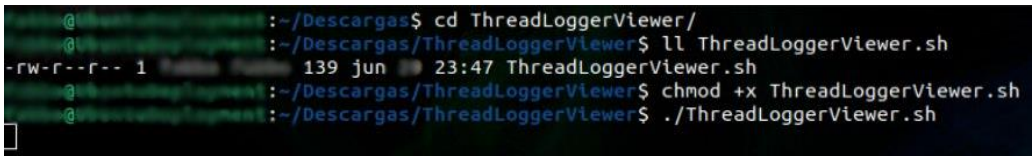

Figura 5.17. Comandos de ejecución en Linux

Figura 5.18. Contenido de la carpeta

Paso 2: Seguir el *Paso 2* del manual de usuarios de Windows (sección 5.4.1.1).

5.4.2. Manuales generales

5.4.2.1. Posibles alertas de errores

a) Si el directorio seleccionado no contiene todos los ficheros log requeridos o el nombre de dichos ficheros no sigue el formato especificado en la sección 5.4.2.2 (Figura 5.19).

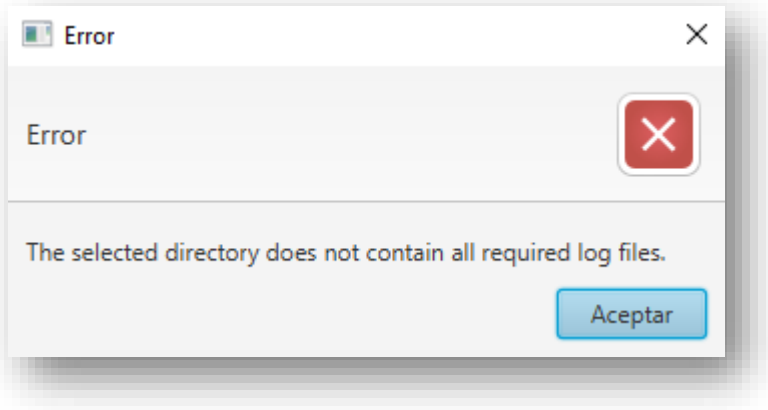

Figura 5.19. Alerta error nombre de los ficheros log

b) Si el contenido de los ficheros log no siguen el formato especificado en la sección 5.4.2.2 (Figura 5.20).

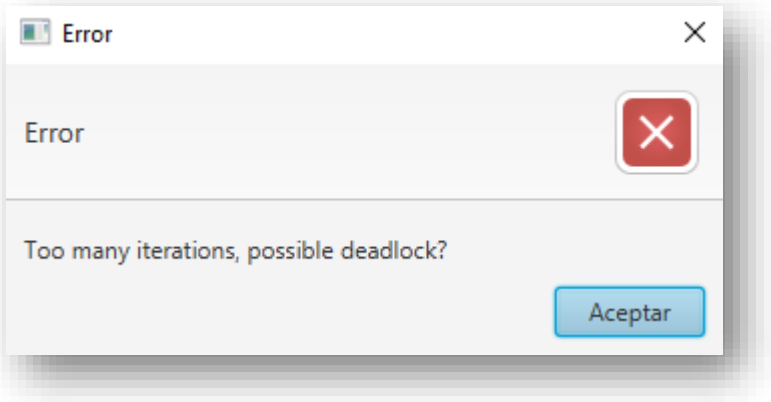

Figura 5.20. Alerta error contenido de los ficheros log

c) Si algún fichero en la carpeta *resources* está corrupto o ha sido modificado (Figura 5.21).

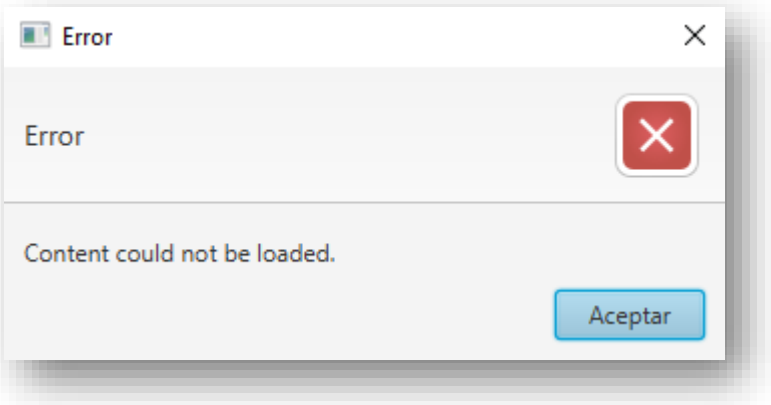

Figura 5.21. Alerta error ficheros de estilo (HTML, CSS, imágenes)

5.4.2.2. Formato del contenido

- Evento *spawn*: {spawn, {nonode@nohost, PID}, success}.
- Evento *send*: {send, n}.
- Evento *deliver*: {deliver, n}.
- Evento *receive*: {receive, n}.

Donde n es el número que identifica el mensaje.

1. PID es el identificador del proceso. Además, los ficheros log de cada proceso deben nombrarse *trace_PID.log*. Y el fichero principal *trace_result.log*.
2. Los eventos "spawn", "send", "deliver" y "receive" deben escribirse literalmente con estos caracteres en minúsculas.
3. Si se lee un evento, pongamos como ejemplo {*deliver*, 1}, deben existir los eventos {*send*,1} y {*receive*,1}.
4. Se permiten espacios en blanco entre símbolos, caracteres y números.
5. La cadena *nonode@nohost* se puede sustituir por cualquier otra combinación de caracteres y símbolos, pero no debe contener ningún dígito numérico ni espacios en blanco.

6. Validación y pruebas

En la ingeniería de *software*, la fase de prueba de *software* es una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y se registran, y se realiza una evaluación de algún aspecto: corrección, robustez, eficiencia, etc.

Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular, en nuestro caso, obtener una traza de ejecución gráficamente en el panel principal de nuestra aplicación. Así pues, en este capítulo vamos a exponer los distintos casos de prueba realizados tanto en Windows como en Linux para garantizar el desempeño correcto de nuestro programa.

Caso de prueba 1: Los datos de entrada son los logs de ejemplo obtenidos del repositorio de los *stakeholders* (9). Nuestro algoritmo procesa los datos y muestra la traza de la Figura 6.1. Adicionalmente, nuestra última versión del programa también muestra las coordenadas X e Y de cada evento por consola (Figura 6.2).

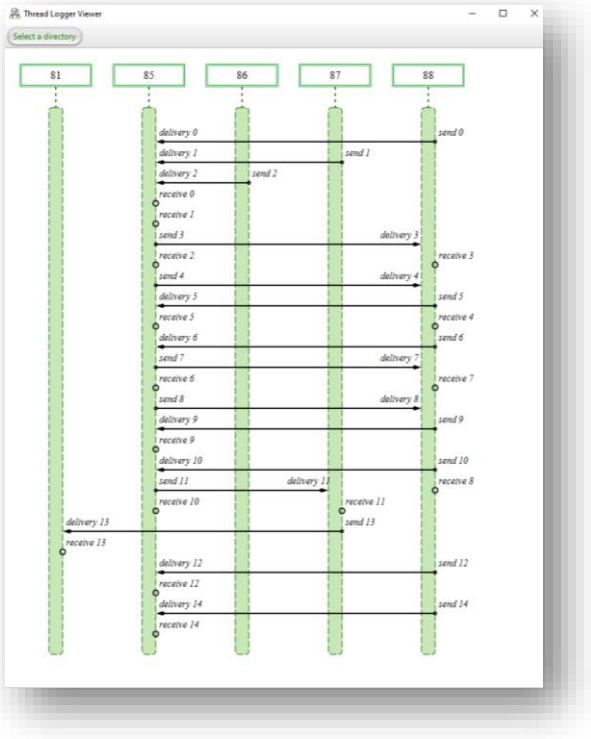

Figura 6.1. Primera traza de prueba (9)

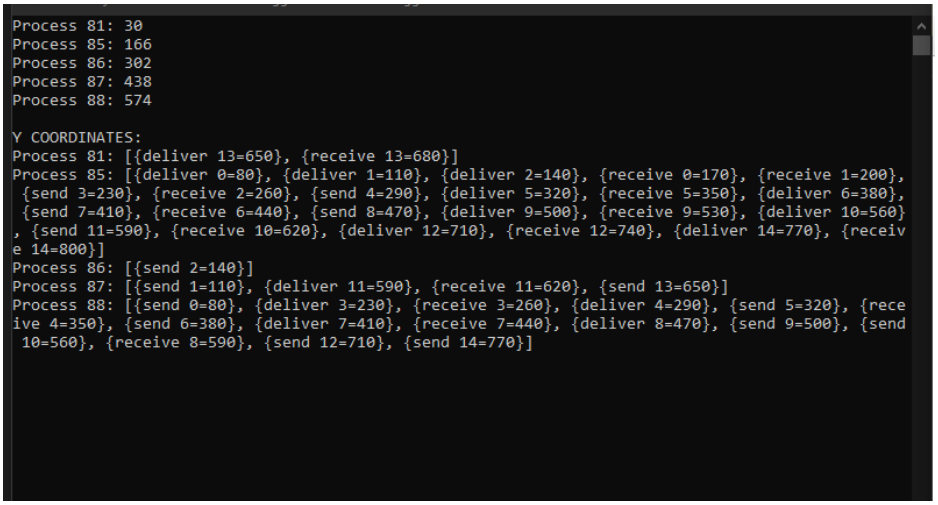

Figura 6.2. Coordenadas X e Y en la visualización de la traza

Caso de prueba 2: La entrada se basa en los logs del caso de prueba anterior; añadimos un nuevo proceso (con PID 89) que envía un mensaje a otro proceso para verificar que nuestro algoritmo es flexible y funciona para cualquier otra entrada, y modificamos el fichero log correspondiente al proceso 81, de modo que tiene un nuevo evento: el proceso se envía un mensaje a sí mismo (Figura 6.3). Recordemos que posteriormente el proceso 81 recibe un mensaje (número 13) de otro proceso (Figura 6.1). Nótese que a pesar de que el nuevo mensaje es el número 15, al ser el primer mensaje del proceso cronológicamente, se antepone al mensaje número 13.

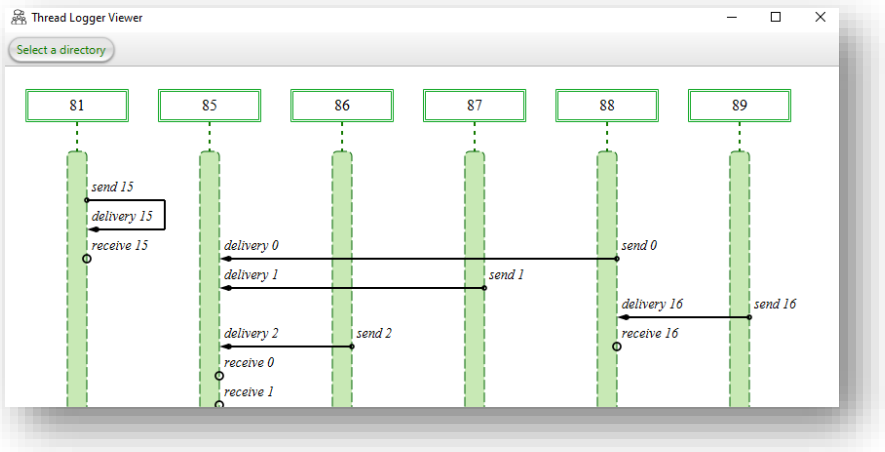

Figura 6.3. Segunda traza de prueba

Caso de prueba 3: La entrada son unos ficheros log con el contenido incorrecto, porque no siguen el formato especificado en la sección 5.4.2.2. En este caso, todos los logs tienen eventos *send* pero ningún evento *deliver* o *receive*, por lo que aparece la alerta de la Figura 6.4. La presencia de un mensaje en la consola demuestra que se ha llegado a leer y procesar los logs.

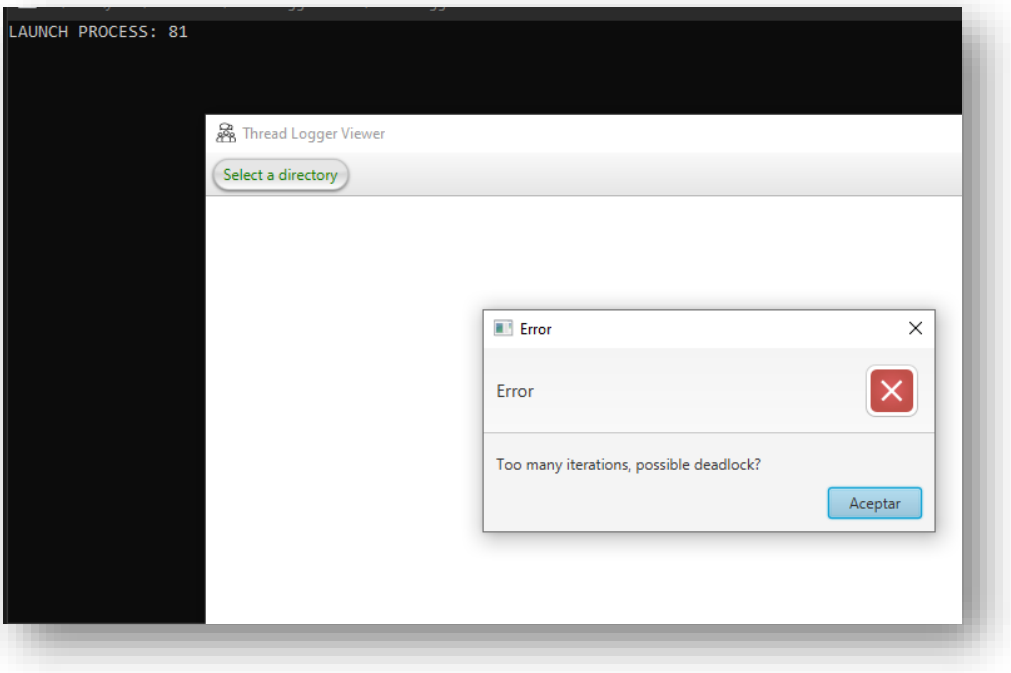

Figura 6.4. Aviso por contenido de logs incorrecto

Caso de prueba 4: En el caso de no haber ninguna entrada de datos, la aplicación no se cierra y su contenido visual sigue intacto. Se avisa con un mensaje al usuario de que no ha seleccionado ninguna carpeta del explorador de archivos (Figura 6.5) en la consola.

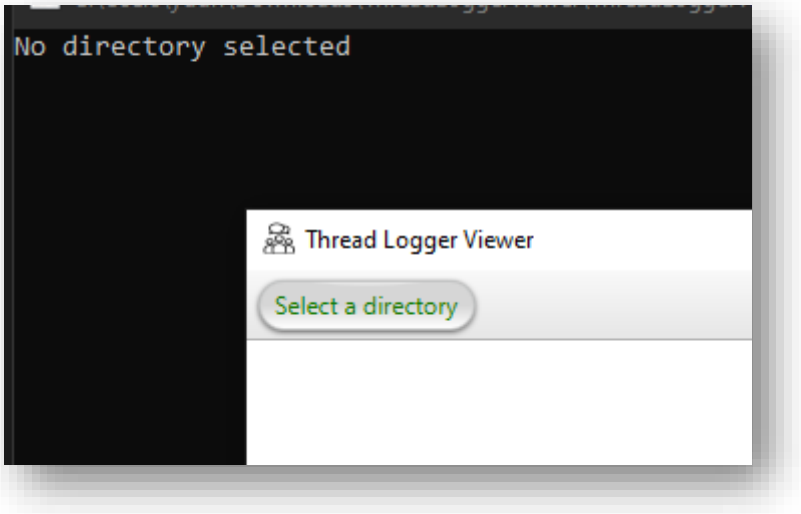

Figura 6.5. No se ha seleccionado ningún directorio

Cabe recalcar que las cuatro pruebas presentadas son unos *tests* de aceptación en los que los *stakeholders* validan la funcionalidad del *software* desarrollado, y cubren los casos habituales con los que el usuario final se puede encontrar.

7. Conclusiones y trabajo futuro

Realizando una retrospectiva de los objetivos marcados en el inicio del proyecto, puede señalarse sin lugar a duda que se ha cumplido el objetivo fundamental: Desarrollar un MVP de un visor de trazas de ejecución a partir de una entrada de datos cargada por el usuario final.

Pese a que estamos tratando con una aplicación que implementa funcionalidades simples, tales como el procesamiento textual de los ficheros y la generación de nuevos datos a partir de su contenido, y la lógica del *backend* no fue considerablemente costosa de estructurar, lo cierto es que se ha tenido que estudiar y profundizar el campo de los desarrolladores de *frontend*, ya que no se tenía una experiencia profesional en el diseño de la parte visual de un *software*. En consecuencia, nuestro código fuente sufría de muchas revisiones y rectificaciones, por lo que finalmente guardamos nuestro proyecto en un repositorio de GitHub para controlar las distintas versiones del código.

El estudio sobre la creación y diseño de una interfaz de usuario lo enfocamos sobre todo al lenguaje de marcado de hipertexto o HTML, que habitualmente se usa para elaborar páginas web, integrado en diversos *frameworks* de Javascript especializados para diseñar interfaces de usuario, como Angular, React y VueJS. Concretamente, nosotros integramos HTML en Java mediante la herramienta web que nos proporciona JavaFX. Con el fin de aplicar un aspecto más realista a nuestros elementos HTML, resultaba imprescindible incorporar estilos mediante CSS y estudiar el gran abanico de opciones que ofrece.

Desde la perspectiva técnica, no nos limitamos a que nuestro programa funcione, sino que también aplicamos la programación modular, las buenas prácticas de añadir *javadocs* en nuestro código y el arte popular conocido como *clean code*.

Por último, cabe mencionar que en este trabajo nos centramos principalmente en el cumplimiento de los requisitos acordados con los usuarios finales de la aplicación. Sin embargo, durante todo el proceso, se han encontrado una serie de características interesantes a incluir.

Nuestra aplicación de escritorio lee los logs de cada proceso y muestra un gráfico de las trazas de ejecución siguiendo el algoritmo descrito en este documento. Se aprecia el orden cronológico de los eventos de cada proceso individualmente; no obstante, carece de una perspectiva global del tiempo real en que ocurre cada evento. Tras una larga reflexión, consideramos que sería interesante cambiar el formato de los logs, de modo que cada evento incluya una marca de tiempo del momento que se genera. De esta forma, la visualización reflejaría el paso de mensajes entre procesos de una manera más realista.

Por otro lado, la conversión de nuestra aplicación de escritorio a una aplicación web sería un gran aspecto de mejora, ya que hoy en día la mayoría de las aplicaciones de su misma categoría se encuentra disponibles tanto en versión escritorio como en versión web.

Bibliografía

1. *JI.FI: Visual test and debug queries for hard real-time.* **Blanton, Ethan, y otros.** 14, 2013, Concurrency and Computation: Practice and Experience, Vol. 26, págs. 2456-2487.

2. **Christian H Bischof; .** *Parallel computing : architectures, algorithms and applications.* s.l. : Jülich, Germany : John von Neumann Institute for Computing;, 2008.

3. **Britton, T, y otros.** *Reversible Debugging software - Quantify the time and cost saved using reversible debuggers.* [En línea] 2012. http://www.roguewave.com.

4. **Erlang.** Erlang Reference Manual. [En línea] 2003-2022. https://www.erlang.org/doc/reference_manual/patterns.html.

5. **WikiLibros.** *Erlang.* [En línea] 20 de 08 de 2014. https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Erlang/Implementaci%C3%B3n_de_corutinas.

6. **Programmer 97-things.** Message Passing Leads to Better Scalability in Parallel Systems. [En línea] 25 de 11 de 2009. https://web.archive.org/web/20150106000512/http://programmer.97things.oreilly.com/wiki/index.php/Message_Passing_Leads_to_Better_Scalability_in_Parallel_Systems.

7. **Wikipedia.** *Variable (programación).* [En línea] https://es.wikipedia.org/wiki/Variable_(programaci%C3%B3n).

8. **Erlang.** *Erlang STDLIB version 4.0.1.* [En línea] 1997-2022. https://www.erlang.org/doc/man/io.html.

9. **Lanese, Ivan, y otros.** CauDEr. [En línea] https://github.com/mistupv/cauder/tree/paper/message-races/case-studies/barber/trace.

10. **Torvalds, Linus.** Git. [En línea] 27 de 06 de 2022. https://git-scm.com/.

11. **Microsoft Corporation.** GitHub. [En línea] 10 de 04 de 2008. https://github.com/.

12. **Java Platform Standard Ed. 8.** *Map.* [En línea] https://docs.oracle.com/javase/8/docs/api/java/util/Map.html.

13. —. *ArrayList.* [En línea] https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html.

14. **Jaiswal, Sonoo.** JavaTpoint. [En línea] 2011. https://www.javatpoint.com/java-swing.

15. **Oracle.** JavaFX. [En línea] 2008-2014. https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784.

16. **Mohtashim, Mohammad.** Tutorials point. [En línea] 01 de 01 de 2006. https://www.tutorialspoint.com/swingexamples/using_jeditorpane_to_show_html.htm.

17. **Stack Exchange, Inc.** StackOverflow. [En línea] 10 de 08 de 2014. https://stackoverflow.com/questions/4117302/swing-jeditorpane-css-capabilities.

18. **Mora Luján, Sergio.** HTML5 y CSS3. [En línea] 2012. http://desarrolloweb.dlsi.ua.es/cursos/2011/html5-css3/.

19. **Rational Software Corporation.** University of Houston: Clear Lake. [En línea] 05 de 2002. https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/.

20. **Wikipedia.** *SCRUM.* [En línea] https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software).

21. **Dzhurov, Yani, Krasteva, Iva y Ilieva, Sylvia.** *Personal Extreme Programming - An Agile Process for Autonomous Developers.* [En línea] http://bg-openaire.eu/bitstream/10867/28/1/S3T2009_37_YDzhurov_IKrasteva_SIlieva.pdf.

22. **GluonHQ.** Gluon. [En línea] 2022. https://gluonhq.com/products/javafx/.

Apéndice

Adjuntamos fragmentos de código auxiliares que hemos creado a lo largo del desarrollo del proyecto y un enlace al repositorio GitHub.

A.1. Algoritmo de cálculo de las coordenadas Y

El algoritmo ha pasado por varias versiones. Su primera versión fue implementada con el lenguaje de programación Python.

```
def algoritmo(processes_dict: dict):
    # Espacio entre cada coordenada Y
   Y_AXIS_SPACE=10
    num_process = len(processes_dict.keys())
    y_axis_process = {}
    last_y_axis_process = {}
    waiting_process = {}
    last_event_process = {}
    for key in processes_dict:
        y_axis_process[key] = []
        last_y_axis_process[key] = 0
        waiting_process[key] = False
        last_event_process[key] = 0
```

Ilustración 1. Condición de parada del algoritmo

```
while True:
    for process in processes_dict.keys():
        if last_event_process[process] >= len(processes_dict[process]): continue
        event = processes_dict[process][last_event_process[process]]
        if waiting_process[process]: continue
        if event["event"] == "deliver" and not waiting_process[process]:
            waiting_process[process] = True
            continue
        elif event["event"] == "send":
            send_proc = process
            for dvr_proc in processes_dict.keys():
                if last_event_process[dvr_proc] >= len(processes_dict[dvr_proc]): continue
                last_event = processes_dict[dvr_proc][last_event_process[dvr_proc]]
                if send_proc != dvr_proc and waiting_process[dvr_proc] and last_event["code"] == event["code"]:
                    y_to_use = max(last_y_axis_process[send_proc], last_y_axis_process[dvr_proc]) + Y_AXIS_SPACE
                    y_axis_process[send_proc].append( (y_to_use, event["event"]+event["code"]) )
                    last_y_axis_process[send_proc] = y_to_use

                    y_axis_process[dvr_proc].append( (y_to_use, last_event["event"]+event["code"]) )
                    last_y_axis_process[dvr_proc] = y_to_use
                    waiting_process[dvr_proc] = False
                    last_event_process[send_proc] += 1
                    last_event_process[dvr_proc] += 1
                    break
        elif event["event"] == "receive":
            y_to_use = last_y_axis_process[process] + Y_AXIS_SPACE
            y_axis_process[process].append(
                (y_to_use, event["event"]+event["code"])
            )
            last_y_axis_process[process] = y_to_use
            last_event_process[process] += 1
```

Ilustración 2. Cálculo de las coordenadas Y

```
cont = 0
        for process in processes_dict.keys():
            if last_event_process[process] != len(processes_dict[process]):
                break
            cont += 1
        if cont == num_process:
                return y_axis_process
```

Ilustración 3. Condición de parada del algoritmo

A.2. Enlace al repositorio GitHub

Se dirige al contenido del fichero *multi_os_release.yml*:

https://github.com/Wyuuri/ThreadLoggerViewer/blob/main/.github/workflows/multi_os_release.yml

A.3. Objetivos de Desarrollo Sostenible

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

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

Reflexión sobre la relación del TFG con los ODS más relacionados

Cada vez somos más conscientes de la importancia de la sostenibilidad. Definitivamente, la sostenibilidad se ha convertido en uno de los principales aspectos a tener en cuenta en áreas de negocio, gobiernos, fundaciones caritativas y, por supuesto, en la industria del *software*. Nosotros estamos explorando el mundo del *software*; particularmente, nuestro trabajo está relacionado, en menor o mayor medida, con los siguientes objetivos de desarrollo sostenible:

- **ODS 4. Educación de calidad:** Proporcionar un instrumento específico para facilitar la comprensión a los informáticos (desde alumnos a seniors) sobre el comportamiento interno de un programa puede ser una gran contribución a la mejora de la calidad del *software* y el aprendizaje permanente.

- **ODS 8. Trabajo decente y crecimiento económico:** Un estudio reciente afirma que el coste de la depuración de *software* se eleva a 312 billones de dólares anuales, de los cuales la mitad se invierte en los salarios. Pese a que depurar un programa concurrente tiene cierta complejidad, este proceso se podría "automatizar" con una versión más formal de nuestra aplicación. La automatización de este proceso tiene un estrecho vínculo con la administración económica del mundo empresarial informático, donde hay una diversidad de roles y puestos de trabajo. Por lo tanto, implicaría un efecto positivo en el crecimiento económico general de una organización.

- **ODS 9. Industria, innovación e infraestructuras:** La depuración de programas concurrentes es un problema de complejidad elevada. La formalización de una aplicación para visualizar trazas de ejecución en una interfaz amigable para su futura distribución en la industria del *software* sería una gran innovación.

- **ODS 11. Ciudades y comunidades sostenibles:** Con este trabajo se busca la optimización del ciclo de vida del desarrollo de *software*, en concreto, el tiempo y los recursos que se invierten en la fase de validación y pruebas. En este sentido, existe una relación directa con la sostenibilidad de comunidades y ciudades, enfocada en la aceleración del desarrollo de servicios informáticos y su distribución para el uso comunitario.

- **ODS 12. Producción y consumo responsables:** El fundamento del trabajo reside en la representación gráfica de las interacciones entre procesos de un programa concurrente. La posibilidad de visualización de las trazas de ejecución de un programa concurrente desde una perspectiva realista, como alternativa a los depuradores, evita un consumo de electricidad excesivo y disminuye el desgaste de los componentes electrónicos del ordenador cuando se quiere encontrar la raíz de un posible fallo en el programa.

- **ODS 17. Alianzas para lograr objetivos:** El tema desarrollado en este trabajo es de especial interés para los investigadores de proyectos relacionados con la trazabilidad de programas concurrentes, que puede vincularse de manera indirecta con el desarrollo de alianzas internacionales entre investigadores, empresarios y consumidores. Este proceso estaría determinado por la confluencia de los intereses de estos actores en cuanto a la creación y la mejora de sistemas de tiempo real robustos y eficientes.