Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Reconocimiento de widgets automático para aplicaciones Java/Swing en TESTAR

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Pastor Ricós, Fernando

Tutora: Vos, Tanja Ernestina

Cotutora: Esparcia Alcazar, Anna Isabel

Curso 2016-2017

Resumen

El grupo de Software Testing and Quality (STaQ) del centro de investigación PROS de la Universidad Politécnica de Valencia (UPV) ha desarrollado una herramienta, denominada TESTAR (www.testar.org) para el testing automatizada a nivel de Interfaz de Usuario (IU). TESTAR genera y ejecuta casos de prueba automáticamente basado en un modelo de árbol derivado automáticamente desde la IU de la aplicación bajo prueba. Este árbol es construido con ayuda del API de accesibilidad del sistema operativo que ayuda a reconocer todos los elementos gráficos de la IU (widgets). La herramienta no es del tipo capturar/reproducir ni usa reconocimientos de formas. Las empresas que han desplegado la herramienta son muy optimistas y lo ven como un cambio de paradigma del testing a largo plazo, pues tiene potencial para resolver muchos de los problemas de las herramientas existentes.

En este proyecto el objetivo es extender e implementar la capacidad de reconocimiento de widgets (elementos gráficos de Interfaz de Usuario) de la herramienta TESTAR para aplicaciones Java en sistemas operativos Microsoft Windows. Esta herramienta facilita el testeo automático de aplicaciones software desde su Interfaz de Usuario (IU), pero actualmente se ha identificado una limitación en el reconocimiento de widgets cuando la tecnología Java utilizada es Swing, funcionando sin problemas para AWT y SWT.

TESTAR se sustenta en tecnologías de accesibilidad que exponen los widgets de las aplicaciones software. El carácter "lightweight" en la implementación de Swing provoca que los widgets Swing no sean identificados por tecnologías de accesiblidad. Para apoyar la accesbilidad de aplicaciones Swing existe un puente denominado Java Access Bridge que expone el API de accesibilidad de Java en una librería dinámica (DLL) de Windows: http://www.oracle.com/technetwork/articles/javase/index-jsp-136191.html

Por tanto, la labor del proyecto será:

- Estudiar el puente Java Access Bridge para facilitar el reconocimiento de widgets de manera automática en aplicaciones Java/Swing existentes.

- Implementar un plug-in para TESTAR, que dote a la herramienta de reconocimiento de widgets en Swing que permitirá automatizar la prueba de software bajo tecnología Swing, además del soporte actual a las tecnologías AWT y SWT.

- Evaluar la capacidad de TESTAR en Java/Swing con dos casos prácticos con aplicaciones en empresas. (En este momento EVERIS y Clearone son empresas que han mostrado interés en tener disponible esta capacidad en TESTAR).

- Documentar los resultados.

Palabras clave: testeo automatizado, aplicaciones java/swing, interfaz de usuario

Abstract

The Software Testing and Quality (STAQ) group of the PROS research center at the Polytechnic University of Valencia (UPV) has developed a tool, called TESTAR (www.testar.org) for automated testing at the user interface level (UI) . TESTAR generates and executes test cases automatically based on a tree model automatically derived from the UI of the application under test. This tree is built using the Accessibility API of the operating system that helps to recognize all graphical UI elements (widgets). The tool is not capture / replay nor uses image recognition. Companies that have deployed the tool are very positive and see it as a paradigm shift for testing. They believe that TESTAR has the potential to solve many problems with existing tools.

In this project the aim is to extend and implement the recognizability of widgets (graphical elements of the User Interface) of the TESTAR tool for Java applications in Microsoft Windows operating systems. TESTAR has a limitation regarding the recognition of widgets when the Java technology Swing is used (it runs smoothly for AWT and SWT).

TESTAR is based on accessibility technologies that expose widgets of the software application under test. The "lightweight" character of Swing makes that some Swing elements are not correctly identified by accessibility technologies . To support the application accesbilidad for Swing there is a bridge called Java Access Bridge exposes the Java Accessibility API in a dynamic link library (DLL) for Windows: http://www.oracle.com/technetwork/articles/javase/index-jsp-136191.html

Therefore, the work of the project will be:

• Study the Java Access Bridge bridge to facilitate recognition of widgets automatically in Java / Swing applications.

• Implement a plug-in for TESTAR to enrich the tool with recognition of Swing widgets, in addition to the current support for AWT and SWT technologies.

• Assess the capacity of TESTAR in Java / Swing with two case studies with industrial applications. (Currently EVERIS and Clearone are companies that have shown interest in having this capacity available in TESTAR).

• Document the results.

Keywords: automated testing, Java Swing applications, User Interface

Tabla de contenidos

1. Introducción....4
1.1. Motivación....4
1.2. Objetivos....4
2. TESTAR....5
2.1. Introducción y funcionamiento de la aplicación....5
2.2. Problemas encontrados con la tecnología Java Swing....9
2.3 Biblioteca gráfica Java Swing....10
2.4. Máquina Virtual Java....12
3. Java Accessibility....14
3.1. Java Accessibility API....15
3.2. Java Accessibility Utilities....15
3.3. Arquitectura *Look and Feel*....16
3.4. Java Access Bridge....16
3.5. Tecnologías para ejecutar los recursos Java de una aplicación en la misma JVM que TESTAR....20
4. Uso de las tecnologías investigadas....23
4.1. Reconocer componentes de una JVM diferente a TESTAR....23
4.2. Reconocer componentes existentes en la misma JVM que TESTAR....24
5. Implementaciones....25
5.1. Arquitectura general de TESTAR....25
5.2. Implementación Swing Reflection....28
5.3. Implementación Java Access Bridge....34
6. Diferencias entre ambas implementaciones....37
6.1. Evaluación inicial empleando ambas implementaciones....37
6.2. Diferencias y selección final de una implementación....39
7. Validación....42
7.1. Validación empleando aplicaciones Java Swing....42

8. Conclusiones..48
8.1. Resumen del trabajo realizado..48
8.2. Futuras investigaciones..49
8.3. Aprendizaje personal en la realización del trabajo...50
9. Bibliografía..51

1. Introducción

1.1. Motivación

El presente trabajo de fin de grado se basa en la aplicación informática TESTAR ([1]), una herramienta de testeo software[1] que realiza las pruebas a través de la interfaz de usuario. El propósito de este proyecto es avanzar un paso más hacía la meta de convertir la herramienta TESTAR en un referente en testeo de software. Para ello se pretende eliminar un problema detectado en la versión actual, consistente en un fallo de funcionamiento cuando se utiliza la aplicación TESTAR conjuntamente con la tecnología Java Swing. Este trabajo será por tanto un ejemplo de actualización software para obtener un funcionamiento adecuado y estable, resolviendo el problema encontrado.

La elección personal de este trabajo se basa en obtener la experiencia de implicarse en un caso real que requiere la necesidad de investigación, estudio y solución, respecto a los problemas que pueden surgir en las aplicaciones informáticas. Además, cabe mencionar el interés en esta herramienta por parte de algunas empresas como EVERIS, con la cual se han mantenido reuniones en la sucursal de Valencia, debido al uso de la tecnología Java Swing en sus interfaces gráficas de usuario. Por todo ello, se tiene la intención de obtener una actualización diseñada específicamente para la resolución de dicho error, además de evitar modificaciones innecesarias que puedan causar grandes cambios en la programación base de la aplicación.

1.2. Objetivos

El objetivo general de este proyecto consiste en realizar en la herramienta TESTAR un conjunto de implementaciones necesarias basadas en la incorporación de clases y librerías, que le permita un correcto funcionamiento con la tecnología Java Swing. Para ello se debe conocer dónde reside el problema, investigar en profundidad la tecnología que provoca un fallo de funcionalidad en TESTAR y su forma de interactuar con la herramienta.

En un principio hay que conocer y presentar la aplicación TESTAR, examinando su configuración actual para el testeo de programas a través de la interfaz. Y a medida que se vayan descubriendo y estudiando tecnologías relacionadas con una posible resolución de este problema, se debe elaborar un estudio con su respectiva documentación[2].

1 Pruebas de software: http://www.economist.com/node/10789417
https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

2 Por su ayuda en los consejos dados, fundamentales para enfocar la resolución de los problemas encontrados en TESTAR, quisiera dar las gracias a Urko Rueda Molina, miembro investigador de la herramienta TESTAR.

2. TESTAR

2.1. Introducción y funcionamiento de la aplicación

2.1.1 Presentación

TESTAR es un software de código abierto destinado a testear de forma automatizada, a través de las interfaces gráficas de usuario (*Graphical User Interface,* GUI)[3], las aplicaciones de escritorio, web y móvil ([6]). Es una herramienta en constante desarrollo, actualmente preparada para trabajar en los principales sistemas operativos como Windows, Linux, Mac OS o Android.

TESTAR es un resultado del proyecto del 7º Programa Marco (FP7) europeo Future Internet Testing (FITTEST), que se desarrolló desde 2010 hasta 2013 con la participación, entre otras entidades, de la Universitat Politècnica de València y la University College de Londres.

El propósito del proyecto FITTEST era el desarrollo de pruebas de automatización y testeo que se deberán realizar en las futuras aplicaciones conectadas a Internet, debido al incremento y dependencia de interconexiones complejas y críticas en sectores como el gobierno, finanzas, negocios, entretenimiento o servicios públicos y sociales. Todo ello ayudará a cumplir y asegurar la alta calidad que se demanda actualmente y en un futuro.

Desde 2014 TESTAR ha sido desplegada y empleada en diversas empresas obteniendo unos resultados que muestran un gran potencial por parte de esta herramienta, pudiendo llegar a convertirse en una aplicación referente para las compañías a la hora de realizar un testeo a nivel de GUI ([2],[3],[4],[5]).

2.1.2 Plataforma de trabajo

La aplicación TESTAR está escrita en Java, y su funcionamiento varía dependiendo del sistema operativo en el que es ejecutada. A lo largo del presente proyecto se trabaja concretamente en una máquina virtual que incluye el sistema operativo Windows 7, empleando el software VirtualBox; por este motivo se utilizará una extensión a TESTAR diseñada específicamente para este sistema, que implementa la realización de llamadas al API de Windows mediante el uso de la tecnología *Java Native Interface* (JNI)[4].

3 *Graphical User Interface*: Componente de una aplicación informática que el usuario visualiza gráficamente, y a través de la cual opera con ella. Está formada por ventanas, botones, menús e iconos, entre otros elementos.

4 *Java Native Inteface*: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html#wp9502

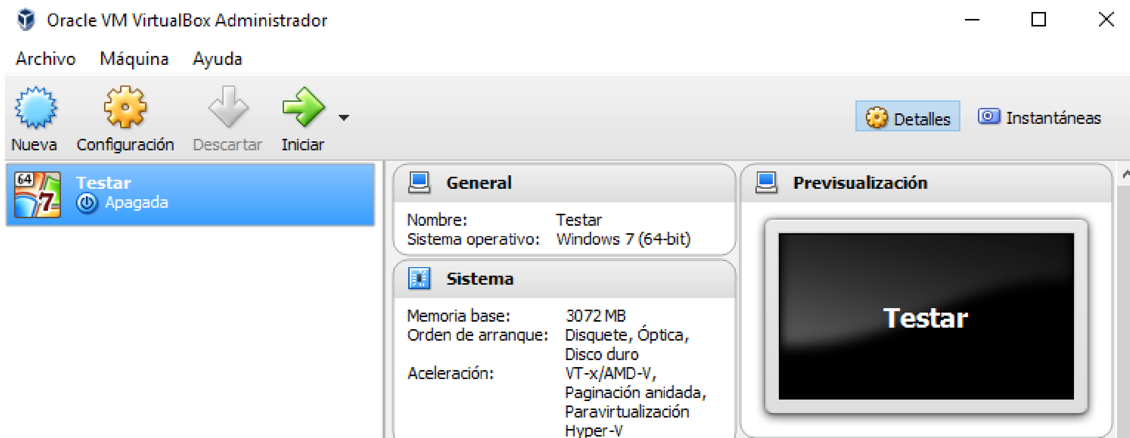

Figura 2-1: Máquina virtual Windows donde se realiza el trabajo dentro de la aplicación VirtualBox.

2.1.3 Funcionamiento general

TESTAR ejecuta de manera automática test secuenciales siguiendo los pasos indicados en el diagrama de la Figura 2-2.

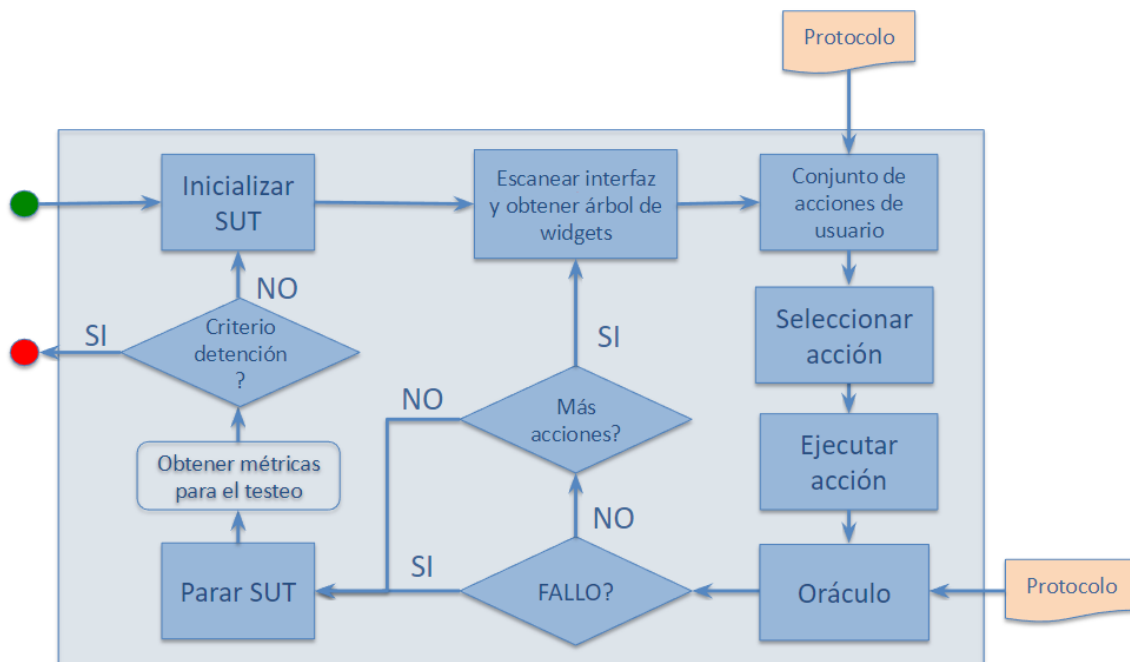

Figura 2-2: Diagrama que representa de forma secuencial la funcionalidad de TESTAR.

- Iniciar el sistema a testear (*System Under Test*, SUT) con la opción de incluir instrumentación adicional (ej. información interna necesaria para guiar la búsqueda).

- Escanear la interfaz de usuario para obtener el árbol de widgets que se encuentran actualmente en pantalla.

- Obtener el conjunto de acciones asociados a los widgets (clics, uso de teclas, arrastrar y soltar elementos, etc.).

- Seleccionar y ejecutar una de las posibles acciones, donde además, se puede condicionar la acción sobre cierto widget omitiendo la interacción sobre este.

- Repetir la obtención del árbol de widgets, la selección de acciones y su ejecución, en caso de no obtener fallos y hasta que se alcance la longitud acciones deseada.

- Parar el testeo en caso de obtener un fallo o alcanzar la longitud de acciones deseada. El SUT es detenido, y las métricas generadas hasta dicho momento son almacenadas.

- Comenzar otra iteración iniciando de nuevo el SUT o finalizar la herramienta, dependiendo de los criterios de detención señalados.

2.1.4 Reconocimiento de componentes

La funcionalidad programada que emplea TESTAR en el reconocimiento de componentes es parecida a la herramienta de inspección de interfaces incluida en el kit de desarrollo (*software development kit,* SDK)[5] de Windows. Emplea la API de accesibilidad a nivel de sistema operativo para el reconocimiento, recopilación de información y realización de acciones, sobre los diferentes componentes de las interfaces que se encuentren en ejecución.

Las aplicaciones al realizar la invocación de sus respectivas interfaces de usuario crean en el entorno del sistema operativo todos sus componentes mediante la realización de llamadas a un conjunto de librerías nativas de interfaz gráfica, definiendo estos componentes como un identificador numérico. Este tipo de componente basado en librerías nativas se conoce como *heavyweight component* (en castellano, componente pesado).

Al trabajar con las diferentes API de accesibilidad de los sistemas operativos, el reconocimiento de componentes en TESTAR varía según el sistema en el cual se esté ejecutando. En este trabajo en concreto se trabaja con la versión TESTAR para Windows, por lo que se hace uso del *framework* UI Automation[6], una tecnología ofrecida por Windows diseñada específicamente para los desarrolladores de software dedicado al testeo de aplicaciones.

Las llamadas realizadas desde TESTAR al entorno del sistema operativo se basa en el uso de la tecnología JNI. Para ello, se incluye en la herramienta una clase basada en el lenguaje de programación C++, denominada concretamente “main.cpp”. Estas llamadas JNI permiten buscar en el sistema operativo los distintos componentes pesados que se encuentren activos haciendo uso de la API de accesibilidad de Windows (UI Automation en este caso), entendiendo como activos aquellos que han sido invocados por su aplicación y están siendo mostrados en la interfaz.

5 Inspect SDK: https://msdn.microsoft.com/en-us/library/windows/desktop/dd318521(v=vs.85).aspx

6 UI Automation: https://msdn.microsoft.com/en-us/library/windows/desktop/ee684009(v=vs.85).aspx

Existen diferentes formas de emplear y obtener los componentes de las interfaces empleando la tecnología UI Automation[7], entre ellas: escuchar los posibles eventos que se realizan en las interfaces del sistema, acceder al elemento de una posición (x,y) concreta del escritorio o interfaz de usuario, o acceder a los elementos “hijos” de una ventana de aplicación concreta.

La herramienta TESTAR emplea esta última forma de interacción con los componentes de la interfaz, accediendo a todos los elementos que descienden de la ventana principal de la aplicación a testear (las ventanas o *hwnd* se identifican como un long). Para ello, se obtienen todas las ventanas existentes que descienden de la ventana escritorio, y se comprueba en ellas si el identificador de su proceso asociado coincide con el *PID* proceso de la aplicación a testear.

Para aclarar este último comentario, cabe añadir que TESTAR incluye la posibilidad de introducir en sus parámetros de entrada, la interacción con los ejecutables de la aplicación objetivo a testear o con sus procesos ya activos en ejecución. Sin embargo, la forma de comunicación final es siempre la misma, ya que si se introduce un ejecutable, TESTAR lo ejecuta y almacena el *PID* del nuevo proceso ejecutado.

2.1.5 Creación del árbol de widgets

El árbol de widgets, mencionado en el funcionamiento general de TESTAR, es una representación superpuesta a la interfaz de usuario de la aplicación a testear, que crea la herramienta en su entorno de ejecución. De esta forma, consigue asociar los diferentes componentes reconocidos de las interfaces de usuario a un conjunto de objetos denominados widgets, que son creados y declarados según las propiedades obtenidas de los componentes de la interfaz, gracias a las tecnologías de accesibilidad. Este árbol de widgets se crea de manera iterativa en TESTAR, para reconocer los diferentes componentes que se encuentran activos en cada instante, y en los cuales, la herramienta puede realizar un conjunto de acciones de usuario.

En la Figura 2-3 se puede observar la información que ofrece TESTAR para los widgets creados en base a los diferentes componentes, gracias al modo *Spy.* Esta funcionalidad de la herramienta permite conocer los diferentes componentes con los cuales es posible interactuar y sus respectivas propiedades, además de desactivar la realización de acciones sobre estos en caso de tener la intención de dirigir el recorrido del testeo. Las características *Role* y *Title*, junto al identificador numérico del widget, son las básicas que se deben implementar y obtener por cada widget creado en base a los componentes mostrados de la interfaz de usuario.

7 Obtaining UI Automation Elements: https://msdn.microsoft.com/en-us/library/windows/desktop/ee671590(v=vs.85).aspx

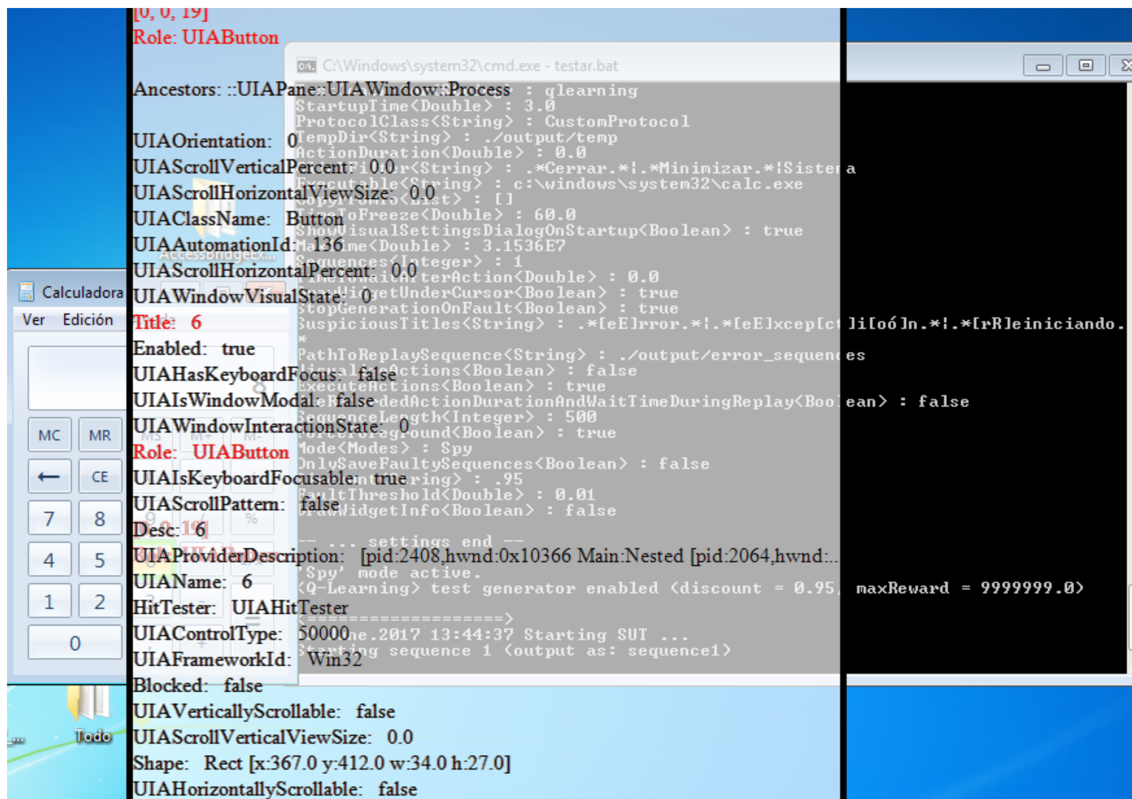

Figura 2-3: Conjunto de propiedades de un componente reconocible por TESTAR, que se encuentra visible en la interfaz de usuario de la aplicación.

2.2. Problemas encontrados con la tecnología Java Swing

El problema encontrado, y en cuya resolución se centrará este trabajo de fin de grado, está relacionado con la funcionalidad de reconocimiento que emplea TESTAR para detectar y obtener los componentes presentes en las interfaces de usuario de las aplicaciones, basado en la accesibilidad de componentes creados a nivel de sistema operativo.

En la ejecución de pruebas de testeo realizadas por TESTAR en ciertas empresas, que emplean para la programación de sus interfaces la biblioteca gráfica de Java llamada Swing, se detectó que dichos componentes Swing no eran añadidos al árbol de widgets por parte de la herramienta TESTAR. A diferencia de Swing, los componentes de la herramienta gráfica de Java llamada AWT (en castellano, Kit de Herramientas de Ventana Abstracta)[8], son detectados correctamente e incluidos en el árbol de widgets.

El reconocimiento actual basado en el tipo de accesibilidad que ofrece UI Automation para los sistemas operativos Windows, u otra tecnología de accesibilidad de otro sistema operativo, sólo sirve para trabajar con tecnologías de interfaces gráficas que realicen una creación de sus componentes en el entorno del sistema (como puede ser la tecnología Java AWT). Por lo tanto, se debe investigar cómo funcionan las tecnologías de interfaces gráficas no basadas en la creación de componentes pesados en el sistema operativo, para implementar en la herramienta TESTAR un conjunto de modificaciones que permitan el reconocimiento de estos componentes.

8 AWT: http://docs.oracle.com/javase/1.5.0/docs/guide/awt/

En la Figura 2-4 se puede observar a la izquierda el correcto reconocimiento de un botón AWT perteneciente a una interfaz gráfica sencilla, mientras que, a la derecha, el intento de reconocer un botón Java Swing (en el cual se observa superpuesto el cursor) no ofrece el reconocimiento esperado, ya que ignora la existencia del componente Java Swing y muestra el panel contenedor AWT de la interfaz.

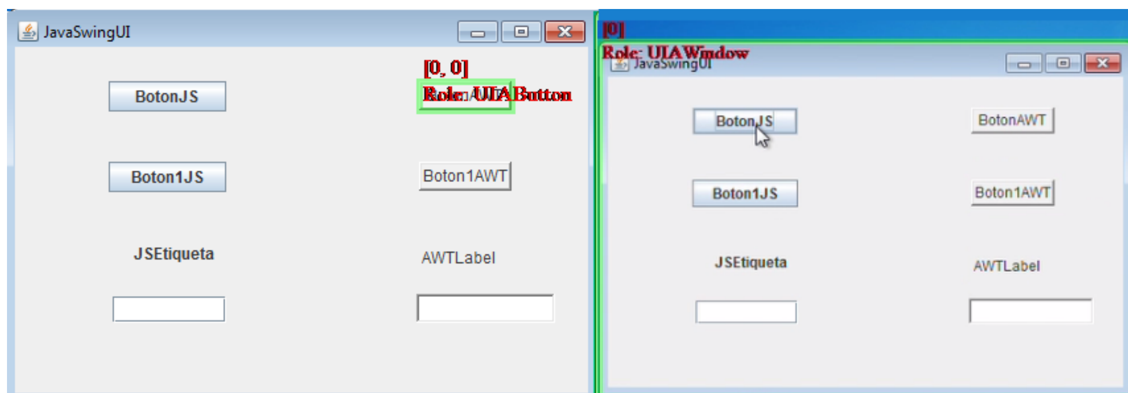

Figura 2-4: Diferencia en el reconocimiento de componentes AWT y Swing en una interfaz simple, donde el componente AWT es reconocido pero el componente Swing no.

2.3 Biblioteca gráfica Java Swing

2.3.1 Arquitectura

Como se ha mencionado anteriormente, el principal problema está causado por la tecnología Java Swing[9], un *framework* modelo-vista-controlador (MVC)[10] para desarrollar interfaces gráficas en Java, con independencia de la plataforma en la cual se está ejecutando. Esta tecnología sigue un modelo de programación por hilos con ciertas características como:

- Implementación en Java que otorga una independencia de plataforma, simulando la apariencia de los componentes nativos pero sin reservar sus recursos. Este tipo de componente independiente de las librerías nativas se conoce como *lightweight component* (en castellano, componente ligero).
- Extensibilidad en los componentes que permite una implementación personalizada de estos, empleando el mecanismo de herencia proporcionado por Java.
- Personalización de los componentes para representar diferentes estilos de apariencia *look and feel*. Todo usuario puede proveer su propia implementación permitiendo cambios en el aspecto de los componentes, pero sin efectuar ningún cambio a nivel de código en el funcionamiento de la aplicación.

9 Java Swing: http://www.oracle.com/technetwork/java/architecture-142923.html

10 *Model View Controller:* http://www.oracle.com/technetwork/articles/javase/index-142890.html

2.3.2 Componentes ligeros

Estos elementos gráficos, presentes en tecnologías como Java Swing, basan su estructura en un contenedor pesado creado en el sistema operativo, con la posterior inclusión de componentes completamente escritos en Java e independientes de la librería gráfica nativa. Esto provoca en TESTAR el reconocimiento de la ventana contenedor, pero impide el correcto reconocimiento de los componentes basados en Java Swing.

Es el propio entorno Java quién decide qué acciones realizar sin esperar que el entorno nativo lo maneje. Se puede decir que esta composición de componentes Java Swing son para el sistema operativo un conjunto de píxeles sin sentido.

Class JComponent

java.lang.Object
    java.awt.Component
        java.awt.Container
            javax.swing.JComponent

Figura 2-5: Estructura de un objeto Swing[11] en Java, donde el componente Swing *JComponent* está incluido en un componente AWT denominado *Container*.

2.3.3 Investigaciones a realizar

Las tecnologías de accesibilidad ofrecidas por las entidades informáticas no solo están destinadas a ayudar a los posibles software de asistencia que emplean las personas con discapacidad, sino que también se diseñan con la intención de ofrecer a los desarrolladores de software de testeo acceso a los diferentes componentes de las aplicaciones, al igual que el *framework* UIAutomation empleado actualmente por TESTAR en su versión Windows.

Hasta ahora, se trabajaba con aplicaciones en las cuales se encontraban interfaces gráficas de usuario formadas por componentes creados a nivel de sistema operativo, los cuales se podían reconocer haciendo uso de las tecnologías de accesibilidad ofrecidas por las propias compañías de estos sistemas operativos. Sin embargo, como hemos mencionado en la sección 2.3.2 Componentes ligeros, ahora se tiene la necesidad de trabajar con unos componentes diferentes basados completamente en Java, cuyo único componente creado a nivel de sistema operativo es el contenedor de la interfaz de usuario.

Por lo tanto, se debe realizar una investigación de las tecnologías de accesibilidad existentes para los entornos Java. Destacando, además, que cada aplicación Java ejecutada se invoca con todos sus recursos en una Máquina Virtual Java (o JVM, por sus siglas en inglés), diferente a la JVM en la cual se encuentra TESTAR. Para comprender mejor esta característica de los entornos Java, es necesario informarse del concepto de JVM y la funcionalidad de una aplicación Java al ejecutarse en los sistemas operativos.

11 Java Jcomponent: https://docs.oracle.com/javase/7/docs/api/javax/swing/JComponent.html

2.4. Máquina Virtual Java

Un factor influyente mencionado a lo largo de este trabajo, y que se debe tener en cuenta para el estudio de las diferentes tecnologías y posibles implementaciones, es la *Java Virtual Machine.* Por lo tanto, se van a comentar ciertos aspectos importantes que se deben tener en cuenta cuando se trabaja en un entorno Java.

Una máquina virtual Java es una máquina virtual que permite ejecutar en los diferentes sistemas operativos aplicaciones basadas en el software Java. Su función es la de ejecutar, interpretar y traducir el código máquina en el *bytecode* Java y viceversa, entendiendo como *bytecode* Java el tipo de instrucciones en el cual se basa la máquina virtual Java para la realización de compilaciones y ejecuciones de sus procesos.

Esta implementación de máquina virtual es la que permite la portabilidad del lenguaje, permitiendo entre diferentes arquitecturas emplear el mismo código Java. Por ejemplo, los programas escritos en Windows pueden ser interpretados en Linux, ya que ambas arquitecturas disponen de una JVM integrada en sus entornos.

En la programación concurrente existen dos unidades básicas de ejecución: procesos e hilos (*threads*). Un sistema informático normalmente tiene muchos procesos e hilos activos, pero concretamente en el lenguaje de programación Java la programación concurrente está referida principalmente a los hilos.

2.4.1 Procesos Java

Un proceso tiene un entorno de ejecución autónomo con un conjunto completo y privado de recursos básicos en tiempo de ejecución, en particular cada proceso tiene su propio espacio de memoria. Aunque los procesos se pueden considerar como sinónimo de programa o aplicación, lo que el usuario ve como una aplicación puede estar formada por un conjunto de varios procesos en cooperación. La mayoría de los sistemas operativos admiten lo llamado *Interprocess Communication*[12] (Comunicación entre procesos), un mecanismo que permite a los procesos compartir espacios de memoria en el sistema y sincronizarse entre ellos.

Concretamente en la tecnología Java, cabe destacar que la mayoría de JVM se ejecutan como un único proceso; es decir, en el caso de tener diferentes aplicaciones Java ejecutándose en nuestro sistema operativo, por lo general se tienen tantas máquinas virtuales Java en ejecución como número de aplicaciones Java. Cabe señalar que, mientras TESTAR está en ejecución en su propia JVM, se debe realizar una comunicación con las aplicaciones Java a testear en otra JVM. Para ello, la herramienta TESTAR debe adaptar las llamadas al sistema operativo mediante el uso del JNI (*Java Native Interface*), para intentar comunicar las diferentes JVM en ejecución.

12 *Interprocess Communication*: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx

Mientras que las llamadas al sistema actuales en TESTAR no son de utilidad, pues no se reconocen los componentes Java Swing, y teniendo dos JVM diferentes con la necesidad de comunicarse entre ellas, una de las posibilidades que ofrece *Oracle* es un conjunto de bibliotecas con diferentes clases disponibles para esta comunicación entre procesos. Cabe la posibilidad de incluir bibliotecas como RMI (Java Remote Invocation)[13] o JMX (Java Management Extensions)[14], pero se debe tener en cuenta que no se puede realizar una modificación en el código de la aplicación Java que se va a testear.

2.4.2 Threads o hilos de ejecución

Los hilos de ejecución a veces se llaman procesos ligeros o subprocesos, y al igual que los procesos, estos también proporcionan un entorno de ejecución pero requiriendo muchos menos recursos que en la creación de un nuevo proceso. Los hilos de ejecución existen dentro de un proceso, y en el caso de existir varios, estos comparten los recursos del proceso incluyendo la memoria y los archivos abiertos. Esto consigue que la comunicación entre ellos sea eficiente, pero a su vez puede ser problemática.

La ejecución multihilos es una característica esencial de la plataforma Java, teniendo cada aplicación Java uno o varios hilos como subprocesos. Desde el punto de vista de la programación de aplicaciones se comienza con un solo hilo, llamado hilo principal, que tiene como única capacidad la de crear hilos adicionales.

Esta situación de trabajar todo en un mismo proceso daría como resultado el tener todos los recursos en la misma JVM, tanto TESTAR como los recursos de la aplicación objetivo a testear. Toda interacción en el reconocimiento de componentes Java Swing se puede implementar sin el uso de las llamadas JNI, sólo empleando llamadas a métodos Java incluidas en un conjunto de clases nuevas dentro la aplicación TESTAR.

13 Java RMI: https://docs.oracle.com/javase/tutorial/rmi/

14 Java JMX: https://docs.oracle.com/javase/tutorial/jmx/

3. Java Accessibility

La tecnología Java Accessibility[15] fue diseñada e implementada con la intención de ofrecer a personas con discapacidad, accesibilidad a la hora de emplear tecnología software Java en su vida cotidiana. Se añaden un conjunto de implementaciones software, para el uso de programas externos que identifiquen y expongan los componentes actuales mostrados en pantalla.

Como ya se ha comentado anteriormente, este tipo de tecnologías de asistencia que ayudan a exponer los diferentes componentes de las interfaces de usuario, también son empleadas por los software de testeo que tienen la necesidad de acceder a la información de dichos componentes. Por lo tanto, se va a hacer uso de esta accesibilidad Java en la herramienta TESTAR, para conseguir solucionar los problemas de reconocimiento con las interfaces Java Swing.

Además, cabe diferenciar la forma de trabajar con todos los recursos en una sola JVM o comunicando varias JVM en ejecución. Por lo que se menciona al final del capítulo un conjunto de aplicaciones y tecnologías que permiten ejecutar todos los recursos de aplicaciones Java en la misma JVM donde se ejecuta TESTAR.

La implementación de esta tecnología está dividida en una estructura de cuatro paquetes diferentes:

- Java Accessibility API[16], para permitir que las aplicaciones sean compatibles con el software de acceso a pantalla.
- Java Accessibility Utilities, un paquete software que permite hacer fácilmente consultas a la JVM.
- El desarrollo de una arquitectura *Look and Feel,* que permite al usuario la posibilidad de interactuar con diferentes interfaces (ej. una interfaz de audio en vez de una visual).
- Java Access Bridge, basado en el mecanismo *Java Native Interface* (JNI) para comunicar la JVM con el entorno nativo. Hace uso de un extenso conjunto de librerías y archivos, para exportar los paquetes de accesibilidad Java a entornos Microsoft Windows.

15 Java Accessibility: https://docs.oracle.com/cd/E17802_01/j2se/javase/technologies/accessibility/docs/jaccess-1.3/doc/core-api.html

16 Java Accessibility API: http://docs.oracle.com/javase/7/docs/technotes/guides/access/jaapi.html

3.1. Java Accessibility API

Esta API ofrece una comunicación entre los componentes de la interfaz de usuario incluidos en una aplicación Java y las tecnologías de asistencia. Además, una aplicación Java que admita e incorpore esta API de accesibilidad puede proporcionar toda la información de su interfaz en un modelo *off screen* (fuera de pantalla), evitando así el uso de ciertos programas adicionales que antes eran necesarios para el reconocimiento *off screen*.

El paquete actual es nombrado como *javax.accessibility*[17], donde se pueden encontrar seis clases y ocho interfaces diferentes, que permiten interactuar con los diferentes componentes. La interfaz principal de esta API es la denominada *Accessible,* que debe ser implementada en todos los componentes diseñados para ofrecer soporte a esta accesibilidad Java. Esta interfaz principal contiene un único método denominado *getAccessibleContext,* que devuelve una instancia de la clase *AccessibleContext*, la cual contiene la cantidad mínima de información de accesibilidad de un objeto Java (rol, nombre, descripción y estado), además de un conjunto de métodos para acceder a información específica de estos objetos (acciones posibles, representación gráfica del objeto, etc).

3.2. Java Accessibility Utilities

Este paquete software de utilidades de accesibilidad Java está diseñado para los proveedores de tecnología de asistencia implicados en desarrollar productos preparados para interaccionar con aplicaciones Java. Proporciona el soporte necesario, para que las tecnologías de asistencia puedan localizar y consultar objetos de interfaz de usuario dentro de las JVM (J*ava Virtual Machine*), y la detección de eventos (*event listeners*) de dichos objetos.

También proporciona una lista de las ventanas de nivel superior (*top-level windows*, una ventana que no es secundaria y tiene la ventana escritorio como padre) de todas las aplicaciones Java, y una arquitectura para escuchar eventos que nos informa de cuando una de estas ventanas aparece o desaparece, se realiza un enfoque sobre ellas, se interactúa con la posición del ratón o se añade un evento en la lista de eventos del sistema.

En este paquete se incluyen tres clases para el seguimiento de eventos en la JVM:

- La primera *AWTEventMonitor* para todos los eventos realizados por componentes AWT, pudiéndose registrar para todo componente y en toda aplicación Java dentro de la JVM, un oyente (*listener*) individual.
- La segunda *SwingEventMonitor,* una extensión de la primera, que proporciona un apoyo adicional para los componentes Swing y sus eventos.
- La tercera *AccessibilityEventMonitor* proporciona soporte para los objetos que implementen la interfaz de accesibilidad y realizan eventos de cambio de alguna propiedad, notificando estos eventos a la tecnología de asistencia.

17 Javax.accessibility: http://docs.oracle.com/javase/7/docs/api/javax/accessibility/package-summary.html

3.3. Arquitectura *Look and Feel*

Las interfaces de usuario en las clases Java Swing están diseñadas según la arquitectura *Pluggable Look and Feel*[18], la cual nos permite separar el código que representa la parte visual de la interfaz de usuario, del código que representa la estructura y los datos del componente.

El concepto *pluggable* o conectable significa que, tanto la representación visual como el comportamiento de un elemento GUI, pueden cambiar sin modificar el programa aunque el componente esté en pantalla. Cuando las aplicaciones Java Swing invocan un nuevo componente, este sabe cómo debe representarse en pantalla y cómo reaccionar a los eventos, delegando estas tareas a sus clases especializadas.

Esto se consigue dividiendo el componente Java Swing en al menos cinco objetos implementados en lenguaje Java. Estos son: el propio componente (ej. botón), la clase interfaz en Java que define la interfaz de usuario (ej. la interfaz de usuario del botón), una implementación por defecto de esa interfaz de usuario (ej. botón básico), una clase interfaz Java que define el modelo del componente (ej. el modelo del botón) y, finalmente, una implementación por defecto de ese modelo (ej. el modelo del botón Swing). Sin embargo, en la mayoría de componentes Java Swing, el programador no necesita conocer ni preocuparse de la subdivisión de objetos que se crean, ya que éste simplemente debe crear una nueva instancia del primer objeto de los cinco y escribir el código deseado para interactuar con él.

3.4. Java Access Bridge

La tecnología Java Access Bridge[19] se encarga de exponer la API del paquete de accesibilidad Java en los sistemas Microsoft Windows, mediante el uso de bibliotecas de enlace dinámico (*dynamic-link library, dll*)[20]. Esto permite que las aplicaciones Java que implementan la Java Accessibility API y se encuentran en ejecución en su propia JVM, sean accesibles para las herramientas que emplean tecnologías de asistencia en los sistemas Microsoft Windows (como el caso de TESTAR). Todo esto es parte de la tecnología mencionada anteriormente Java Accessibility, un conjunto de clases que ayudan a las tecnologías de asistencia a acceder a los componentes Java de las interfaces gráficas de usuario.

En el caso concreto de la herramienta TESTAR, la única tecnología Java que presenta problemas con el reconocimiento de componentes de las interfaces, empleando la API de accesibilidad a nivel de Sistema Operativo, es la biblioteca gráfica Swing; sin embargo, el uso de este *bridge* se puede implementar en TESTAR como una comunicación general hacía todo tipo de aplicación Java. Dichos componentes gráficos, que se encuentran en ejecución en una JVM distinta, serían accesibles para TESTAR, consiguiendo su reconocimiento y su uso posterior para el testeo sobre la aplicación Swing.

18 *Pluggable Look and Feel:* https://community.oracle.com/docs/DOC-983327

19 Java Access Bridge: http://docs.oracle.com/javase/accessbridge/2.0.2/introduction.htm

http://www.oracle.com/technetwork/articles/javase/index-jsp-136191.html

20 *dll*: https://support.microsoft.com/es-es/help/815065/what-is-a-dll

3.4.1 Arquitectura

La comunicación que Java Access Bridge permite entre tecnologías de asistencia y aplicaciones Java se denomina comunicación entre procesos (*inter-process communication*), permitiéndose tanto en sistemas Windows de 64-bits, como en los sistemas de 32-bits.

La arquitectura concreta y forma de actuación del Access Bridge es la presente en la Figura 3-1.

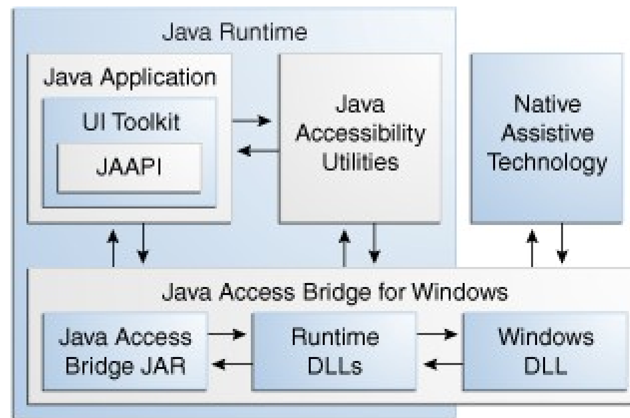

Figura 3-1: Diagrama de la Arquitectura Java Access Bridge.

Se pueden observar dos estructuras principales que se superponen entre sí, Java Runtime[21] y Java Access Bridge for Windows, además del componente Native Assistive Technology que se comunica únicamente con el Access Bridge.

El Java Access Bridge proporciona la API de accesibilidad de Java como una *dll* de Windows (*WindowsAccessBridge.dll*), para que las tecnologías de asistencia la carguen y se vinculen con ella, además de otras dos *dll* (*JavaAccessBridge.dll* y *JAWTAccessBridge.dll*) que son cargadas por el Java Runtime. Este par de *dll* se comunicarán con la aplicación a través de la Java Accessibility API, y a través de ella lo harán los componentes de la interfaz de usuario.

También se incluye dentro de Java Access Bridge un conjunto de clases Java dentro de un archivo JAR (access-bridge.jar), el cual gestiona la comunicación entre las *dll* mencionadas anteriormente y el resto de código Java que se encuentra en el Java Runtime. Este archivo JAR se carga en el *Java SE runtime* a través del archivo *accessibility.properties,* y a su vez carga las *dll* de Java a través del JNI (*Java Native Interface*).

3.4.2 Instalación

La tecnología Java Access Bridge tiene diferentes versiones preparadas según la versión Java SE empleada en nuestro sistema, cada una con una forma de instalación y activación diferente. Las últimas versiones a destacar presentadas por *Oracle* son:

- Versión 2.0.2, a partir de Java SE 5
- Versión 2.0.3 y superiores, a partir de Java SE 7u6

21 Java Runtime Enviroment: Paquete software que contiene lo necesario para que las aplicaciones Java puedan correr en el sistema operativo.

La versión 2.0.2 se basa en la instalación de los archivos ofrecidos por *Oracle* en unas rutas específicas del sistema operativo Windows y Java Runtime Enviroment (JRE), tal y como se indica en su página de instalación. Cabe añadir que la cantidad de archivos y rutas varían dependiendo de si el sistema de trabajo es de 32 o 64 bits.

A partir de la versión 2.0.3, es decir para versiones Java superiores a la 7u6, se indica que todos los archivos necesarios vienen ya instalados y sólo es necesario la activación del Access Bridge desde el centro de accesibilidad del sistema. Se puede acceder al centro de accesibilidad a través del panel de control, en concreto en la ruta "*Panel de control\Todos los elementos de Panel de control\Centro de accesibilidad\Facilitar el uso del equipo*".

3.4.3 Java Monkey

Java Monkey es una de las dos aplicaciones ofrecidas por *Oracle,* dentro de la versión 2.0.2 del Access Bridge, que emplea la API de accesibilidad para obtener y mostrar de las diferentes JVM en ejecución, un árbol jerárquico de componentes Java, permitiendo al usuario recorrer los diferentes componentes para mostrar toda su información asociada.

Dentro de todos los archivos ofrecidos, se encuentran pertenecientes a Java Monkey tres aplicaciones compiladas y preparadas para su ejecución, cada una con un nombre diferente el cual indica la versión del sistema donde deben usarse (JavaMonkey.exe, JavaMonkey-32.exe y JavaMonkey-64.exe). Además, contiene el código fuente de la clase principal (JavaMonkey.cpp), donde se hace referencia a un archivo de cabecera (JavaMonkey.h), y otros archivos propios del Access Bridge. Sin embargo, en el conjunto de archivos ofrecido por *Oracle,* no se incluyen todos los archivos mencionados dentro de la clase principal.

Una posible opción de emplear esta aplicación como intermediario entre TESTAR y los componentes Java Swing no puede llevarse a cabo debido a la falta de archivos para su compilación. Por lo tanto, se mantiene como caso de estudio para comprender mejor la API de accesibilidad dentro del Access Bridge, pero queda descartada como posible implementación.

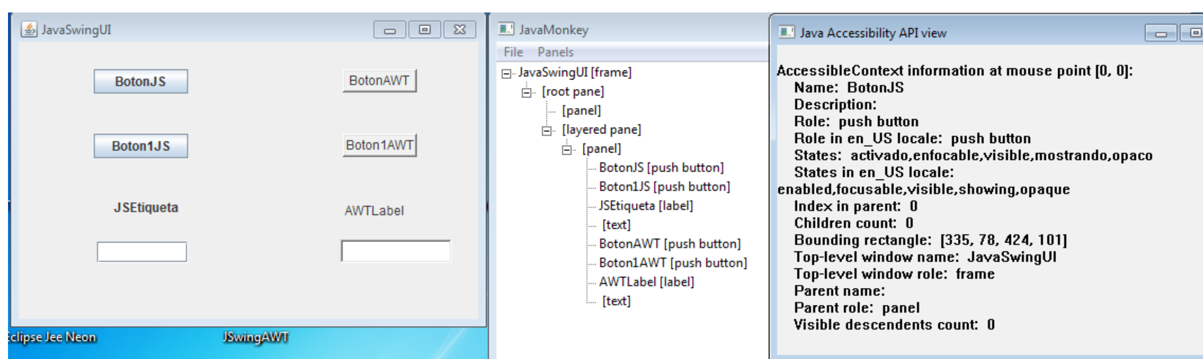

Figura 3-2: Funcionamiento de la aplicación Java Monkey sobre una interfaz sencilla, donde se muestra la información de los componentes Java.

3.4.4 Java Ferret

Java Ferret es la otra aplicación ofrecida por *Oracle*, que emplea la API de accesibilidad para examinar la información sobre los diferentes componentes presentes dentro de las interfaces Java. A diferencia de Java Monkey, con esta aplicación se tienen diversos métodos para obtener la información de accesibilidad de los componentes Java mediante el uso de eventos.

Los archivos ofrecidos siguen el mismo criterio que los de Java Monkey, tres aplicaciones compiladas y preparadas para su ejecución, cada una con un nombre diferente el cual indica la versión del sistema donde deben usarse (JavaFerret.exe, JavaFerret-32.exe y JavaFerret-64.exe); junto al código fuente de la clase principal (JavaFerret.cpp). Existen de la misma manera ciertas referencias a archivos no incluidos, como la cabecera (JavaFerret.h) y otros propios del Access Bridge.

Sin embargo, al contrario que Java Monkey, si se realiza una búsqueda en la red, se puede encontrar un proyecto en *GitHub*[22] que incluye una versión más actualizada de Java Ferret, incluyendo los archivos que faltan en la versión de *Oracle*. Se realiza una descarga de estos nuevos archivos como caso de estudio para comprender mejor la API de accesibilidad dentro del Access Bridge; pero finalmente no se han integrado en la herramienta TESTAR.

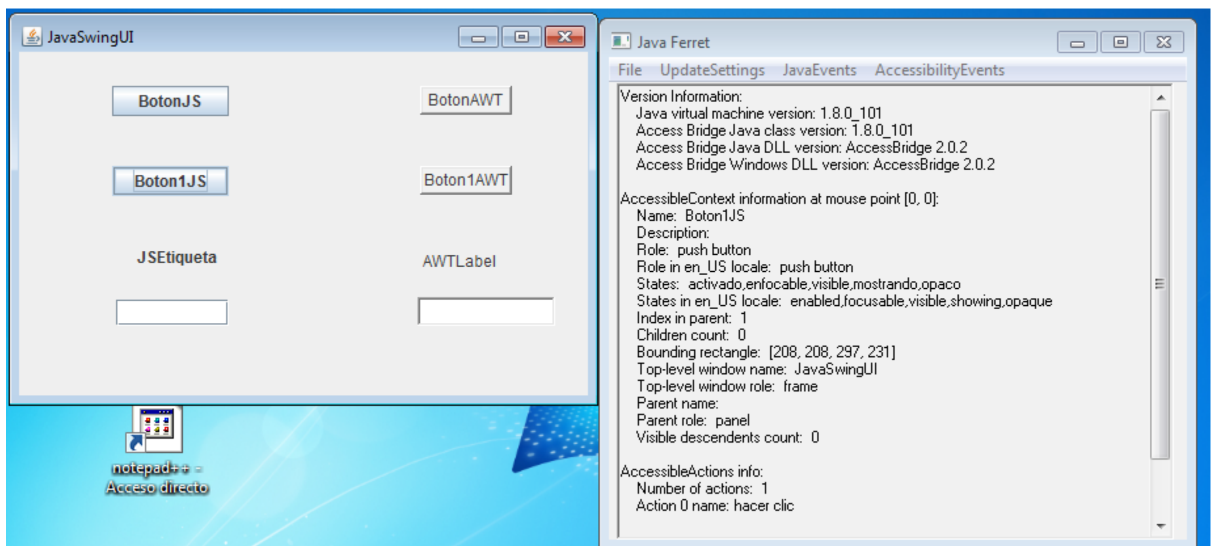

Figura 3-3: Funcionamiento de la aplicación Java Ferret sobre una interfaz sencilla, donde se muestra la información de los componentes Java.

22 Proyecto Java Ferret: https://github.com/nmquan2504/JavaFerret

3.5. Tecnologías para ejecutar los recursos Java de una aplicación en la misma JVM que TESTAR

Las tecnologías y aplicaciones comentadas a continuación se han estudiado con el fin de intentar ejecutar los recursos Java de las aplicaciones a testear en la misma JVM que TESTAR. El posterior reconocimiento de todos los componentes Java Swing se basa en el uso del paquete de accesibilidad Java llamado *javax.accessibility.*

3.5.1 Swing Explorer

En una de las búsquedas iniciales donde se recopila información acerca de los componentes Java Swing y su posible reconocimiento a implementar en TESTAR, se encontró un programa de código abierto llamado Swing Explorer, destinado a la exploración de los componentes visuales Java Swing. No obstante, la página oficial del proyecto Swing Explorer no está activa [23], pero se puede encontrar la herramienta y su código en el repositorio *GitHub*[24] de la compañía *robotframework.*

Aun siendo una herramienta desactualizada y en desuso, se decide descargar la herramienta y su código para realizar pruebas sobre la aplicación de demostración SwingSet2, una aplicación Java Swing con una gran variedad de elementos, que incluye la compañía *Sun Microsystems*[25] en sus entornos Java; y otra sencilla interfaz propia con cuatro componentes Swing y otros cuatro AWT.

La inicialización de la herramienta se basa en una invocación de comandos desde el símbolo del sistema, en la ruta de Swing Explorer, donde, siguiendo la indicación de la sección *Wiki* del *GitHub*, los archivos.jar del programa Swing Explorer se invocan junto al archivo.jar de la aplicación a testear, llamando concretamente a la clase interna *Launcher*. Además, como parámetro adicional necesario, se indica el nombre de la clase "main" de la aplicación almacenada en el archivo.jar.

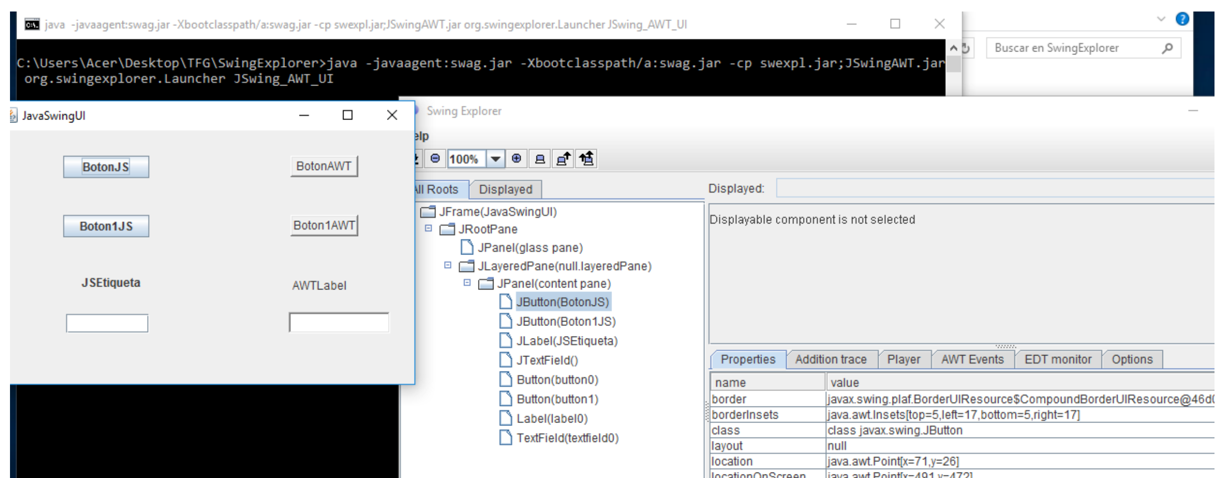

Figura 3-4: Funcionamiento de la aplicación Swing Explorer sobre una sencilla aplicación que contiene componentes Java Swing y AWT.

23 Foro *GitHub* sobre página principal Swing Explorer: https://github.com/robotframework/SwingLibrary/issues/72

24 Proyecto Swing Explorer: https://github.com/robotframework/swingexplorer

25 *Sun Microsystems:* *https://www.oracle.com/sun/index.html* *https://es.wikipedia.org/wiki/Sun_Microsystems*

3.5.2 Java Reflection & ClassLoader

En el estudio de la herramienta mencionada anteriormente, se realiza una invocación del método principal de las aplicaciones (como archivos.jar), gracias a la tecnología denominada Java Reflection[26]. Concretamente en Swing Explorer, la invocación se realiza añadiendo como parámetro el nombre de la clase "main"; sin embargo, en una posible implementación en TESTAR, se debe intentar eliminar este parámetro de entrada y obtener de otra forma dicha invocación de este método principal.

La denominada Reflection es una tecnología de Java que permite inspeccionar y llamar dinámicamente clases, métodos, atributos, etc., que se encuentran en ejecución en la *Java Virtual Machine* (JVM). Se puede "reflejar" la estructura del programa contenido en el archivo.jar que se debe testear, para intentar obtener desde la interfaz invocada los componentes Swing necesarios.

Para complementar esta invocación, se hace uso de la clase Java ClassLoader[27], un objeto Java que permite ejecutar dinámicamente clases Java en la JVM. La forma de introducir o indicar la ubicación del archivo.jar, será concretando la ruta o directorio del sistema, para ello se hace uso de la clase URLClassLoader[28].

Se necesita realizar la invocación del método principal de la aplicación como se ha mencionado, y para ello, se debe conocer el nombre concreto del método "main". En un principio, TESTAR hace uso únicamente de la ruta de los archivos como parámetro de entrada, por lo que barajando opciones para evitar añadir un segundo campo de entrada, surge la idea de hacer uso del archivo "Manifest". Este es un archivo específico contenido en los archivos.jar, el cual se usa para definir datos relativos a la extensión y sus clases. En este, se debe especificar entre otras características, cuál es el método principal de la aplicación, campo que se puede emplear para la obtención del método "main" y su posterior invocación.

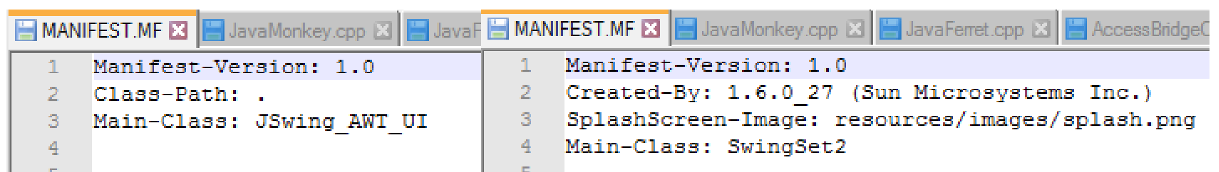

Figura 3-5: Archivos Manifest donde se observa la especificación de la Main-Class.

26 Java Reflection: https://docs.oracle.com/javase/tutorial/reflect/TOC.html

27 ClassLoader: https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html

28 URLClassLoader: https://docs.oracle.com/javase/7/docs/api/java/net/URLClassLoader.html

3.5.3 AWT Event Listener & Toolkit

La clase AWTEventListener[29] es una interfaz de escucha para recibir notificaciones de los eventos creados por AWT, que observa pasivamente todos los eventos en la JVM en la que se inicie. En su propia especificación dentro de *Oracle*, se hace mención al uso de esta clase para las aplicaciones dedicadas al testeo o en implementaciones relacionadas con el paquete Java Accessibility.

Junto al uso de la clase Toolkit[30], una superclase abstracta de todas las implementaciones actuales de AWT que incluye métodos que pueden consultar el sistema de ejecución de Java directamente, se pueden obtener los eventos creados por las interfaces de usuario y saber cuál es el componente causante de dicho evento. La implementación para recoger los eventos se realiza obteniendo el *toolkit* por defecto que está en ejecución dentro de nuestra JVM, gracias al método *getDefaultToolkit()*, y añadiendo concretamente el método *addAWTEventListener*, que será el encargado de escuchar dentro de la máquina Java los distintos eventos que realizan las aplicaciones.

```
Toolkit.getDefaultToolkit().addAWTEventListener(new AWTEventListener() {
    public void eventDispatched(AWTEvent event) {
        System.out.println("--> " + event);
    }
}, FocusEvent.FOCUS_EVENT_MASK | WindowEvent.WINDOW_FOCUS_EVENT_MASK);
```

Figura 3-6: Ejemplo de la creación de una interfaz de escucha de eventos[31].

29 AwtEventListener: https://docs.oracle.com/javase/7/docs/api/java/awt/event/AWTEventListener.html

30 Toolkit: https://docs.oracle.com/javase/7/docs/api/java/awt/Toolkit.html

31 Ejemplo AWTEventListener: http://www.programcreek.com/java-api-examples/java.awt.event.AWTEventListener

4. Uso de las tecnologías investigadas

Al existir diferencias en trabajar con todos los componentes Java Swing y sus recursos en la misma JVM que TESTAR, o en dos JVM diferentes, se describen en este capítulo dos posibles implementaciones a realizar. Ambas se basan en el uso de la tecnología de accesibilidad Java descrita anteriormente, diferenciando su uso según la JVM en la que se encuentre en ejecución la aplicación Java Swing que se debe testear.

4.1. Reconocer componentes de una JVM diferente a TESTAR

En la situación de tener en ejecución la herramienta TESTAR en una JVM diferente a la aplicación Java Swing a testear, se debe hacer uso de la tecnología Java Access Bridge, perteneciente a la accesibilidad Java descrita anteriormente.

Existen en ejecución en el sistema dos JVM diferentes:

- La JVM de la aplicación Java Swing, con todos los recursos Java pertenecientes a los componentes de sus interfaces de usuario.
- La JVM de la herramienta TESTAR, desde la cual es necesario realizar una lectura de recursos hacía la otra JVM para obtener la información de sus componentes Java.

Para realizar esta lectura de recursos Java se emplea la tecnología Access Bridge, la cual se debe activar en el panel de accesibilidad del sistema operativo (ya que trabajamos en una versión superior a la Java SE 7u6). Para el uso de este *bridge* en la herramienta TESTAR, es necesario introducir en los diferentes archivos de compilación el conjunto de archivos en lenguaje C y librerías dinámicas ofrecidas por *Oracle.*

Una vez estos archivos sean compilados correctamente en TESTAR, para su posterior uso en el entorno de ejecución, podemos introducir en las llamadas JNI empleadas desde el archivo “main.cpp”, las funciones descritas en el API del Access Bridge.

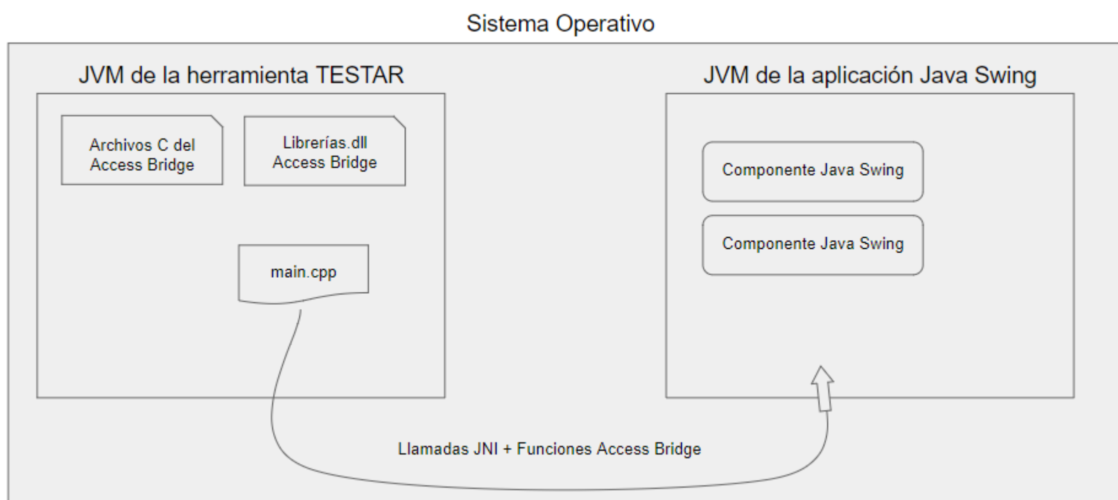

Figura 4-1: Diagrama que representa la lectura de componentes Java, empleando las funciones de accesibilidad del Access Bridge en el archivo “main.cpp” encargado de realizar las llamadas JNI a través del sistema operativo.

4.2. Reconocer componentes existentes en la misma JVM que TESTAR

En el caso de emplear las tecnologías mencionadas en la sección 3.5. Tecnologías para ejecutar los recursos Java de una aplicación en la misma JVM que TESTAR, dispondremos de todos los recursos ejecutándose en el mismo proceso Java. En este caso, el acceso a toda la información de los componentes Java Swing se basa en el uso del paquete *javax.accessibility* sin la necesidad de emplear el Access Bridge. Además, cabe mencionar que, al trabajar directamente con el componente Java y su información asociada de accesibilidad, el conjunto de propiedades obtenido es mayor a los ofrecidos por el Access Bridge.

Entre los distintos recursos Java en ejecución en la JVM, se debe obtener el contenedor JFrame de la interfaz Java Swing, es decir, obtener el componente Java raíz de las interfaces de usuario basadas en esta tecnología. Una vez se consiga obtener este componente se puede hacer uso del paquete *javax.accessibility*, la accesibilidad Java que permite extraer la información de los componentes para la creación de widgets en TESTAR. Con la obtención del componente raíz, se puede acceder a toda la información de los componentes pertenecientes a la interfaz, mediante las llamadas a sus componentes "hijos".

La obtención de este JFrame se realiza mediante los siguientes pasos:

1. Iniciar en la JVM de la herramienta TESTAR, una interfaz de escucha de eventos (*AWTEventListener*) que detecte cuando una ventana se active (*Windows.Activated)* y compruebe que el componente detonante de este evento es un objeto JFrame.
2. Disponer del archivo.jar de la aplicación objetivo a testear, en el cual se puede acceder a la clase "main" que inicia la aplicación.
3. Realizar en un nuevo hilo de ejecución en la JVM de TESTAR, con la invocación de esta clase "main", empleando las tecnologías Java Reflection y ClassLoader.

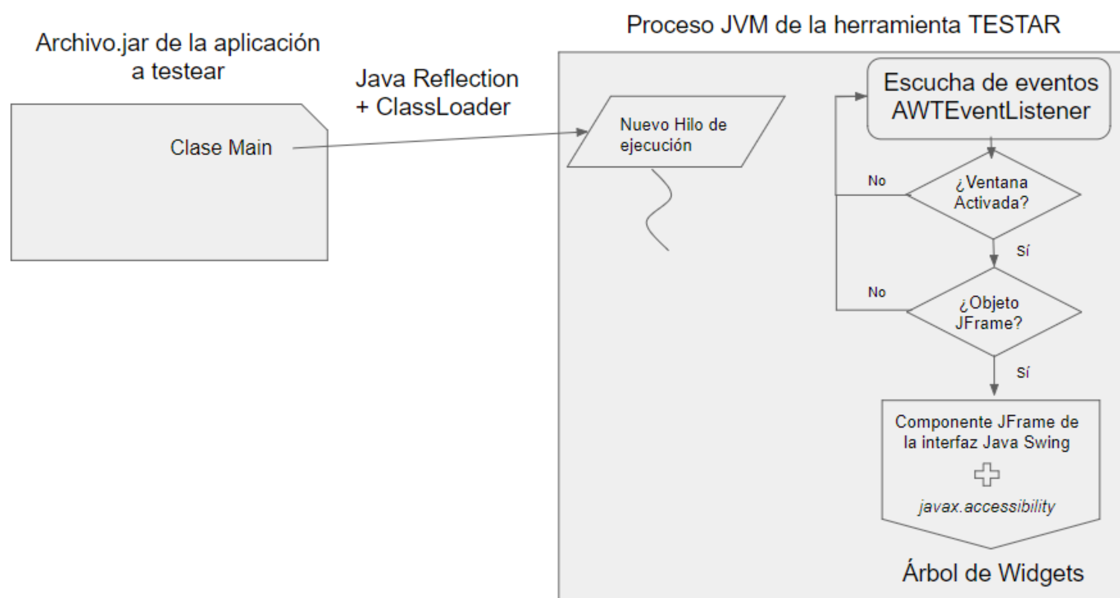

Figura 4-2: Diagrama que representa la ejecución de un nuevo hilo en la JVM de TESTAR empleando las tecnologías Reflection y ClassLoader, con la posterior obtención del componente JFrame gracias a la interfaz de escucha de eventos AWTEventListener. Para así conseguir crear el árbol de widgets que representa los componentes Java mediante el uso del paquete de accesibilidad *javax.accessibility.*

5. Implementaciones

5.1. Arquitectura general de TESTAR

Antes de comenzar a describir las implementaciones realizadas a lo largo de esta investigación, hay que explicar la arquitectura con la cual se va a trabajar desde un principio. Cabe destacar, que desde el comienzo, la herramienta TESTAR ha sido actualizada por diversos equipos de trabajo para complementar, entre otras mejoras, la funcionalidad de testeo respecto a ciertos componentes, o añadiendo una implementación adecuada para su funcionamiento en Linux. Por lo tanto, la descripción realizada a continuación, puede haber sido modificada debido a trabajar con una aplicación en continua actualización.

En general, se puede decir que TESTAR está dividida en cuatro proyectos Java principales, con la posterior utilización de un quinto, dependiendo del sistema operativo en el que se encuentre realizando los testeos. Todos estos proyectos son enlazados junto a unas librerías de enlace dinámico mediante la herramienta *Apache Ant*[32], encargada de la fase de compilación y construcción final de nuestra aplicación. Los diferentes proyectos y su implicación en TESTAR son los siguientes:

- core: El pilar de TESTAR en cuanto a la creación del SUT (*System Under Test*)[33], mediante la declaración de múltiples clases e interfaces, las cuales distribuyen la estructura del objeto *State* principal. Este objeto es lo que se denomina un *Widget* (*State* es el *Widget* raíz), un elemento de control dentro del SUT que incorpora la característica de ser *Taggable*. Un *Tag* es una etiqueta similar a las *keys* de los Java *Map*, propiedades que se implementan en los *Widget* con un nombre y valor concreto, para el futuro reconocimiento de TESTAR y su uso en las pruebas de testeo.

- graph: Esta es la parte encargada de monitorizar el SUT y los *Widget* objetivos en los cuales se han realizado diferentes acciones. Según estas acciones y sus respuestas generadas a lo largo de las múltiples iteraciones, se inicia y almacena la creación de los resultados o informes, en forma de gráficos.

- native: Consta actualmente de una única clase denominada *NativeLinker*, en ella residen diferentes métodos para obtener el SUT con el cual va a trabajar TESTAR. Según los parámetros empleados para iniciar el SUT, se invoca el método correspondiente, pudiendo introducir la ruta de la aplicación o el nombre de un proceso en ejecución entre otras posibilidades.

32 Apache Ant: https://ant.apache.org/manual/

33 SUT: Sistema en el cual se están realizando pruebas para comprobar un funcionamiento correcto.

- testar: El proyecto principal de la herramienta, donde según el tipo de funcionalidad seleccionado se realiza una invocación diferente. Existen, entre otros modos de funcionalidad, la visualización de los componentes de la interfaz y su información, permitiendo deshabilitarlos manualmente (para dirigir el testeo); o las pruebas de testeo con un número de repeticiones y acciones a elegir. Estas funcionalidades generan un archivo.log donde se muestra, entre otros campos, la fecha, el modo de funcionalidad empleado y las acciones que se han realizado en los *Widget.*

- windows: Este proyecto permite el funcionamiento de TESTAR en los sistemas operativos Windows, conteniendo las invocaciones JNI que se realizan desde nuestra herramienta. También se incluyen clases que implementan las interfaces del paquete *core*, preparadas para la creación del *State* raíz y sus respectivos *Widget*.

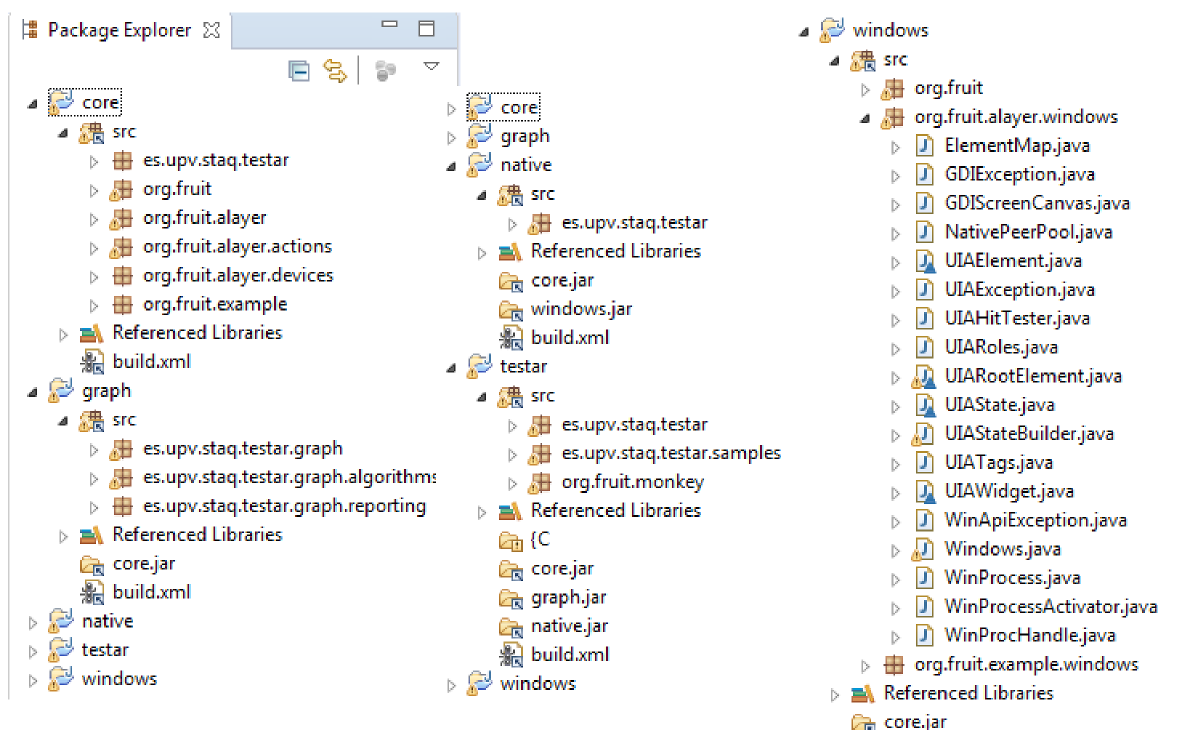

Figura 5-1: Estructura de los diferentes proyectos Java que forman la herramienta TESTAR.

5.1.1 Creación de widgets en base a la interfaz de usuario

Todos los componentes reconocidos de las interfaces de usuario, se trabajan en la herramienta TESTAR como objetos *UIAWidget,* que incluyen entre sus propiedades la característica de ser *Taggable.* Estos *Tags* son propiedades propias de los widgets, y representan características o estados de los componentes en ejecución de las interfaces gráficas. Cuando se comienza a trabajar con el componente raíz de la interfaz de usuario, se crea un widget denominado *UIAState* diseñado para ser empleado como widget raíz.

A partir de este widget raíz *UIAState,* se van creando sucesivamente tantos *UIAWidget* como componentes sean reconocidos de las interfaces, respetando la jerarquía de componentes padres e hijos. Además, a medida que se vayan reconociendo los componentes para la creación de widgets, se hará uso de la accesibilidad Java para añadir las características propias de estos componentes en sus respectivos widgets creados en TESTAR.

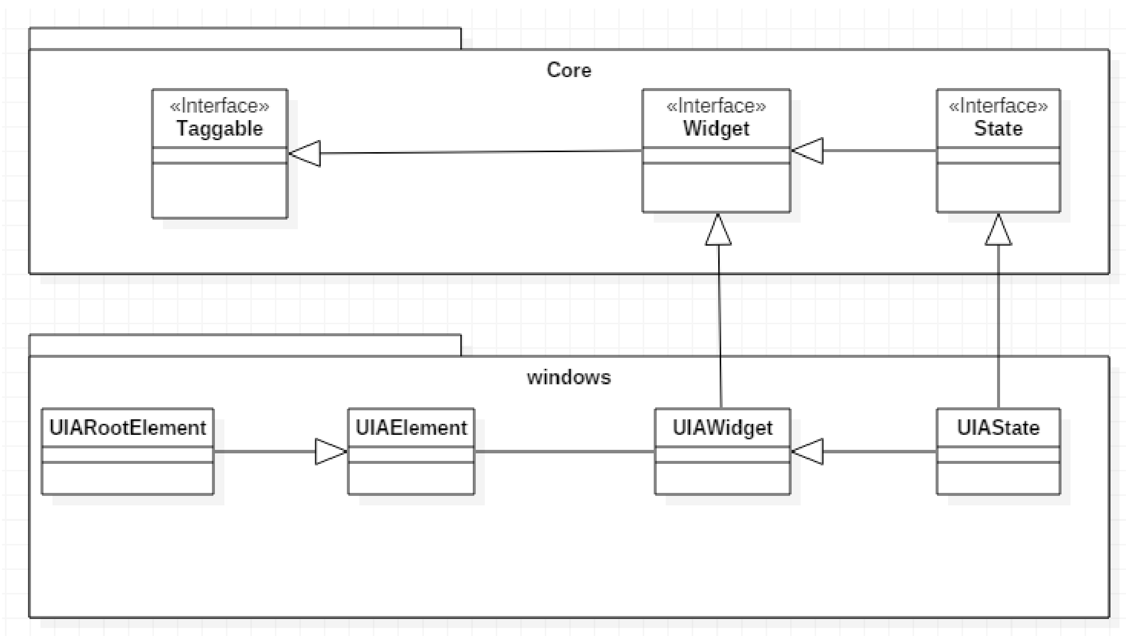

Figura 5-2: Diagrama UML simple de los *Widget*, donde se observa la asociación existente entre los diferentes proyectos Java y clases.

Como se observa en la Figura 5-2, también existen en TESTAR unos objetos denominados *UIAElement,* con el objeto *UIARootElement* representando un elemento raíz. Estos objetos no tienen una interfaz definida previamente en la parte *core* de TESTAR; son directamente creados en el proyecto *windows,* al igual que los *UIAWidget* pero sin la propiedad de ser *Taggable.* Estos *UIAElement* también se emplean para representar características de los componentes detectados en la interfaz gráfica de usuario.

Por lo tanto, se tiene una creación de dos objetos en función de los componentes de las interfaces. Estos objetos relacionados entre sí forman los denominados widgets, y son los empleados posteriormente por TESTAR, cuando se inicie la funcionalidad de obtención de información sobre la interfaz, o la funcionalidad de testeo y realización de acciones sobre la aplicación mediante la interfaz de usuario.

5.2. Implementación Swing Reflection

Esta es la implementación diseñada para ejecutar todos los recursos de la aplicación objetivo a testear, en la JVM donde se encuentra en ejecución la herramienta TESTAR (4.2. Reconocer componentes existentes en la misma JVM que TESTAR). Lo primero es indicar el código añadido en TESTAR para la extracción del componente JFrame, donde posteriormente se hace uso del paquete de accesibilidad Java *javax.accessibility,* para la obtención de información de los componentes que componen la interfaz de usuario.

5.2.1 Extracción y uso del JFrame

La primera fase de esta implementación consistirá en obtener el componente principal JFrame de la aplicación a testear, teniendo como único parámetro de entrada el *path* del archivo.jar que la contiene. Cuando se establece un *path* como entrada en TESTAR, la función invocada para crear el SUT (*System Under Test*) es la denominada "*public static SUT getNativeSUT(String executableCommand)*". Esta función se encarga de llamar al proyecto *windows* (dependiendo del sistema operativo), donde, mediante llamadas al JNI, se crea un nuevo proceso basado en la aplicación incluida del archivo seleccionado.

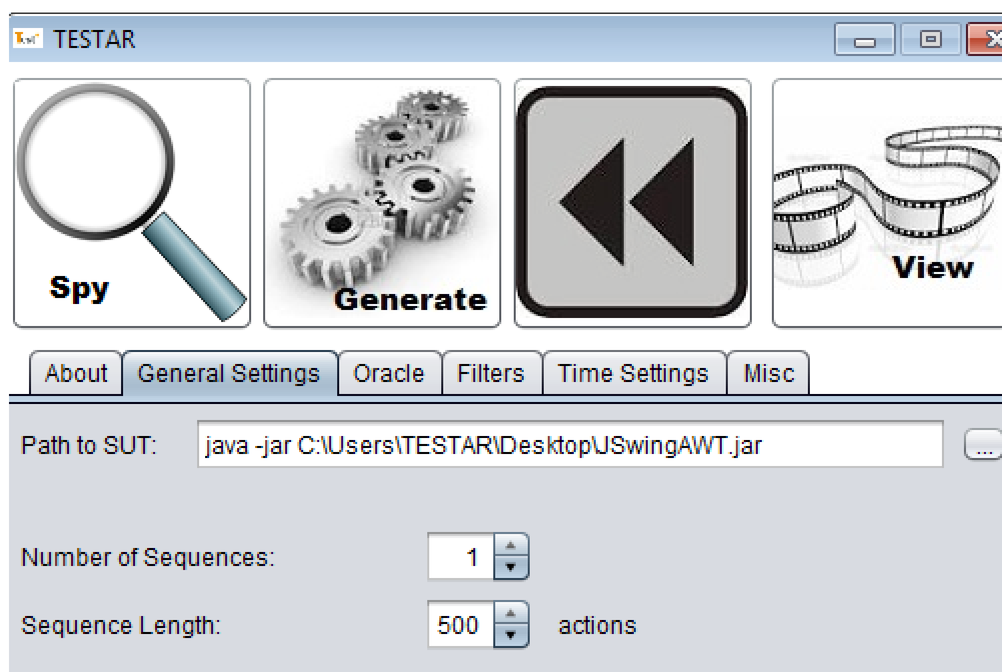

Figura 5-3: Se indica el *path* de un archivo para inicializar la aplicación y realizar el testeo.

Donde este nuevo proceso se creaba como SUT para ser empleado por TESTAR, se aplica la tecnología Reflection ejecutando un nuevo hilo de ejecución en la JVM, en vez de crear un nuevo proceso Java con su propia JVM. Por lo tanto, el SUT empleado para realizar el testeo estará contenido en el mismo proceso que TESTAR como unos de sus hilos de ejecución.

Para preparar la interfaz encargada de obtener el JFrame, se debe crear en el Toolkit del entorno Java actual de ejecución (JVM de TESTAR), una interfaz encargada de escuchar eventos (*AWTEventListener)*. Se condiciona un evento para el caso concreto de detectar la activación de una ventana en la JVM (*WINDOW_ACTIVATED*), y posteriormente se comprueba si el componente detonante del mismo es un objeto JFrame *(Instanceof JFrame)*. En caso afirmativo, se asocia con nuestra herramienta TESTAR, gracias al añadido de un nuevo *Tag* en el SUT que representa la aplicación a testear.

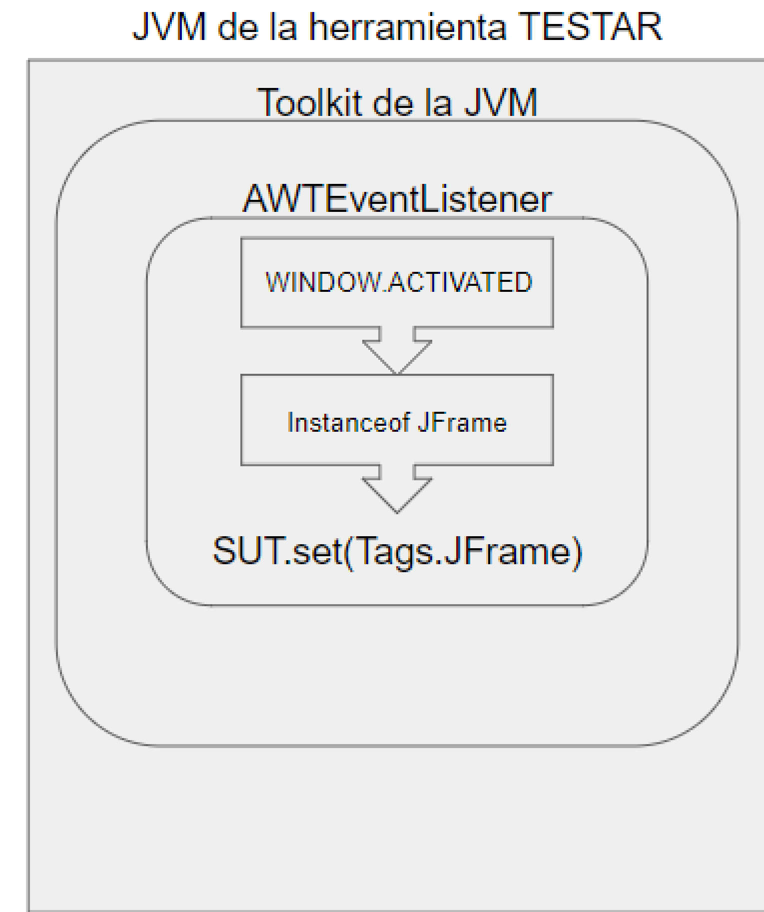

Figura 5-4: Creación de una interfaz de escucha de eventos en TESTAR, para obtener un componente JFrame que active una ventana.

Una vez la interfaz para escuchar eventos está preparada, debemos ejecutar la aplicación Java Swing con sus recursos en la JVM de TESTAR. Primero se crea un objeto JarFile en base al archivo.jar que contiene la aplicación Java, por lo que se debe conocer el *path* de dicho archivo; además, gracias al *Manifest* de las aplicaciones Java, se puede obtener la clase principal de la aplicación.

Acto seguido se crea un objeto ClassLoader gracias a la clase URLClassLoader, encargado de crear una clase genérica que representa la clase principal de la aplicación a testear. En este objeto de clase genérica, se accede al método "main" que contiene la aplicación Java.

Finalmente, realizamos una invocación del método principal como un nuevo hilo de ejecución, lo que provoca ejecutar la aplicación Java Swing y todos sus recursos dentro de la JVM de TESTAR.

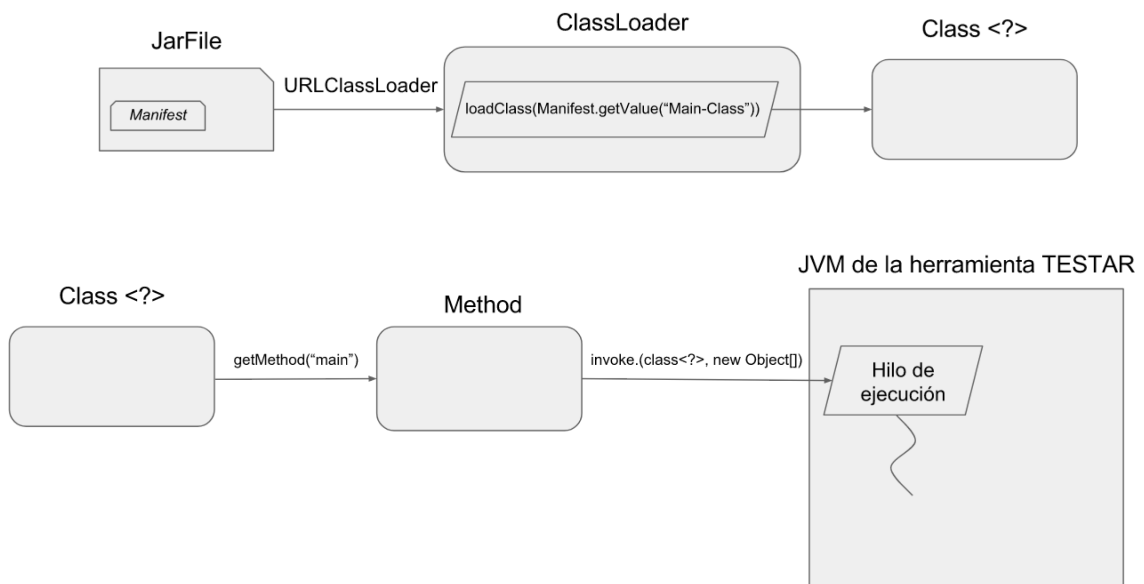

Figura 5-5: Secuencia de creación de objetos en base al archivo.jar que contiene la aplicación a testear, para ejecutar todos sus recursos Java como nuevo hilo de ejecución en la JVM de la herramienta TESTAR.

5.2.2 Uso del paquete de accesibilidad Java *javax.accessibility*

La forma de enlazar con la ventana de la interfaz a testear es la implementada para trabajar con la accesibilidad de Windows UI Automation, debido a que se trata de una ventana contenedor basada en la tecnología AWT, creada a nivel de sistema operativo. Para esta obtención se trabaja con el *PID* que tiene asociado el SUT, buscando en la ventana existente en el proceso con dicho *PID,* para después asociarla como *UIARootElement*, es decir, el elemento raíz.

Una vez se ha obtenido el elemento raíz y creado el *UIAState* asociándose mutuamente, comienza el recorrido recursivo desde el componente JFrame a través de todos sus denominados hijos. La previa asociación que se realizaba a los elementos, extrayendo las características de las llamadas al sistema operativo mediante el JNI, se sustituye por asociaciones hacía las diferentes propiedades extraídas de cada componente Java desde su *AccessibleContext*[34].

Esta propiedad denominada *AccessibleContext* es la perteneciente al paquete de accesibilidad Java, existe asociada a todo objeto de interfaz gráfica Java y permite obtener entre otras características: el rol del componente (*getAccessibleRole()*), el número de hijos existentes (*getAccessibleChildrenCount()*) y su acceso a este, si está siendo mostrado (*isShowing()*), su tamaño (*getSize()*) y su posición (*getLocationOnScreen()*). Por lo tanto, desde el JFrame raíz, se van creando los diferentes *UIAElement* y *UIAWidget* asociados (en la Figura 5-6 este conjunto es denominado como un único objeto llamado *Widget*), introduciendo en cada uno de ellos las características de los componentes Java.

El resultado de este procedimiento encargado de profundizar en el árbol de componentes Java, que representan la interfaz gráfica de usuario, es la creación del árbol de widgets dentro de la herramienta TESTAR.

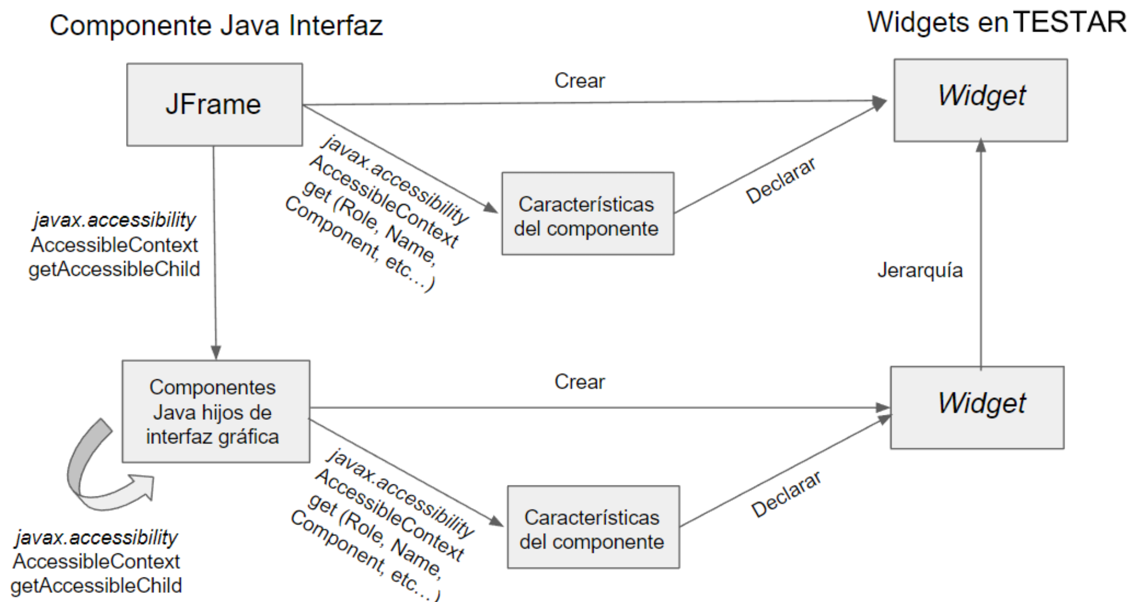

Figura 5-6: Uso de la accesibilidad Java para la creación de *Widgets* en base a los objetos Java de la interfaz gráfica, declarando además las características propias de los componentes y su jerarquía.

34 *Accessible Context* : https://docs.oracle.com/javase/7/docs/api/javax/accessibility/AccessibleContext.html

En las Figuras 5-7 y 5-8, se pueden observar las propiedades de un componente Swing dentro del *UIAState* creado, y se implementan sólo las propiedades principales que permiten a TESTAR realizar las pruebas de testeo. Por lo tanto, sólo se incluye información totalmente relevante como posición, rol, bloqueado (si está visible y se puede interaccionar), habilitado, antecesores y el título o nombre del componente; mientras que el resto de características se presentan como cero en caso de números, o *false* en caso de condicionantes lógicos.

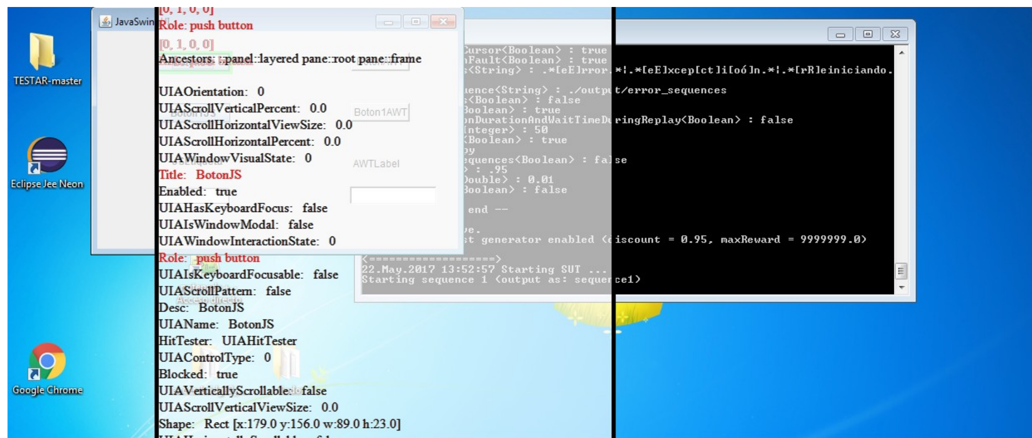

Figura 5-7: Propiedades de un *Widget* Swing nuestra interfaz simple.

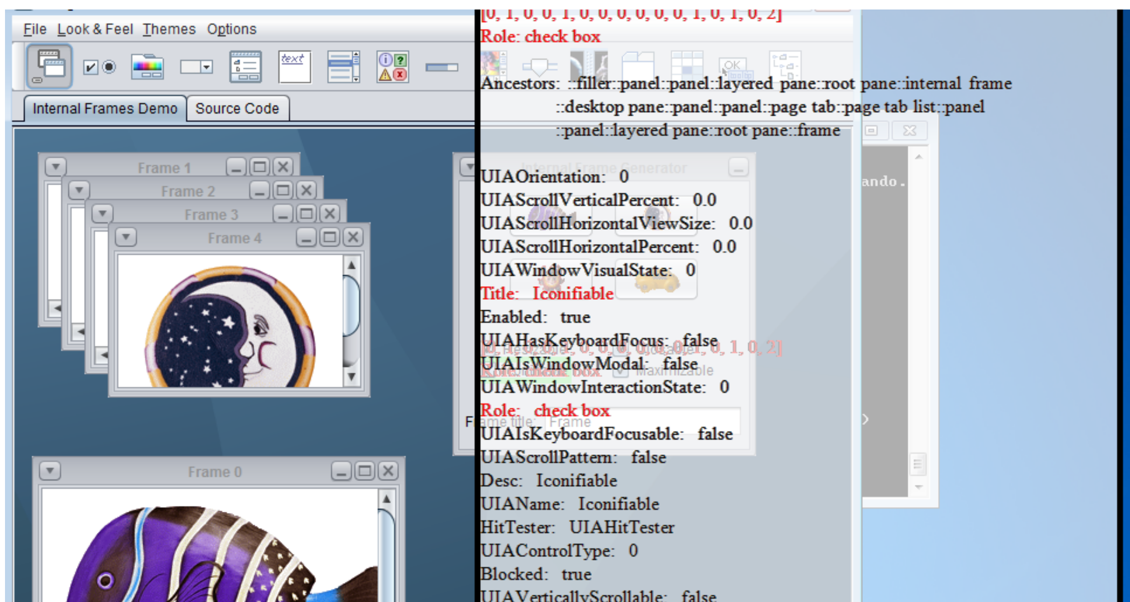

Figura 5-8: Propiedades de un *Widget* Swing en la demo SwingSet2.

5.2.3 Cambios necesarios

El uso de TESTAR para testear aplicaciones Java Swing no se debe restringir solamente a aquellas de las cuales se dispone de su archivo.jar, puesto que un posible campo de entrada como referencia a una aplicación puede ser también un proceso que se encuentre actualmente en ejecución. TESTAR dispone de implementaciones que le permiten detectar nuevos procesos que entren en ejecución, o búsquedas de procesos según su nombre o *PID*, por lo tanto esto no supone ningún cambio en cuanto a los parámetros de entrada actuales.

Sin embargo, esta forma de iniciar la herramienta dificulta la obtención del componente JFrame y supone un cambio grande en la implementación anterior. Desde la JVM donde se encuentra TESTAR, no es sencillo acceder a un objeto remoto en ejecución en una JVM distinta, tal y como se ha explicado en la parte 2.4.2 Procesos Java. Aun existiendo herramientas capaces de comunicar diferentes JVM, cabe recordar que no se tiene potestad para implementar en aplicaciones externas, las herramientas para interactuar con objetos remotos.

Al no disponer de una comunicación cliente-servidor entre las JVM, se debe recurrir a las llamadas al sistema mediante JNI. Esta era la forma inicial que tenía TESTAR para el reconocimiento de componentes, y sobre esta implementación se va a añadir la tecnología estudiada Java Access Bridge.

Para ello se activará el Access Bridge en el panel de accesibilidad del sistema (*Panel de control\Accesibilidad\Centro de accesibilidad\Facilitar el uso del equipo*), y se añadirá en TESTAR una carpeta con las *dll* y archivos C del Access Bridge, para su compilación junto al resto de la herramienta añadiendo en el *Makefile* su ruta (el archivo que contiene las directivas usadas en la compilación)[35]. Los métodos descritos en la API serán los necesarios para obtener desde la interfaz de la aplicación todos los componentes que se encuentren en ella, obteniendo las características básicas para su uso en TESTAR: rol, nombre, número de hijos, posición dentro de la interfaz y tamaño del componente.

35 Makefile: https://msdn.microsoft.com/en-us/library/windows/desktop/aa380049(v=vs.85).aspx

5.3. Implementación Java Access Bridge

Esta implementación se basa en la comunicación de dos JVM diferentes en ejecución, una de la aplicación Java Swing a testear, y la otra de la herramienta TESTAR, desde la cual, se debe realizar un conjunto de llamadas a la JVM que contiene los componentes Swing (4.1. Reconocer componentes de una JVM diferente a TESTAR).

Los archivos añadidos en TESTAR serán los ofrecidos por la compañía *Oracle* en la versión 2.0.2[36]. En el archivo *Makefile* de TESTAR, se incorpora la nueva ruta que contiene las clases necesarias para el uso del Access Bridge, siendo empleadas desde el archivo principal "main.cpp" que realiza las llamadas JNI.

El primer problema encontrado al comenzar con la implementación de los métodos descritos en la API[37] es la correcta utilización o activación del método *initializeAccessBridge()*. Es la función encargada de iniciar el Java Access Bridge, entre la aplicación TESTAR y el sistema, un método *boolean* que permite obtener la respuesta *true* en caso de ser posible esta inicialización.

La causa del problema reside en la forma correcta de emplear esta función, pues una simple llamada no sirve para el posterior uso de los diferentes métodos. Al igual que se puede observar en el código de las aplicaciones Java Monkey y Java Ferret, es necesario mantener un paso de mensajes (también conocido como *Handshaking*) entre las partes que harán uso del *bridge.*

Esta aclaración personal sobre la inicialización correcta del *bridge,* la encontré en un interesante post llamado "***Not receiving callbacks from the java access bridge***"[38], donde distintos usuarios se plantean una pregunta sobre si la forma de iniciar el *bridge* era incorrecta, o si el fallo residía en la introducción de una ventana Java adecuada. Todo esto ayudó a crear una inicialización correcta en TESTAR duplicando el mismo paso de mensajes que emplea Java Ferret, donde, finalmente, se crea un hilo de ejecución nuevo dentro de la herramienta, encargado de realizar la llamada al método de inicialización.

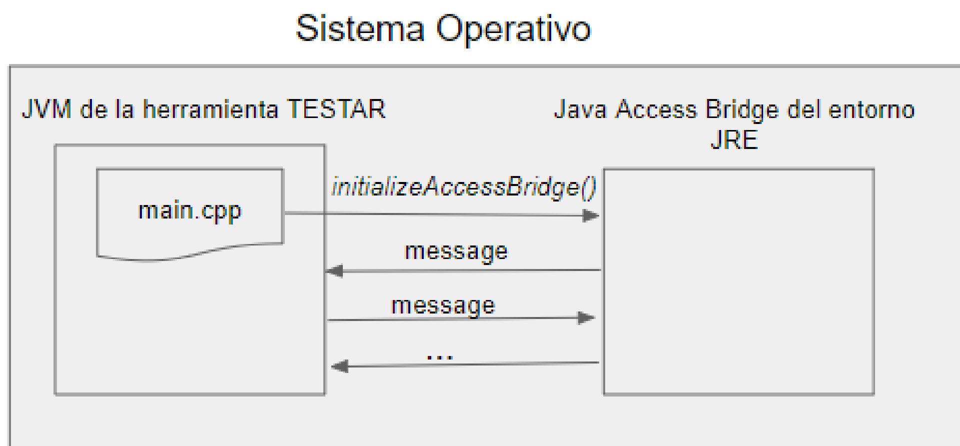

Figura 5-9: Paso de mensajes necesario en un hilo de ejecución de TESTAR para inicializar el Access Bridge.

36 Archivos empleados: http://www.oracle.com/us/technologies/java/jab-2-0-2-download-354311.html

37 API Access Bridge: http://docs.oracle.com/javase/accessbridge/2.0.2/api.htm

38 Aclaración para inicializar Access Bridge: https://stackoverflow.com/questions/1161142/not-receiving-callbacks-from-the-java-access-bridge

5.3.1 Obtención AccessibleContext & AccessibleContextInfo

La obtención de las propiedades necesarias de los componentes Java Swing se puede conseguir con el uso de cuatro funciones una vez el Access Bridge se encuentra activo; estas funciones serán invocadas desde el proyecto *windows* de TESTAR. La primera comprobación que se realiza es confirmar que se está trabajando con una ventana implementada en Java, gracias al método "*BOOL IsJavaWindow(HWND window)*"; esta ventana *hwnd* se obtiene en TESTAR como un *long*, es decir, la identificación numérica de la ventana en el sistema. Se puede realizar sin problemas una conversión o *casting* con el identificador para obtener la ventana, donde se confirma que se trata de una ventana Java, para continuar con las funciones descritas en el API.

Después se pueden emplear las tres funciones restantes para obtener el *AccessibleContext* de la ventana principal, y posteriormente todos los *AccessibleContext* hijos, para acceder de forma recursiva a sus propiedades asociadas. Para ello se emplean los métodos "*BOOL GetAccessibleContextFromHWND(HWND target, long *vmID, AccessibleContext *ac)*" y "*AccessibleContext GetAccessibleChildFromContext(long vmID, AccessibleContext ac, jint index)*", que permiten acceder de manera recursiva a todos los componentes. Además, a medida que accedemos a cada componente de la interfaz, se emplea la función "*BOOL GetAccessibleContextInfo(long vmID, AccessibleContext ac, AccessibleContextInfo *info)*", para conocer las propiedades de cada componente.

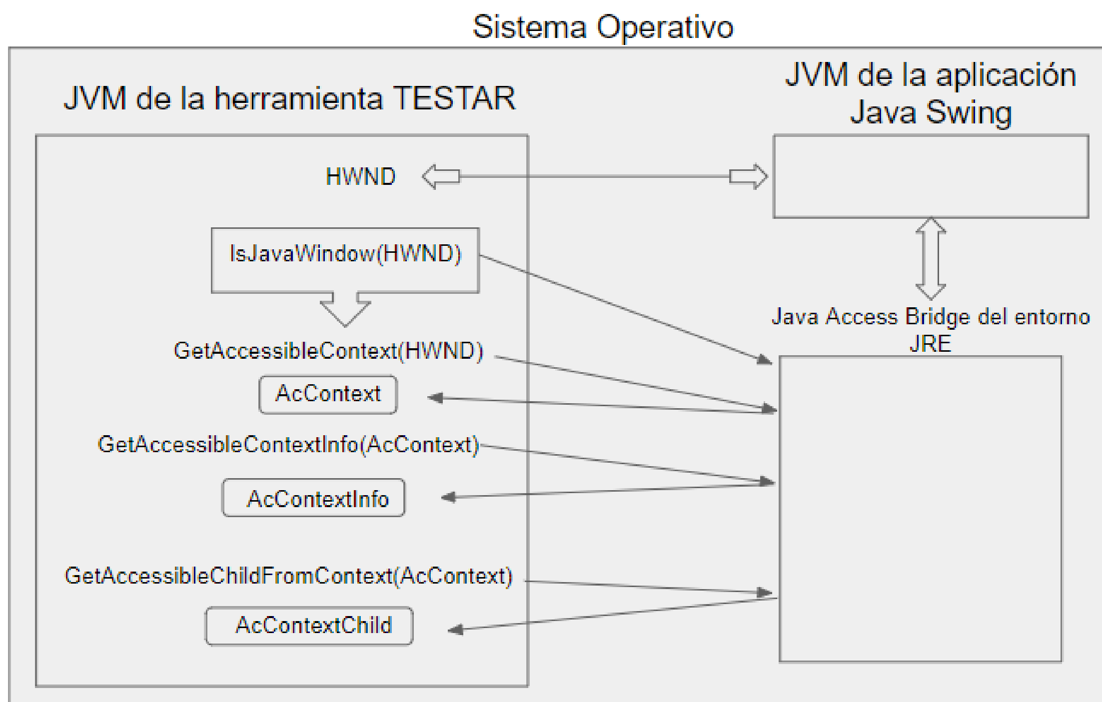

Figura 5-10: Funciones del Acces Bridge empleadas desde TESTAR

El objeto representado como AcContextInfo se obtiene de la clase *AccessibleContextInfo* del Access Bridge, y es el que permite obtener las propiedades de cada componente Java de la interfaz, concretamente: rol, nombre, número de hijos, componente padre, tamaño y posición del componente, y ciertos valores booleanos de propiedades de accesibilidad.

```
struct AccessibleContextInfo {
 wchar_ name[MAX_STRING_SIZE];        // the AccessibleName of the object
 wchar_ description[MAX_STRING_SIZE]; // the AccessibleDescription of the object
 wchar_ role[SHORT_STRING_SIZE];      // localized AccesibleRole string
 wchar_ states[SHORT_STRING_SIZE];    // localized AccesibleStateSet string
                                      //   (comma separated)
 jint indexInParent                   // index of object in parent
 jint childrenCount                   // # of children, if any
 jint x;                              // screen x-axis co-ordinate in pixels
 jint y;                              // screen y-axis co-ordinate in pixels
 jint width;                          // pixel width of object
 jint height;                         // pixel height of object
 BOOL accessibleComponent;            // flags for various additional
 BOOL accessibleAction;               // Java Accessibility interfaces
 BOOL accessibleSelection;            // FALSE if this object doesn't
 BOOL accessibleText;                 // implement the additional interface
 BOOL accessibleInterfaces;           // new bitfield containing additional
```

Figura 5-11: Propiedades de los componentes Java gracias al objeto *AccessibleContextInfo.*

Por cada AcContext obtenido de los componentes de la interfaz, se realizará la creación de un *UIAElement* y *UIAWidget* asociado (en la Figura 5-14 este conjunto es denominado como un único objeto llamado *Widget*), añadiendo de forma jerárquica estos objetos a medida que se vaya descendiendo en la estructura de componentes. Además, se declaran las propiedades de estos widgets y elementos, haciendo uso del objeto AcContextInfo obtenido de cada AcContext.

Al igual que en la implementación descrita Swing Reflection, el resultado es la creación del árbol de widgets en la herramienta TESTAR. Sin embargo, existe una diferencia importante respecto a la otra implementación, ya que no se trabaja directamente con los componentes Java, y se tiene la necesidad de transmitir las propiedades obtenidas desde las llamadas JNI al entorno JVM de TESTAR. Para ello, se hace una implementación básica empleando un archivo de texto como entrada/salida de información, que deberá ser modificada en el caso de elegir esta implementación como definitiva a integrar en TESTAR.

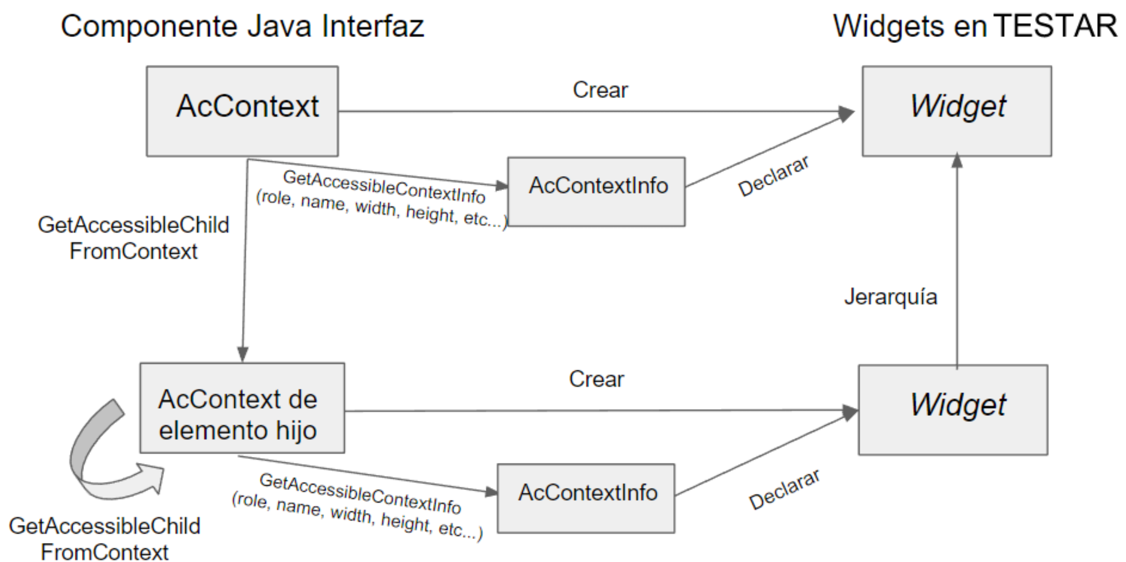

Figura 5-12: Creación de widgets en TESTAR en base a los componentes detectados de la interfaz de usuario, empleando los objetos devueltos por el Access Bridge.

6. Diferencias entre ambas implementaciones

A continuación, se muestran los resultados obtenidos al emplear las dos implementaciones descritas sobre la aplicación SwingSet2, para una comparación básica entre ambas. Para después, comentar los cambios y mejoras que deben realizarse en cada una, y elegir e integrar definitivamente una de ellas en TESTAR.

6.1. Evaluación inicial empleando ambas implementaciones

En ambas implementaciones, la funcionalidad de TESTAR, respecto a la generación de pruebas (modo *Generate*), permite el reconocimiento y realización de acciones sobre widgets. Este tipo de funcionalidad no se modifica en ninguna de las implementaciones realizadas en este proyecto, pero se hace uso de ella para comprobar los resultados concretos en la detección y uso de los componentes.

En los resultados de la evaluación simple realizada, con un conjunto de 20-50 acciones de usuario realizadas en varias secuencias, se obtienen unos *logs* (archivo.txt) que representan las información concreta de las acciones realizadas sobre cada widget, y un conjunto de gráficos que muestran los cambios de estado que ha sufrido la aplicación. El *log* obtenido tiene la estructura de la misma forma que la Figura 6-1, un ejemplo donde se muestran cuatro acciones ejecutadas por TESTAR.

```
Executed [3]: ACTION_n551353140 ROLE = LeftClickAt      TARGET = WIDGET_155495532               ROLE = scroll bar
                                SHAPE = Rect [x:374.0 y:293.0 w:192.0 h:15.0]   DESCRIPTION = Left Click at '<no description>'

Executed [4]: ACTION_1245134984 ROLE = LeftClickAt      TARGET = WIDGET_956676490               ROLE = toggle button
                                SHAPE = Rect [x:670.0 y:53.0 w:42.0 h:42.0]     DESCRIPTION = Left Click at '<no description>'

Executed [5]: ACTION_n1475316473 ROLE = LeftClickAt     TARGET = WIDGET_1702413962              ROLE = page tab         TITLE
                                SHAPE = Rect [x:455.0 y:98.0 w:89.0 h:23.0]     DESCRIPTION = Left Click at 'Source Code'

Executed [6]: ACTION_3626902 ROLE = LeftClickAt TARGET = WIDGET_n2061751862             ROLE = menu             TITLE = File
                                SHAPE = Rect [x:329.0 y:32.0 w:29.0 h:19.0]     DESCRIPTION = Left Click at 'File'      TEXT =
```

Figura 6-1: Ejemplo del *log* obtenido por TESTAR de la aplicación SwingSet2, donde se observa el rol del widget testeado, su posición y tamaño, y el identificador asociado a la acción.

Las siguientes figuras muestran una representación visual de los componentes reconocidos de la interfaz. La Figura 6-2 muestra las diferentes capturas que genera TESTAR de los componentes, cuando se ha elegido un widget para realizar una acción, y las Figuras 6-3 y 6-4 son gráficos donde se muestran los diferentes cambios de estado que ha sufrido la la aplicación.

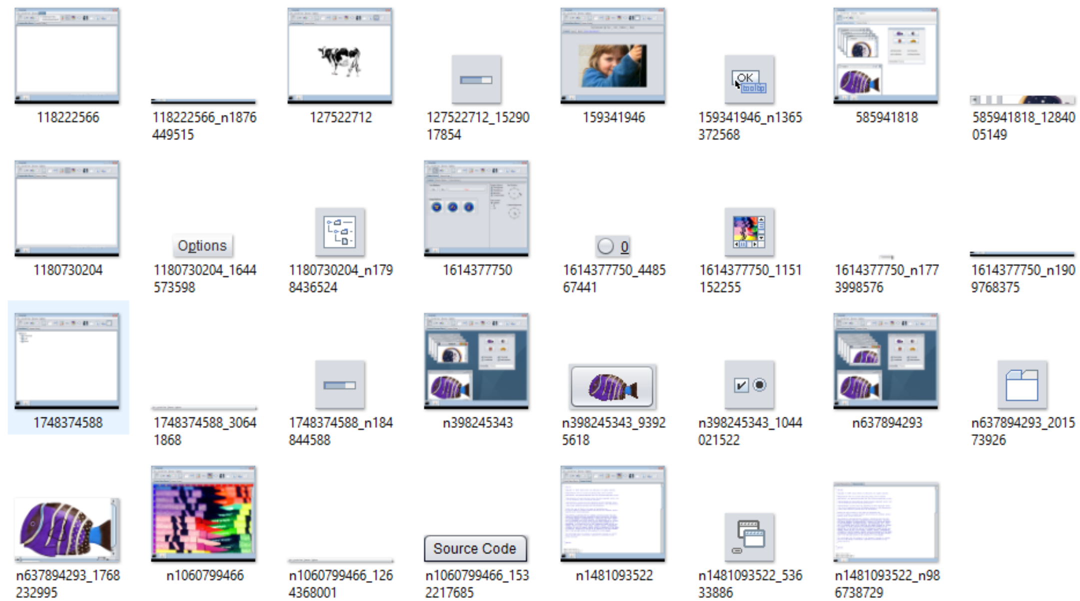

Figura 6-2: Capturas de algunos componentes activados por TESTAR en la aplicación SwingSet2.

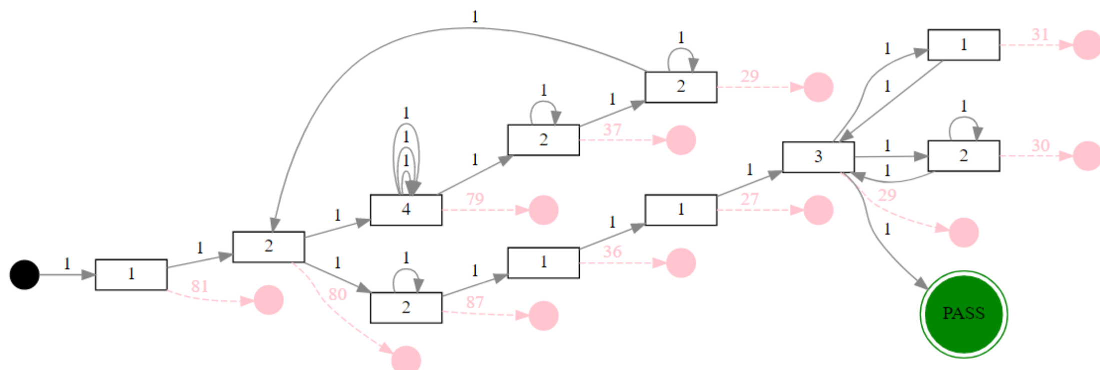

Figura 6-3: Cambios de estado de la aplicación SwingSet2 representados en forma de nodos.

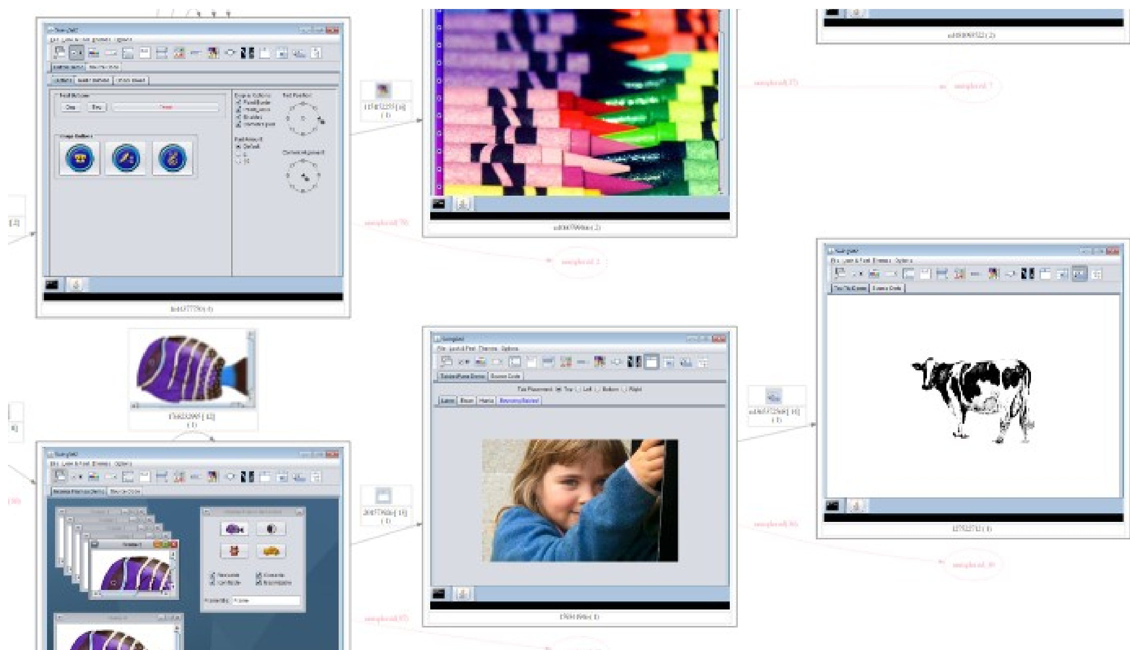

Figura 6-4: Cambios de estado en capturas de la aplicación SwingSet2 de los nodos generados.

6.2. Diferencias y selección final de una implementación

Las dos implementaciones preparadas para TESTAR, a la hora de reconocer componentes Java Swing de las interfaces, tienen un conjunto de cualidades diferentes una respecto a la otra, y en ambas se deben realizar mejoras y corrección de errores. Sabiendo que ambas pueden permitir el reconocimiento y generación de pruebas en TESTAR, se comentan sus diferencias principales y mejoras que requieren en mayor medida una solución.

6.2.1 Implementación Swing Reflection

- ✔ Incluye la posibilidad de conocer las acciones posibles a realizar en los componentes reconocidos, algo de gran importancia cuando se testean aplicaciones a través de la interfaz. En cada componente y su *AccessibleContext,* se puede emplear el método *getAccessibleAction()*, encargado de mostrar las acciones asociadas a cada componente.
- ✗ Se tiene la necesidad de trabajar con el archivo.jar que contiene la aplicación, además de conocer el nombre de la clase "main". Esto limita en gran medida el uso de nuestra herramienta, debido a las interacciones necesarias con procesos en ejecución.
- ✗ Es necesario añadir un parámetro de entrada adicional en la interfaz de TESTAR, encargado de indicar el nombre de la clase principal de la aplicación. Se emplea el *Manifest* incluido habitualmente en los archivos.jar, pero existe la posibilidad de no poder hacer uso de este.
- ✗ En ciertos componentes denominados *Frame* (tecnología AWT) existentes en la aplicación, se detecta que sus componentes hijos que representan las características de cerrar, minimizar o maximizar dicho componente, no se añaden correctamente en el árbol de widgets. Este problema ocurre a su vez en la implementación Access Bridge.
- ✗ En esta implementación funciona el reconocimiento de casi todos los componentes Java Swing sobre la aplicación SwingSet2, sólo se encuentran problemas con el componente denominado *Jdialog* (en la Figura 6-5 puede apreciarse uno de estos componentes). En una rápida investigación, se deduce que el problema reside concretamente en la creación de este componente como una nueva ventana adicional, por lo que se debe investigar la parte donde TESTAR realiza el enlace con este tipo de ventanas.

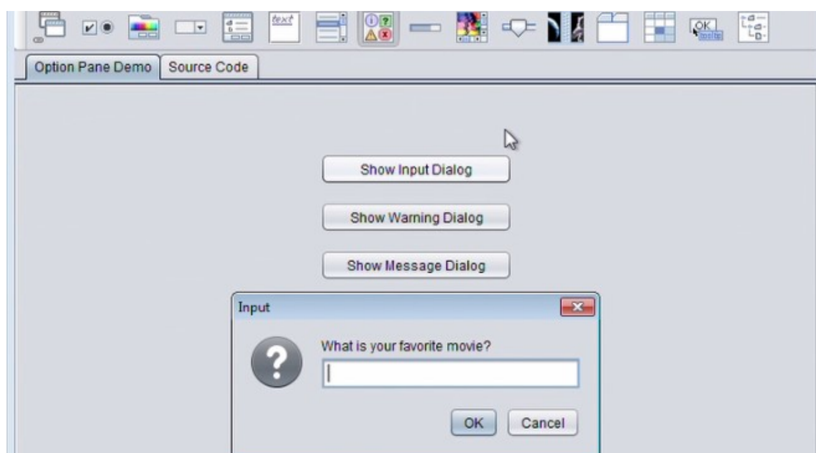

Figura 6-5: Ventana adicional de la aplicación que causa un error de reconocimiento en TESTAR.

6.2.2 Implementación Access Bridge

- ✔ Permite trabajar tanto con los procesos en ejecución de las aplicaciones Java que se encuentran en su propia JVM, como con los archivos que las contienen.
- ✔ Se trabaja con una tecnología y un conjunto de librerías proporcionada oficialmente por la entidad *Oracle*, lo que implica un conjunto de interacciones fiables y su importante implicación a la hora de actualizar y mejorar esta tecnología.
- ✔ A diferencia de la implementación Reflection, el reconocimiento del componente Swing *Jdialog,* que funciona como creación de una ventana adicional, se realiza correctamente con las funciones ofrecidas por el Access Bridge. Por lo tanto, no es necesario realizar ninguna modificación adicional en este aspecto.
- ✗ Necesita una mejora de comunicación entre las propiedades de los componentes obtenidas en las llamadas nativas y TESTAR. Una posible opción sería el uso del objeto Map, pues se puede crear la estructura jerárquica del árbol de widgets obtenido en las funciones *bridge* del JNI.
- ✗ En esta implementación, se ha encontrado un error de reconocimiento en un componente Swing que funciona como contenedor de otros (en la Figura 6-6 puede observase). Al tratarse de un contenedor, no se puede confirmar que el resto de componentes existentes dentro sean reconocidos correctamente, sin embargo, la implementación Reflection lo añade correctamente al árbol de widgets. Por lo tanto, se realiza una comparación en el reconocimiento de estos componentes empleando ambas implementaciones, para integrar en la implementación Access Bridge una solución.

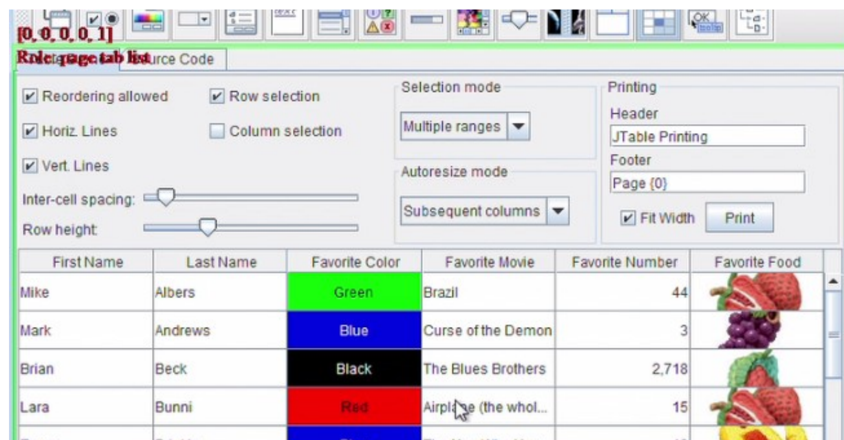

Figura 6-6: Componente Swing en la implementación Access Bridge, que no es añadido correctamente al árbol de widgets.

6.2.3 Elección de una implementación para su integración final en TESTAR

Se encuentran errores y modificaciones importantes a realizar en ambas implementaciones. Sin embargo, existe la necesidad de interactuar directamente con los procesos en ejecución de las aplicaciones Java, ya que no todas las entidades pueden ofrecer el conjunto de archivos que componen su aplicación. En estos casos se trabaja en un entorno donde lo único disponible es una JVM en el sistema, que contiene los recursos de la aplicación sobre la cual se debe realizar el testeo. Por lo tanto, se decide seleccionar la implementación Access Bridge, y realizar las modificaciones necesarias de los problemas mencionados anteriormente sobre TESTAR.

La implementación Reflection también será almacenada y empleada para la realización de pruebas, pues puede llegar a ofrecer un mayor conjunto de propiedades al trabajar directamente con los recursos de las interfaces Java Swing.

6.2.4 Modificaciones necesarias en la implementación Access Bridge

La comunicación empleada para transmitir el conjunto de propiedades obtenidas desde las llamadas JNI a la herramienta TESTAR, consistía en el uso de un archivo de texto, sobre el cual se realizaba una escritura y lectura en las recursivas llamadas que realiza TESTAR para actualizar su árbol de widgets. Sin embargo, al ser esta versión la elegida para una integración definitiva en TESTAR, se ha modificado este paso de parámetros de una manera adecuada.

En TESTAR se realiza la declaración de un mapa, que será creado en su JVM, encargado de asociar los posibles componentes Java Swing que se pueden obtener en el reconocimiento de interfaces, con el conjunto de componentes que empleaba TESTAR en la implementación Windows, gracias a la tecnología de accesibilidad UIAutomation. Es decir, se declara para cada componente Java Swing, el componente correspondiente de accesibilidad Windows (ej. "push_button" = "UIA_ButtonControlTypeId").

Por otro lado, desde el archivo encargado de realizar las llamadas al JNI "main.cpp", en el cual se obtienen las propiedades de los componentes Java Swing empleando el Access Bridge, se crean dos *arrays* en el entorno de ejecución de TESTAR, es decir, su JVM. El primer *Array* está formado por dos identificadores *long* asociados a la Id y *AccessibleContext* de la ventana interfaz, y el segundo contiene nueve Strings, que representan el conjunto de propiedades Java que nos ofrece el objeto *AccessibleContextInfo*. Estos *arrays* se van creando y usando en las recursivas llamadas que se realizan sobre la interfaz, para obtener el árbol de widgets activos en cada instante.

Además, los componentes que no presenten un correcto reconocimiento empleando el Access Bridge, pero sí empleando la implementación Reflection (como el problema mencionado de la Figura 6-6), pueden ser declarados manualmente en el mapa de asociación "Componente JavaSwing = Componente Windows".

7. Validación

7.1. Validación empleando aplicaciones Java Swing

Para validar correctamente la versión encargada de funcionar con aplicaciones Java Swing, se decide elegir un conjunto de aplicaciones a emplear como SUT y realizar sucesivamente mil acciones de usuario posibles sobre las interfaces a lo largo de 24h.

7.1.1 Selección de aplicaciones a testear

- SwingSet2 es una aplicación demo de Java, que incorpora una gran cantidad de componentes Swing diferentes. Ha sido empleada a lo largo de este trabajo para ir probando el reconocimiento de widgets sobre su interfaz y ha ayudado a detectar ciertos errores que eran necesarios corregir, tanto en esta versión Access Bridge como en las pruebas realizadas con la versión Reflection.
- JEdit[39] es una aplicación de edición de texto basada en Java que contiene componentes Swing, formada por cientos de personas en su desarrollo a lo largo de varios años. Esta es la segunda de las aplicaciones elegidas para testear como SUT, por lo que se descarga e instala la última versión estable para su uso, concretamente la versión 5.4.0
- CM-ED[40] es una herramienta cognitiva para la edición de mapas conceptuales basada en Java. Es la tercera de las aplicaciones seleccionadas, con la particularidad de incluir componentes que nos permiten "dibujar" sobre ellos; sin embargo, en este testeo de validación, no ha sido posible añadir todavía al conjunto de acciones de usuario la posibilidad de realizar un clic y arrastrar sobre un componente determinado.

7.1.2 Resultados finales obtenidos

Como resultado de esta validación, las secuencias obtenidas, de mil acciones cada una, se dividen en cuatro tipos:

- **Sequences_ok**, las secuencias realizadas que han mostrado un resultado correcto en lo referido al comportamiento de los SUT, tanto en las respuestas a las acciones como en el tiempo transcurrido que han tardado estas.
- **Sequences_suspicioustitle**, secuencias que muestran en algún momento como respuesta a una acción, un componente de la interfaz con un título sospechoso (*suspicious title*). Este título se introduce en los parámetros iniciales de TESTAR para emplearse en la funcionalidad de testeo *Generate*, por defecto se emplean los títulos "Error" y "Exception", pero el usuario que decida hacer uso de TESTAR puede añadir sin problemas otros títulos a su elección.

39 JEdit: http://www.jedit.org/

40 CM-ED: http://galan.ehu.es/Galan/es/products/cmed

- **Sequences_unexpectedclose,** secuencias que no llegan a completar todas las acciones indicadas debido a que la aplicación se ha cerrado como respuesta de alguna acción. Entre los parámetros que se introducen en TESTAR por defecto, se deshabilitan los componentes nombrados como “Cerrar” para evitar una acción sobre estos. Sin embargo, puede que existan otros no detectados previamente, y la aplicación se haya cerrado como una respuesta correcta y no de forma abrupta. Esto quiere decir que las secuencias de este tipo deben ser analizadas individualmente, para conocer si es una secuencia fallida o una secuencia cerrada como una respuesta correcta.
- **Sequences_unresponsive,** secuencias que no llegan a completar todas las acciones indicadas, debido a que TESTAR no detecta una respuesta en la aplicación para alguna acción realizada.

7.1.2.1 Secuencias finalizadas correctamente

En estas secuencias que no son abortadas por ningún tipo de error se pueden evaluar las métricas obtenidas, relacionadas con la cantidad de grafos/nodos obtenidos y acciones realizadas sobre estos, o el consumo de CPU durante el testeo. A continuación, se muestran unos ejemplos de gráficos obtenidos en el testeo del SUT SwingSet2, que muestran la correcta realización de acciones sobre componentes Java Swing.

Esto valida, en cuanto a la funcionalidad respecto a la tecnología Java Swing, que la mayoría de componentes son reconocidos y empleados por parte de TESTAR, permitiendo un conjunto de realización de acciones, las cuales provocan cambios de estado en la aplicación.

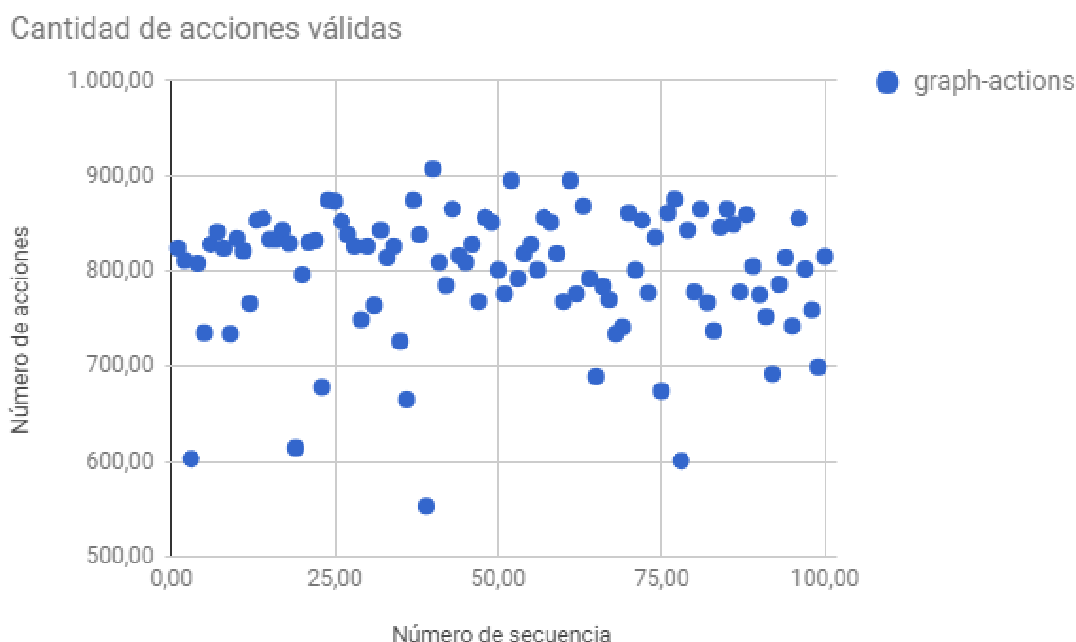

Figura 7-1: Gráfico obtenido del testeo de la aplicación SwingSet2, que representa el conjunto de acciones válidas realizadas a lo largo de cien secuencias de mil acciones. Se denominan acciones válidas aquellas que interaccionan con un widget de la interfaz, y causan un evento o cambio en la aplicación.

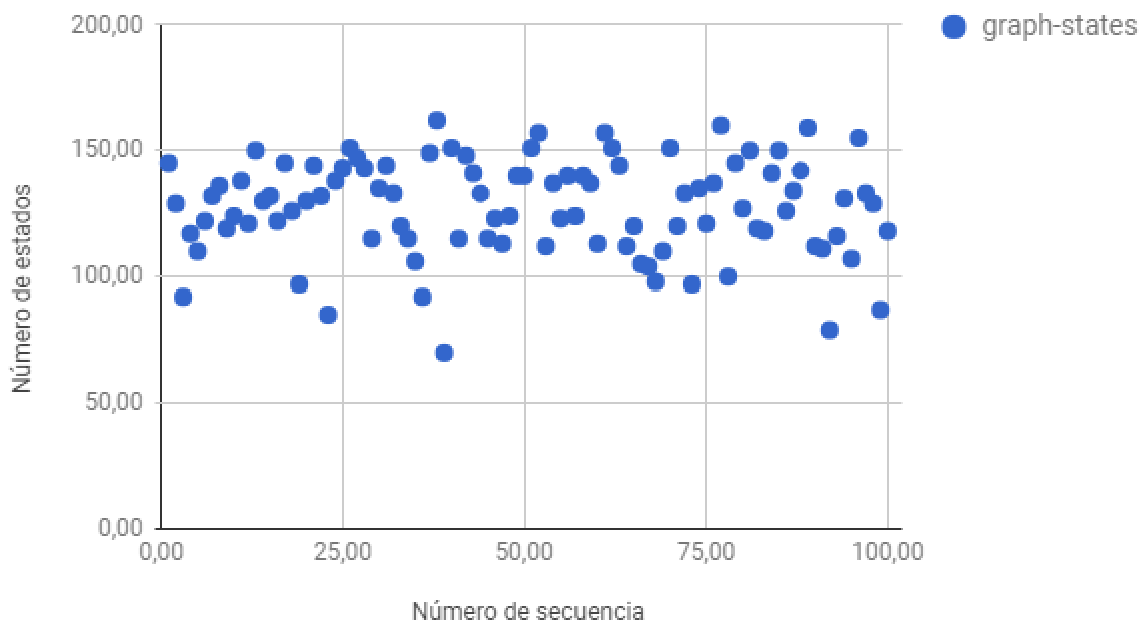

Figura 7-2: Gráfico obtenido del testeo de la aplicación SwingSet2, que representa el conjunto de estados/nodos obtenidos a lo largo de cien secuencias de mil acciones. Estos estados se obtienen cuando una acción válida sobre un widget causa un cambio de ventana en la interfaz de la aplicación.

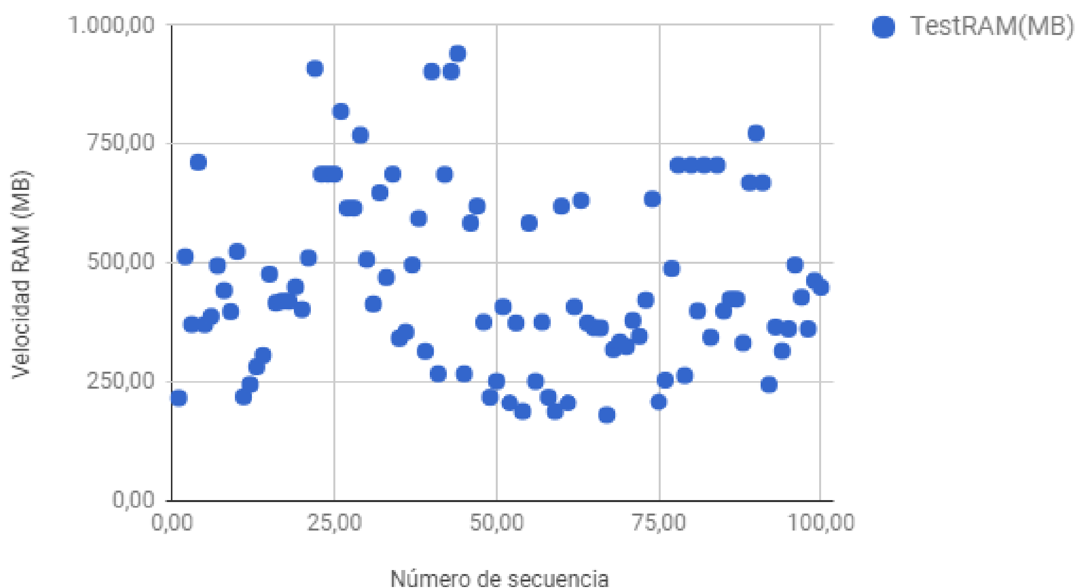

Figura 7-3: Gráfico obtenido del testeo de la aplicación SwingSet2, que representa la cantidad media de memoria RAM consumida en cada secuencia.

7.1.2.2 Secuencias con títulos sospechosos

En la aplicación JEdit empleada como SUT, se han obtenido varias secuencias con títulos sospechosos. En estos casos, lo ideal es comprobar la secuencia concreta y encontrar el componente que contiene el título de alerta, o su componente causante. A continuación, se muestra un error detectado correctamente por la herramienta JEdit, es decir, un falso positivo, donde se intenta crear un fichero con un nombre vacío y JEdit lo impide mostrando el error.

Este caso ayuda a validar el reconocimiento de varios componentes Swing y sus propiedades. Por una parte, se reconoce correctamente en el testeo la aparición de ventanas emergentes, como puede ser la ventana causante del error perteneciente a la Figura 7-4 o la propia ventana que indica el error en la Figura 7-5. Además, se reconoce correctamente tanto el botón de "Aceptar" de la Figura 7-4 para crear un fichero, como el título en la Figura 7-5 que contiene la palabra "Error".

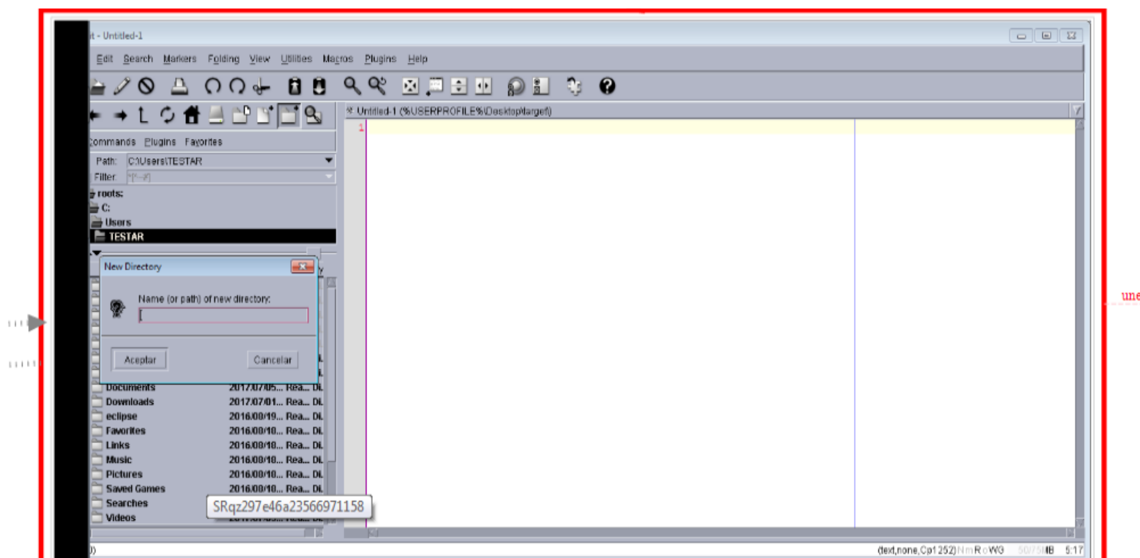

Figura 7-4: Secuencia del SUT testeado que va a causar la invocación de una ventana de error.

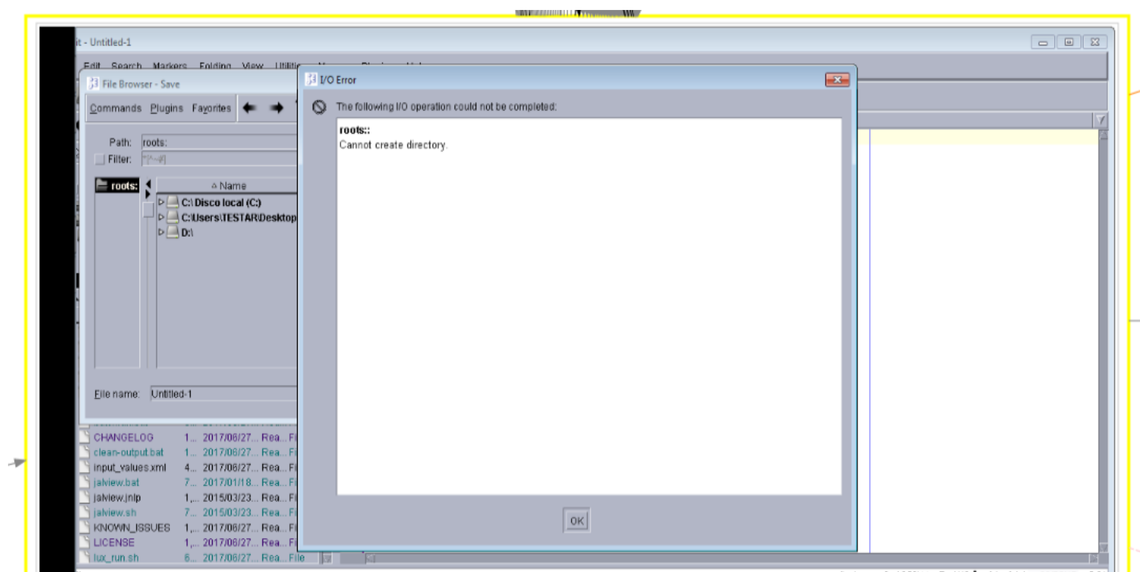

Figura 7-5: Secuencia del SUT testeado que ha invocación una ventana de error.

7.1.2.3 Secuencias que finalizan inesperadamente

Empleando la aplicación CM-ED como SUT se han obtenido varias secuencias que han finalizado la aplicación antes de alcanzar el número de acciones indicadas. En la Figura 7-6 la aplicación ha cambiado su posible idioma a euskera, y seleccionado la opción de “cerrar” en este idioma, por lo tanto, se trata de una respuesta correcta a una acción de usuario; y valida respecto a los componentes Swing, el recorrido sobre los distintos elementos que forman el menú de la aplicación.

En la Figura 7-7, el componente causante es un botón llamado “Aceptar” perteneciente a las opciones de configuración de la aplicación CM-ED. Respecto al funcionamiento de la aplicación, esta secuencia debe ser analizada para conocer si esta acción de usuario debe causar esta respuesta concreta de finalización. Y en relación con la validación de reconocimiento de componentes Swing, se vuelve a confirmar un recorrido sobre el menú de la aplicación, la selección de componentes sobre estos y el reconocimiento de ventanas emergentes.

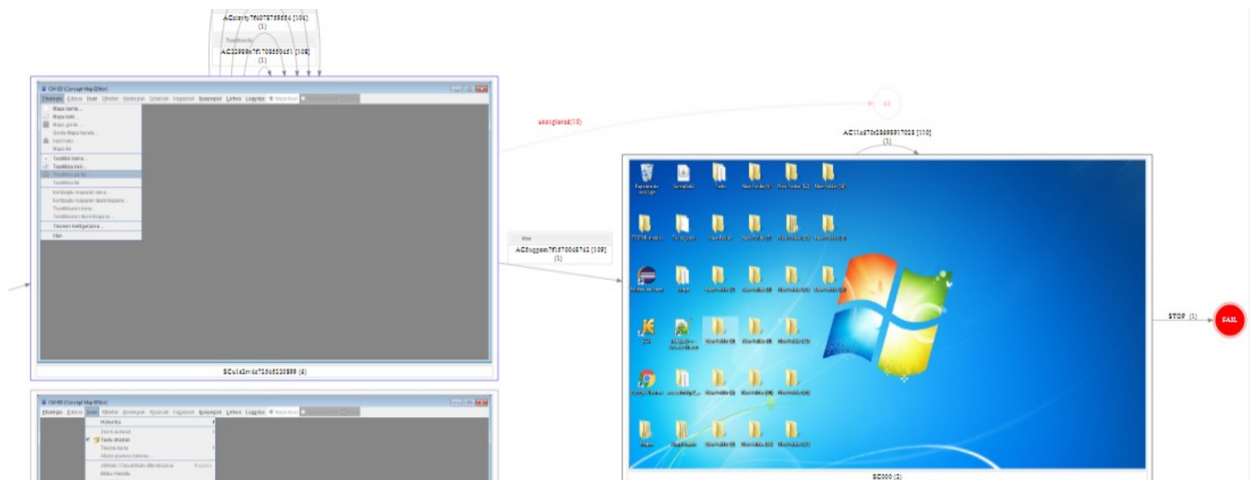

Figura 7-6: Secuencia del SUT testeado que ha finalizado la aplicación, mediante la interacción con un componente encargado de cerrar la aplicación.

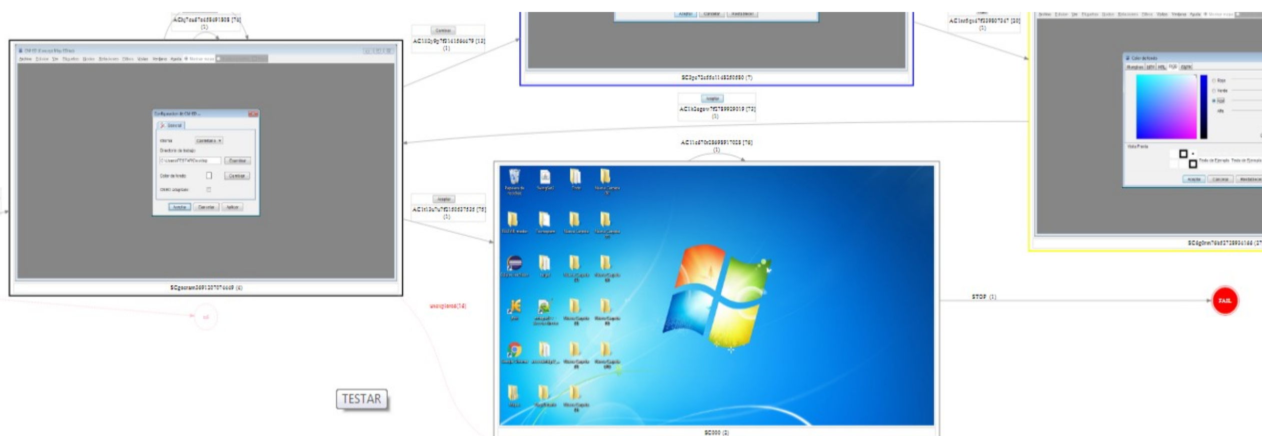

Figura 7-7: Secuencia del SUT testeado que ha finalizado la aplicación, mediante la interacción con un componente destinado a la configuración de la aplicación.

7.1.2.4 Secuencias que no responden correctamente

En las aplicaciones JEdit y CM-ED empleadas como SUT, se han obtenido varias secuencias que impiden a TESTAR la detección de respuestas al realizar un conjunto de acciones. Las siguientes figuras muestran los componentes causantes, en los cuales se debe averiguar si los errores se obtienen de una acción concreta sobre un componente, o de algún conjunto de acciones que haya sobrecargado la aplicación y causado un importante aumento de tiempo en lo referido a una respuesta por parte del SUT.

Respecto a la validación en el reconocimiento de componentes Swing, las Figuras 7-8 y 7-9 confirman la detección y realización de acciones, sobre los diferentes elementos que forman las aplicaciones. Pues estas secuencias erróneas se han obtenido gracias a la sucesión de diferentes acciones de usuario, sobre un conjunto concreto de componentes Swing.

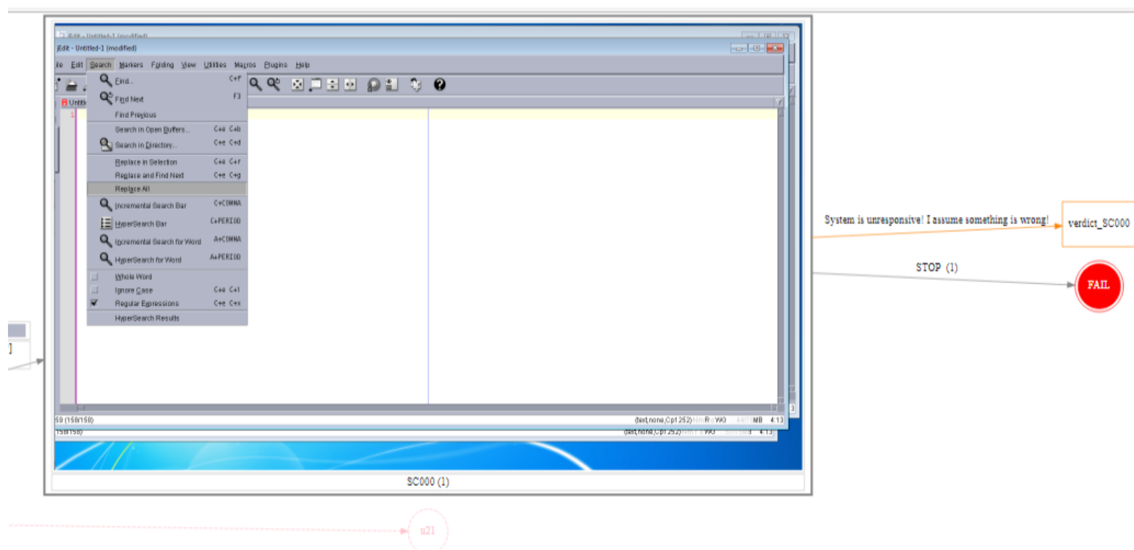

Figura 7-8: Secuencia del SUT que impide a TESTAR la obtención de respuestas, concretamente en la realización de una acción sobre el componente "Replace All".

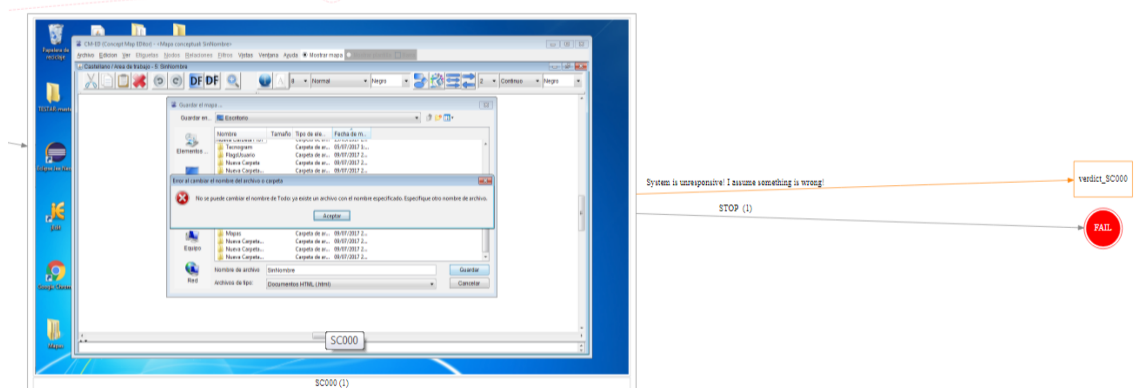

Figura 7-9: Secuencia del SUT que impide a TESTAR la obtención de respuestas, se muestra concretamente una ventana emergente de error que impide continuar realizando acciones de usuario.

8. Conclusiones

Para finalizar la realización de este trabajo, se hará un breve resumen donde se mencionen: las tecnologías investigadas, los objetivos cumplidos junto a la implementación final obtenida, así como las mejoras que deben seguir investigándose en un futuro. Además, se añadirá una sección dedicada a las aportaciones que me ha aportado este proyecto, tanto a nivel académico como a nivel personal.

8.1. Resumen del trabajo realizado

TESTAR es un software de código abierto destinado a testear de forma automatizada, a través de la interfaz gráfica de usuario, las aplicaciones de escritorio, web y móvil. Está preparado para trabajar en los principales sistemas operativos como Windows, Linux, Mac OS o Android, y emplea las tecnologías de accesibilidad propias de estos sistemas para realizar un reconocimiento de los distintos componentes que forman la interfaz.

En la ejecución de pruebas de testeo realizadas en ciertas empresas, que emplean para la programación de sus interfaces la biblioteca gráfica de Java llamada Swing, se detectó que dichos componentes Swing no eran correctamente reconocidos por la herramienta TESTAR. Por lo que las tecnologías de accesibilidad empleadas de los sistemas operativos muestran problemas de funcionalidad con la tecnología Java Swing.

Para la investigación y resolución de este problema, se ha trabajado en el sistema operativo Windows 7, empleando una extensión de la herramienta TESTAR diseñada para los sistemas Microsoft Windows. La tecnología de accesibilidad utilizada para el reconocimiento de componentes en estos sistemas Windows es el *framework* UIAutomation, usada por TESTAR en sus llamadas JNI realizadas al sistema operativo.

Como las interfaces Java Swing están basadas en un contenedor existente a nivel de sistema operativo, con la posterior inclusión de componentes creados completamente en Java, era necesario realizar una investigación de las tecnologías de accesibilidad existentes para los entornos Java. Además, es importante conocer que la ejecución de una aplicación Java, carga todos sus recursos en una JVM diferente a la de la herramienta TESTAR.

La tecnología descubierta y estudiada, que ha permitido la realización de este trabajo, es la denominada Java Accessibility, la cual ofrece un conjunto de clases y librerías que permiten el reconocimiento de componentes Java Swing de las interfaces gráficas de usuario. Se puede diferenciar, en el uso de esta tecnología, trabajar con los recursos de los componentes Swing en la misma JVM de la herramienta TESTAR y emplear el paquete *javax.accessibility;* o tener dos JVM distintas en el sistema y emplear la tecnología Java Access Bridge, destinada a exponer la API de accesibilidad Java en sistemas Microsoft Windows.

También se han estudiado, junto a la tecnología de accesibilidad Java, la aplicación Swing Explorer y las tecnologías Java Reflection, Class Loader y AWTEventListener, con la intención de ejecutar los recursos Java Swing de la aplicación a testear, en la misma JVM de la herramienta TESTAR.

Al principio se realizaron dos implementaciones basadas en la tecnología Java Accessibility, una de ellas denominada Reflection, que necesitaba la ubicación del archivo.jar que contiene la aplicación a testear, para ejecutar los componentes Swing y la herramienta TESTAR en la misma JVM; y la otra basada en la tecnología Java Access Bridge, en el caso de tener dos JVM en ejecución. Sin embargo, se ha comprobado que la implementación Reflection no ofrece la interacción con los procesos en ejecución de las aplicaciones a testear, algo necesario en ciertas empresas, que no pueden ofrecer el conjunto de archivos que forman sus aplicaciones.

Finalmente, la implementación con el Access Bridge ha sido la elegida para su integración en TESTAR, la cual permite obtener y asociar las propiedades de los diferentes componentes Java Swing que forman las interfaces de usuario, tanto de procesos Java en ejecución como de archivos contenedores de la aplicación a testear.

Sobre esta implementación final, se ha realizado una validación de funcionalidad respecto al reconocimiento de componentes Swing e interacción sobre estos, en las cuales se detectan un conjunto de desafíos en el reconocimiento de algunos componentes, sobre los que se debe realizar una futura investigación. Por lo tanto, la realización de este proyecto, finaliza como el comienzo de una versión en TESTAR, que permite a esta herramienta el reconocimiento e interacción con los componentes basados en en la tecnología Java Swing.

8.2. Futuras investigaciones

La validación final sobre la versión Access Bridge integrada en TESTAR muestra un conjunto de desafíos en el reconocimiento de algunos componentes, sobre los que se debe realizar una futura investigación.

- En los componentes denominados *Frame* de la aplicación SwingSet2, sus componentes hijos que representan las características de cerrar, minimizar o maximizar el componente, no se añaden en el árbol de widgets. En pruebas con la aplicación Java Ferret, se confirma que este tipo de componentes se pueden detectar, por lo que se debe seguir investigando empleando esta aplicación de *Oracle*.

- Antes de la realización de este trabajo, lo único que detectaba TESTAR al intentar reconocer una interfaz Java Swing era el contenedor AWT (2.3.2 Componentes ligeros). Sin embargo, con esta implementación donde se debería reconocer este contenedor AWT perteneciente al lenguaje Java, se ha comprobado que no es añadido correctamente al árbol de widgets. La solución es comprobar el correcto reconocimiento de componentes AWT, y, en caso de no existir ningún problema en este reconocimiento Java, comparar esta nueva implementación Java Swing con la anterior a la realización de este trabajo.

- A medida que se van realizando pruebas en diferentes aplicaciones con componentes Java Swing, pueden surgir ciertos componentes no declarados correctamente en TESTAR (donde se asocian los componentes Java Swing y los componentes de UIAutomation). Para ello, se puede hacer uso de las herramientas Java Monkey y Java Ferret, o de la implementación Reflection en caso de decidir seguir con su desarrollo.

En las futuras investigaciones a realizar sobre TESTAR, es necesario mencionar a las empresas interesadas en esta herramienta, como puede ser el caso de EVERIS. La gran cantidad de pruebas que se pueden realizar en sus numerosas aplicaciones benefician tanto a las empresas, en el caso de detectar fallos relacionados con respuestas erróneas de sus aplicaciones como al desarrollo de este software de testeo, ya que se pueden encontrar nuevas mejoras a integrar respecto el reconocimiento Swing, o surgir nuevas tecnologías que impliquen mejorar la funcionalidad de TESTAR.

8.3. Aprendizaje personal en la realización del trabajo

La realización de este proyecto a nivel general ha servido como experiencia de implicación en un caso real de software que requiere la necesidad de investigación, estudio y solución, respecto a los problemas encontrados. Cabe añadir, también, el interés mostrado por diferentes empresas como EVERIS, a la hora de emplear TESTAR para comprobar la correcta funcionalidad de sus aplicaciones.

Concretando las diferentes tecnologías investigadas, he aprendido sobre la necesidad y utilidad de las tecnologías de accesibilidad que diseñan las empresas informáticas, tanto para ayudar a personas con discapacidad a emplear tecnología software en su vida cotidiana, como para ofrecer a los programas de testeo un acceso a la información de los componentes existentes. Además, he adquirido una serie de conocimientos respecto a los entornos Java, ya sea sobre el funcionamiento de ejecución de una Máquina Virtual Java y las posibles formas de comunicación entre ellas; o a un conjunto de clases Java diseñadas para cargar ciertos recursos desde una JVM específica.

El futuro de este software y las posibles investigaciones a realizar, me motivan a seguir en el desarrollo de la herramienta TESTAR. Por una parte, la validación final del proyecto muestra un conjunto de desafíos por resolver sobre ciertos componentes, y, por otra, para probar e investigar la funcionalidad de la aplicación en otros sistemas operativos diferentes a Microsoft Windows.

9. Bibliografía

[1] *TESTAR*. Software Testing Innovation Alliance. Url: http://www.testar.org/

[2] Mireilla Martinez, Anna Esparcia-Alcazar, Urko Rueda, Tanja E. J. Vos, Carlos Ortega: Automated Localisation Testing in Industry with Test*., 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, pp241-248, 2016

[3] U. Rueda, T.E.J. Vos, F. Almenar, M. Oreto, A. Esparcia: TESTAR – from academic protoype towards an industry-ready tool for automated testing at the User Interface level. Jornadas de Ingeniería de Software y Bases de Datos, JISBD 2015, September 2015, Santander.

[4] Evaluating rogue user testing in industry: an experience report. S. Bauersfeld, A. de Rojas, and T. E. J. Vos. In Proceedings of 8th International Conference RCIS. IEEE, 2014.

[5] Evaluating the TESTAR tool in an Industrial Case Study. S. Bauersfeld, T.E.J. Vos, N. Condori-Fernández, A. Bagnato and E. Brosse. In Proceedings of the 8th ACM EEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2014, Industrial Track, Torino, 2014.

[6] Tanja E.J. Vos, Peter M. Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. 2015. TESTAR: Tool Support for Test Automation at the User Interface Level. Int. J. Inf. Syst. Model. Des. 6, 3 (July 2015), 46-83. DOI=http://dx.doi.org/10.4018/IJISMD.2015070103

[7] Jiantao Pan. *Software Testing*. Carnegie Mellon University, Spring 1999.
Url:https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

[8] *Software that makes software better*. The econosmist, Mar 6th 2008.
Url:http://www.economist.com/node/10789417

[9] *Glosario de Informática e Internet. Url: http://www.internetglosario.com/*

[10] Dr. Tanja E. J. Vos. *FITTEST European Project*. 2010-2013.
Url:http://crest.cs.ucl.ac.uk/fittest/project.html

[11] *Java Native Interface Specification*. Oracle Corporation.
Url:http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

[12] *Inspect.exe (Windows)*. Microsoft Windows SDK.
Url:msdn.microsoft.com/en-us/library/windows/desktop/dd318521(v=vs.85).aspx

[13] *Abstract Window Toolkit (AWT)*. Oracle Corporation.
Url:http://docs.oracle.com/javase/1.5.0/docs/guide/awt/

[14] Amy Fowler. *A Swing Architecture Overview.* Oracle Corporation. Url:http://www.oracle.com/technetwork/java/architecture-142923.html

[15] Sharon Zakhour and Anthony Petrov. *Mixing Heavyweight and Lightweight Components.* Oracle Corporaton, April 2010.

Url:http://www.oracle.com/technetwork/articles/java/mixing-components-433992.html

[16] Robert Eckstein. *Java SE Application Design With MVC.* Oracle Corporation, March 2007. Url:http://www.oracle.com/technetwork/articles/javase/index-142890.html

[17] *Java Platform, Standard Edition 7 API Specification.* Oracle Corporation. Url:https://docs.oracle.com/javase/7/docs/api/overview-summary.html

[18] *Java Accessibility Utilities.* Oracle and Sun Microsystems. Url:https://docs.oracle.com/cd/E17802_01/j2se/javase/technologies/accessibility/docs/jaccess-1.3/doc/index.html

[19] *Java Accessibility API.* Oracle Corporation. Url:http://docs.oracle.com/javase/7/docs/technotes/guides/access/jaapi.html

[20] chrisadamson. *Using Swing's Pluggable Look and Feel Blog.* Oracle Community, 22th Frebuary 2004. Url:https://community.oracle.com/docs/DOC-983327

[21] *Java SE Desktop Accessibility.* Oracle Corporation. Url:http://www.oracle.com/technetwork/articles/javase/index-jsp-136191.html

[22] *Java Access Bridge 2.0.2.* Oracle Corporation. Url:http://www.oracle.com/technetwork/articles/javase/index-jsp-136191.html

[23] *¿Qué es un archivo dll?.* Microsoft. Url: https://support.microsoft.com/es-es/help/815065/what-is-a-dll

[24] nmquan2504. Java Ferret. Url: https://github.com/nmquan2504/JavaFerret

[25] robotframework. SwingExplorer. Url:https://github.com/robotframework/swingexplorer

[26] iaavo. SwingSet2. Url: https://github.com/iaavo/swingset2

[27] *The Reflection API.* Oracle Corporation. Url:https://docs.oracle.com/javase/tutorial/reflect/index.html

[28] *Java Concurrency.* Oracle Corporation. Url:https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

[29] *Interprocess Communications.* Microsoft. Url: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx

[30] *Java RMI.* Oracle Corporation. Url: https://docs.oracle.com/javase/tutorial/rmi/

[31] *Java JMX*. Oracle Corporation. Url: https://docs.oracle.com/javase/tutorial/jmx/

[32] *Apache Ant 1.10.1 Manual*. Apache Software Foundation. Url:https://ant.apache.org/manual/

[33] *Makefiles*. Microsoft. Url: https://msdn.microsoft.com/en-us/library/windows/desktop/aa380049(v=vs.85).aspx

[34] *Not receiving callbacks from the java access bridge.* Stack Overflow Community. Url: https://stackoverflow.com/questions/1161142/not-receiving-callbacks-from-the-java-access-bridge