UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación

Universitat Politècnica de València

Cobertura de código: más información para evaluar TESTAR

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Autor: Antonio Martín López-Asúnsolo

Tutoras: Tanja E.J. Vos, Anna I Esparcia-Alcazar

2016-2017

Resumen

La herramienta TESTAR permite realizar testing de forma automatizada a través de las interfaces gráficas de las aplicaciones software. Sin embargo, no es posible saber si esta herramienta ha testeado adecuadamente una aplicación determinada. Si al realizar pruebas sobre una aplicación se identifican todos los errores que contiene, podemos concluir que la tarea de testing se ha realizado correctamente. Sin embargo, el problema reside precisamente en saber si se han localizado dichos errores en su totalidad. El hecho de que TESTAR realice el testing a través de la interfaz de las aplicaciones implica que dicha aplicación es vista como una caja negra en la que no es posible conocer su estructura interna, limitando así la capacidad de testear. La cobertura de código es una medida porcentual en las pruebas de software que mide el grado en que el código fuente de un programa ha sido comprobado. Es comúnmente utilizada en pruebas de caja blanca, como las pruebas unitarias, en las que sí se tiene acceso al código y estructura del software que se está testeando. La cobertura de código es de gran utilidad a la hora de evaluar la calidad de las pruebas realizadas. Esta medida no es proporcionada por TESTAR y hasta el momento no ha sido investigada. En este trabajo se tratará de analizar la cobertura de código alcanzada por las pruebas generadas mediante TESTAR sobre diferentes aplicaciones.

Palabras clave: TESTAR, testing, automatizado, GUI, cobertura de código, SUT, JaCoCo.

Abstract

The TESTAR tool allows automated testing through the graphical interfaces of the software applications. However, it is not possible to know if this tool has properly tested a particular application. If all bugs are identified during the tests in an application, we can conclude that the testing task was successful. However, the problema is precisely to know if all these errors have been located entirety. The fact that TESTAR performs the testing through the interface of the applications implies that said application is seen as a black box in which it is not possible to know its internal structure, thus limiting the ability to test. Code coverage is a percentage measure in software testing that measures the degree to which the source code of a program has been tested. It is commonly used in white box tests, such as unit tests, in which you can access the code and structure of the software being tested. Code coverage is very useful when evaluating the quality of tests performed. This measure is not provided by TESTAR and so far has not been investigated. In this paper we will try to analyze the code coverage reached by the tests generated by TESTAR on different applications.

Keywords: TESTAR, testing, automated, GUI, code coverage, SUT, JaCoCo.

Tabla de contenidos

Índice de figuras

1. Introducción

El testing/testeo supone una tarea indispensable dentro del campo de la ingeniería del software [1]. En el proceso de desarrollo de cualquier aplicación es necesario realizar pruebas, entre otros, sobre su interfaz gráfica de usuario (GUI), ya que será la forma en la que se permita al usuario interaccionar con el software. Al realizar pruebas sobre la interfaz, se están realizando a su vez sobre un gran conjunto de componentes del sistema relacionados entre sí, en lugar de comprobar su correcto funcionamiento por separado como ocurriría con los tests unitarios. Así se podrá descubrir la existencia de fallos de comunicación entre los elementos que conforman la aplicación, o argumentar la ausencia de los mismos. No obstante, puede ser una tarea larga y tediosa testear una aplicación mediante su interfaz, aún más si tenemos en cuenta que éstas pueden verse alteradas de forma habitual por parte de los desarrolladores. Este hecho supondría para los testers no solo implementar numerosos casos de prueba, sino también realizar el mantenimiento de los mismos. Con el fin de disminuir el tiempo que supondría realizar pruebas de forma manual han aparecido diferentes herramientas de testeo automático que permiten realizar fácilmente dicha tarea. Entre estas herramientas tenemos TESTAR [2], la cual es resultado de investigaciones realizadas por el Research Center on Software Production Methods (PROS) del Departamento de Sistemas informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia (UPV).

La herramienta TESTAR permite realizar testing de forma automatizada a través de las interfaces gráficas de las aplicaciones. Para lograr su objetivo, TESTAR se sirve de la API de accesibilidad proporcionada por el sistema operativo [3]. Con ella, TESTAR puede reconocer los distintos elementos presentes en la interfaz y sus correspondientes propiedades. Una vez identificados estos elementos la herramienta genera un conjunto de posibles acciones para el actual estado de la interfaz, selecciona una de ellas y la lleva a cabo. Debido a esta forma de actuar, la secuencia de testeo no es previamente definida y posteriormente ejecutada, sino que se diseña y ejecuta al mismo tiempo. Con la sucesiva ejecución de casos de prueba y la interacción con los distintos elementos que conforman la GUI se espera poder detectar fallos en la aplicación o poder asegurar la ausencia de los mismos. En los trabajos [4][5][6][7] se hizo uso de TESTAR sobre diferentes aplicaciones, tras lo cual se detectaron diversos errores de relevancia hasta el momento pasados por alto por las empresas responsables.

Sin embargo, el número de errores descubiertos en una aplicación durante su testeo no supone una información lo suficientemente significativa para determinar la calidad de las pruebas realizadas. El hecho de no encontrar fallos durante la ejecución de las pruebas siempre es algo positivo, pero no es indiferente si se ha testeado la mayor parte de la aplicación o si se ha testeado un pequeño porcentaje. Aunque el objetivo final del testing es encontrar fallos en el software no basta con prestar atención únicamente a la cantidad de errores hallados, se deben tener en cuenta otras métricas de calidad de software, como la cobertura de código alcanzada [8]. Por estos motivos se realizarán a lo largo de este proyecto análisis relativos a distintas aplicaciones que

han sido testeadas mediante TESTAR. Con esto se persigue el objetivo de responder a las preguntas de investigación que se plantearan posteriormente.

Una vez que TESTAR finaliza la generación de acciones sobre la GUI se generan una serie de datos o métricas que proporcionan información sobre las pruebas efectuadas. De esta forma podemos obtener información variada, como por ejemplo el número de veces que la aplicación bajo testeo falló o produjo errores, el número de acciones que se realizaron o el número de estados visitados de la GUI, entendiendo el término estado como el conjunto de elementos que forman la interfaz (barras de herramientas, botones, imágenes, etc.) en un momento determinado. Además de esta información, sería interesante conocer la cobertura de código alcanzada por las pruebas realizadas mediante TESTAR. Este dato no es proporcionado por la herramienta y hasta el momento no ha sido investigado.

En este trabajo se tratará de analizar la cobertura de código alcanzada durante la ejecución de diferentes tests gracias a la herramienta JaCoCo [9]. La cobertura de código supone una de las métricas de calidad de testing de caja blanca más conocidas. Durante este proyecto se experimentará con una combinación entre pruebas de caja blanca y pruebas de caja negra. Esto es debido a que TESTAR realiza pruebas sobre distintas aplicaciones sin tener en cuenta su estructura interna ni detalles de implementación (pruebas de caja negra) para posteriormente, sirviendonos del software JaCoCo, poder obtener y estudiar la cantidad de código cubierta por estas pruebas (pruebas de caja blanca).

La estructura de este trabajo es la siguiente. En la siguiente sección se estudiará de una manera más ampliada la importancia de las pruebas de software junto a diferentes conceptos relevantes. En la sección 3 se especificará el objetivo propuesto así como la metodología empleada para su resolución. Posteriormente se hará una valoración de los resultados obtenidos para finalmente obtener las conclusiones sobre las cuales se plantearán futuras líneas de trabajo.

2. Contexto: software testing

Errare humanum est. Ya lo decían los antiguos: errar es humano. También en la Edad Moderna, una de las principales premisas del método científico fue el "ensayo y error". Desde tiempos inmemoriales hemos sabido que los errores son inherentes a la naturaleza humana. Con eso contamos. Es fácil darse cuenta de que aplicamos el principio de prueba y error a ámbitos tan cotidianos de la vida como la cocina o las relaciones personales. No obstante, tal vez sea en el sector informático donde la necesidad de realizar pruebas es más acuciante y determinante. Además, el ritmo al que se trabaja en este sector y las exigencias de los clientes hacen que las empresas de software deban lanzar versiones en tiempo récord, lo que implica cometer fallos. Adelantarse a la competencia es esencial, sea al precio que sea. Y el precio suelen ser los errores.

El objetivo de las pruebas de software es presentar información objetiva sobre la calidad del producto a las personas responsables de éste. Las pruebas de calidad presentan los siguientes objetivos: encontrar defectos o bugs, aumentar la confianza en el nivel de calidad, verificar que el producto es apto para el uso, facilitar información para la toma de decisiones, evitar la aparición de defectos.

El testing es una fase muy importante en el ciclo de desarrollo de software. Tan importante que debería introducirse al inicio del proceso para que los errores no se acumulen en la fase final de entrega. Las empresas de software surgen y desaparecen en la misma medida en la que conceden la importancia debida a su área de control de calidad y testing. Veamos algunas razones que explican claramente la relevancia de esta tarea [10].

- Es absolutamente esencial para identificar los errores que se han cometido en las fases de desarrollo.
- Garantiza que el software es fiable y asegura la satisfacción del cliente.
- Garantiza la calidad del producto, lo que en última instancia permite fidelizar al cliente.
- Reduce los costes de mantenimiento.
- Puede resultar muy costoso corregir los errores a posteriori o en fases más avanzadas del desarrollo de la aplicación. El coste de subsanar los errores hallados en etapas tempranas del desarrollo es 200 veces menor que el de aquellos identificados de forma tardía.
- Es vital para mantenerse en el negocio. Recordemos que los programas de software cada vez tienen más competencia, son más complejos y cuentan con más usuarios. Esto significa que hoy en día una empresa sobrevive sólo si puede ofrecer un producto fiable y de calidad.
- Ofrece a los comerciales de la compañía la confianza que necesitan a la hora de vender el producto, es decir, pueden estar seguros de que no tiene defectos y proporciona las funcionalidades que se supone debe proporcionar. Al fin y al cabo, el objetivo de una empresa es vender su producto.

Las pruebas de software se pueden realizar en diferentes niveles. Entre estos niveles podemos destacar algunos de los que gozan de mayor popularidad [11]:

- **Pruebas Unitarias**

Las pruebas unitarias tienen como objetivo verificar la funcionalidad y estructura de cada componente individualmente una vez que ha sido codificado. Constituyen la prueba inicial de un sistema y las demás pruebas deben apoyarse sobre ellas. Los pasos necesarios para llevar a cabo las pruebas unitarias son los siguientes:

- Ejecutar todos los casos de prueba asociados a cada verificación establecida en el plan de pruebas, registrando su resultado. Los casos de prueba deben contemplar tanto las condiciones válidas y esperadas como las inválidas e inesperadas.
- Corregir los errores o defectos encontrados y repetir las pruebas que los detectaron. Si se considera necesario, debido a su implicación o importancia, se repetirán otros casos de prueba ya realizados con anterioridad.

La prueba unitaria se da por finalizada cuando se hayan realizado todas las verificaciones establecidas y no se encuentre ningún defecto, o bien se determine su suspensión.

- **Pruebas de Integración**

El objetivo de las pruebas de integración es verificar el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente con el fin de comprobar que interactúan correctamente a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes. Los tipos fundamentales de integración son los siguientes:

Integración incremental

Se combina el siguiente componente que se debe probar con el conjunto de componentes que ya están probados y se va incrementando progresivamente el número de componentes a probar. Con el tipo de prueba incremental lo más probable es que los problemas que surjan al incorporar un nuevo componente o un grupo de componentes previamente probado, sean debidos a este último o a las interfaces entre éste y los otros componentes.

Integración no incremental

Se prueba cada componente por separado y posteriormente se integran todos de una vez realizando las pruebas pertinentes. Este tipo de integración se denomina también *Big-Bang*.

- **Pruebas de Sistema**

Las pruebas del sistema tienen como objetivo ejercitar profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el

funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.

- **Pruebas de Aceptación**

Las pruebas de aceptación son definidas por el usuario del sistema y preparadas por el equipo de desarrollo, aunque la ejecución y aprobación final corresponden al usuario.

Estas pruebas van dirigidas a comprobar que el sistema cumple los requisitos de funcionamiento esperado, recogidos en el catálogo de requisitos y en los criterios de aceptación del sistema de información, y conseguir así la aceptación final del sistema por parte del usuario.

Al margen de los diferentes niveles en los que se diseñan pruebas de software, estas pueden agruparse en dos grandes grupos:

- Pruebas de caja negra

Las pruebas de caja negra [12], es una técnica de pruebas de software en la cual la funcionalidad se verifica sin tomar en cuenta la estructura interna de código, detalles de implementación o escenarios de ejecución internos en el software. En las pruebas de caja negra el interés reside en las entradas y salidas del sistema, sin preocuparnos en tener conocimiento de la estructura interna del programa de software. Para obtener el detalle de cuáles deben ser esas entradas y salidas, nos basamos únicamente en los requerimientos de software y especificaciones funcionales.

- Pruebas de caja blanca

Las pruebas de caja blanca [13] se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente. El ingeniero de pruebas escoge distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida adecuados. Al estar basadas en una implementación concreta, si ésta se modifica, por regla general las pruebas también deberán rediseñarse. Aunque las pruebas de caja blanca son aplicables a varios niveles —unidad, integración y sistema—, habitualmente se aplican a las unidades de software. Su cometido es comprobar los flujos de ejecución dentro de cada unidad (función, clase, módulo, etc.) pero también pueden probar los flujos entre unidades durante la integración, e incluso entre subsistemas, durante las pruebas de sistema.

2.1. Test automatizado

Dentro de la ingeniería del software, el testing automatizado es uno de los campos que está en auge debido a la gran utilidad que está teniendo aplicado a la industria [14]. Es considerado un campo complicado ya que diseñar casos de prueba no es una tarea trivial y debe de realizarse por un trabajador experimentado. A causa de los beneficios

que puede aportar la automatización de casos de prueba es considerado un área que merece y requiere tiempo y esfuerzo para ser explotado.

De manera general la ejecución de pruebas automatizadas conlleva un menor gasto de tiempo que las manuales ya que permiten realizar pruebas repetidamente de forma continuada. Como excepción, cabe mencionar que si lo que se desea es realizar una cantidad pequeña de pruebas, el tiempo requerido para la preparación de su automatización puede no compensar al tiempo que se ahorra al automatizar dichas pruebas.

Uno de los mayores problemas de las herramientas automatizadas está en el diseño y desarrollo de comportamientos complejos. Es decir, supone una gran dificultad diseñar un comportamiento que proporcione siempre buenos resultados, ya que cada software es diferente a los demás y lo que puede dar buenos resultados en uno no tiene por qué darlos en otro.

2.2. Testeo de interfaces gráficas de usuario

Las interfaces gráficas de usuario suponen aproximadamente entre el 45% y el 60% [15] del código de una aplicación. La interfaz supone el medio de interacción entre el usuario y los servicios que proporciona el software en cuestión. Por este motivo realizar pruebas a través de la GUI supone una manera eficaz de detectar errores en el funcionamiento de la aplicación, por lo que resulta interesante generar pruebas a través de la GUI que permitan garantizar la calidad del software.

Puede suponer una tarea compleja y tediosa realizar pruebas manuales a través de la GUI de las aplicaciones, principalmente debido al tamaño que pueden alcanzar. Supongamos los siguientes casos que nos permitan imaginar la complejidad de las pruebas:

- En formularios, los valores que debe introducir el usuario son muy susceptibles, ya que a menudo se suelen almacenar en bases de datos o tienen ciertas restricciones, como que solo puedan introducirse números. Si hubiera que comprobar al menos un elemento de todos los posibles en cada caso de prueba se podría perder mucho tiempo. Sin embargo, mediante la automatización únicamente habría que crearlo una vez y se podría ejecutar cualquier número de veces.
- Elementos invisibles. En ocasiones para facilitar las pruebas, los desarrolladores crean elementos invisibles que sirven para facilitar las pruebas, pero cuyo acceso no debería estar permitido a los usuarios finales de la aplicación. Con el uso de estas herramientas se facilitaría su localización.
- Enlaces entre ventanas. Las GUI a menudo suelen estar conectadas entre sí, de manera que sería adecuado comprobar todos los enlaces. Sin embargo el número de enlaces puede ser inmenso, por lo que si esto se hace de forma automática el tester puede ahorrar mucho tiempo.

Actualmente multitud de empresas están introduciendo herramientas de testing automatizado en sus procesos. Existen diferentes herramientas que automatizan el testing de las aplicaciones a nivel de interfaz gráfica de usuario, como TESTAR, Selenium, Murphy tools. La clasificación de estas herramientas varía en función de la fuente consultada. En este trabajo la dividiremos en tres grupos: “Capture and replay”, “Visual testing” y “Traversal testing”.

En la actualidad la mayor parte de las herramientas de testeo para las GUI se encuentran dentro del grupo de Capture and replay. Con el uso de estas herramientas se permite al usuario almacenar las acciones realizadas en forma de scripts de manera que puedan repetirse posteriormente las veces deseadas. Este tipo de herramientas gozan de popularidad debido mayormente a que son intuitivas y fáciles de utilizar. Por otra parte, tienen un gran inconveniente, ya que los scripts pueden fallar fácilmente si la interfaz gráfica de usuario es alterada por algún cambio por pequeño que sea. Esto supone que se tenga que invertir tiempo en el mantenimiento de estos scripts en vez de crear otros nuevos.

Para corregir los problemas anteriormente planteados han surgido el Visual testing y Traversal testing. El Visual testing o testeo visual se sirve del reconocimiento de imágenes para poder interpretarlas y traducirlas a la estructura interna de la aplicación. Al utilizar el reconocimiento visual, las herramientas de este grupo proporcionan una mayor resistencia si se cambia la interfaz, pero tienen otros inconvenientes. Por ejemplo, como el único medio de extraer información son imágenes, la obtención de información es un proceso más complicado, principalmente si se desea información concreta de la estructura interna de la herramienta.

Las herramientas pertenecientes al grupo de Traversal testing, como TESTAR, inspeccionan la GUI para poder comprobar sus propiedades generales. Mediante el estado de la GUI se obtiene la estructura de la aplicación. Esta estructura no será fija, se irá calculando constantemente. De esta forma es posible comprobar los elementos existentes en la interfaz para posteriormente poder interactuar con ellos. TESTAR es una herramienta perteneciente a este grupo, la cual se sirve de la API de accesibilidad de Microsoft para poder obtener la estructura de la aplicación a través de su interfaz.

2.3. TESTAR

TESTAR es una herramienta de testeo automatizado que realiza pruebas a través de la interfaz gráfica de usuario de la aplicación que se va a testear (o software under test, SUT). Para generar estas pruebas utiliza la API de accesibilidad proporcionada por Microsoft. Esta API permite reconocer los diferentes elementos que conforman la GUI y proporciona información sobre los mismos, como qué tipo de elemento es, posicionamiento en la interfaz, identificador, etc. TESTAR se servirá de toda esta información para generar la representación interna del SUT, que almacena de manera jerárquica los diferentes componentes del estado actual de la interfaz. El estado de la

interfaz es el conjunto de elementos y sus correspondientes atributos en un determinado instante de tiempo. Dicho estado se ve modificado de forma automática en cada momento en que TESTAR interactúa con el SUT. Cada interacción supone una acción.

TESTAR divide su ejecución en conjuntos de acciones agrupadas en secuencias. Cada ejecución tiene al menos una secuencia y cada secuencia tiene un número determinado de acciones que debe llevar a cabo. Tanto el número de secuencias como el número de acciones por secuencia deben de especificarse en TESTAR antes de comenzar su ejecución.

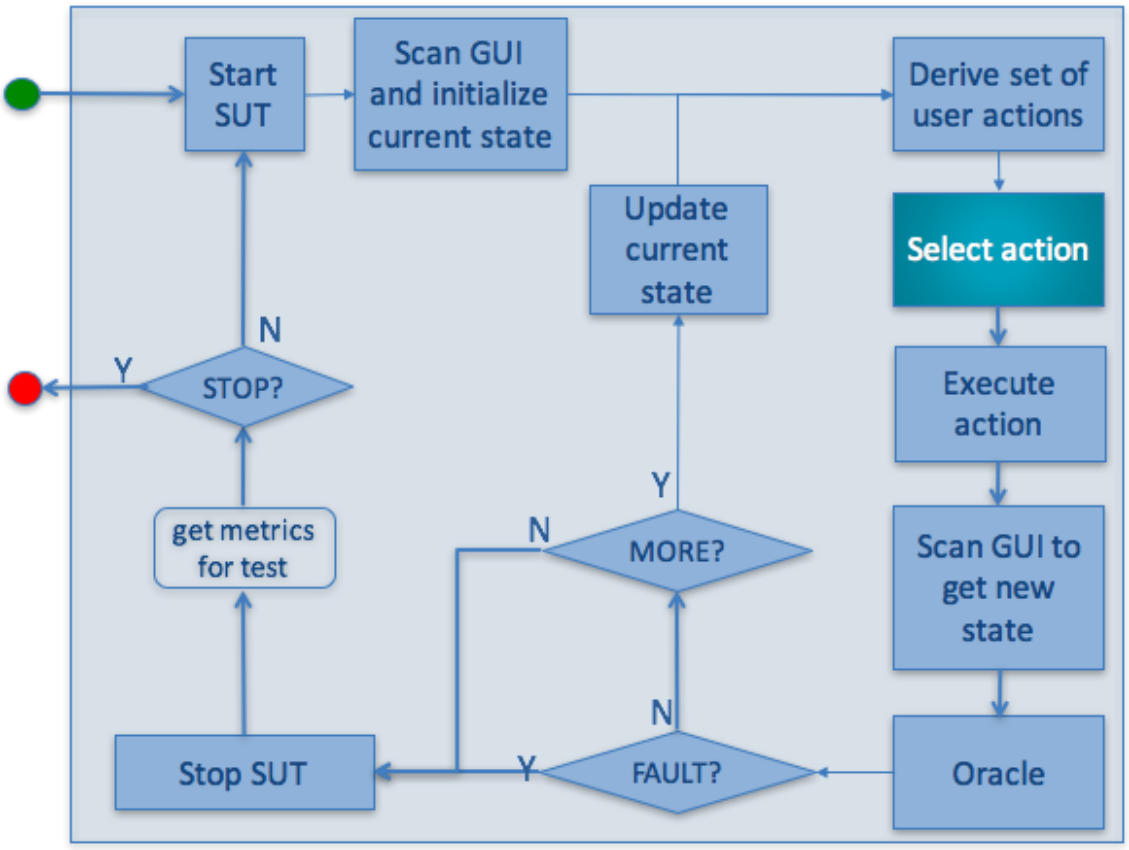


Figura 1: Ciclo de TESTAR

En la Figura1 podemos apreciar el funcionamiento de la herramienta. En ciclo de ejecución es un bucle que finaliza al encontrar un error o si se ha alcanzado cierto criterio de parada, el cual será el número de acciones que tenían que llevarse a cabo o el tiempo empleado.

El funcionamiento de TESTAR está separado en dos fases. La primera es manual y consiste en adaptar TESTAR al SUT y la segunda consiste en la ejecución de las pruebas.

La primera fase empieza definiendo los oráculos y los filtros. Un oráculo es un conjunto de instrucciones que sirven para comprobar si la ejecución de las acciones ha causado algún error en el SUT. Los filtros sirven para establecer determinadas acciones que queremos evitar durante la ejecución de las pruebas, por ejemplo que se pulse sobre el botón 'Cerrar' del SUT. Para poder identificar estos elementos sobre los que deseamos que TESTAR no interactúe es conveniente que previamente nos sirvamos del

modo Spy de la herramienta para posteriormente proceder a la ejecución de las pruebas. Al ejecutar el modo Spy sobre el SUT en cuestión podremos obtener información de los elementos de la interfaz sobre los que se encuentre el cursor del ordenador. De esta forma sabremos cuál es el título identificativo del elemento con el que se desea evitar la interacción para poder añadirlo a los filtros y así asegurarnos que no se realizarán acciones sobre el elemento en cuestión una vez que comience la ejecución de los tests.

También es necesario tener en cuenta que TESTAR ofrece tres maneras distintas de conectar la herramienta con el SUT correspondiente mediante la propiedad SUTConnector:

- Command_line: SUTConnectorValue debe ser una línea de comando que inicie la ejecución del SUT. Ejemplo: java –jar SUTs/calculadora.jar
- SUT_WINDOW_TITLE: SUTConnectorValue debe de ser el título presente en la ventana principal del SUT. De esta forma el SUT debe de ser iniciado y cerrado manualmente.
- SUT_PROCESS_NAME: SUTConnectorValue debe de ser el nombre del proceso del SUT. De esta forma el SUT debe de ser iniciado y cerrado manualmente.

Tras la primera fase comienza la segunda y comienza el funcionamiento de TESTAR, mediante el cual se producirá el escaneado de los elementos de la interfaz. De esta manera se construye un árbol donde aparecen representados todos los elementos de la GUI relacionados mediante jerarquías. Cada interacción con el SUT provoca que el árbol se genere nuevamente con los elementos de la interfaz actualizados. A partir del árbol la herramienta determina un conjunto con todas las posibles acciones que es posible efectuar en el estado en que se encuentra. TESTAR utiliza uno de sus algoritmos de selección para determinar de entre todas las posibles acciones cuál es la que va a ejecutar. En este trabajo tendremos en cuenta y estudiaremos los resultados de cobertura alcanzado por cada uno de sus cinco algoritmos:

- Random [2]: como su nombre indica, selecciona de manera aleatoria la acción que se efectuará sobre la GUI.
- Random+ [16]: este algoritmo comprueba si hay acciones disponibles que no hayan sido ejecutadas. Si se da este caso, realiza la acción sobre uno de los elementos con los que no se ha interactuado. En el caso contrario selecciona de forma aleatoria la siguiente acción de entre todas las disponibles.
- Qlearning [17]: es una técnica de aprendizaje que aplica recompensas y penalizaciones por cada acción ejecutada. Al encontrarse la GUI en un determinado estado, se debe determinar qué acción derivará en otro estado en el que reciba una mayor recompensa. Las acciones que no hayan sido ejecutadas hasta el momento generarán una recompensa mayor que las que ya se hayan ejecutado. Podrán realizarse penalizaciones en los casos en los que se elija una acción que ya ha sido ejecutada varias veces. Esta técnica incorpora la capacidad de prever, para cada posible estado futuro, la cantidad de acciones ejecutadas y no ejecutadas disponibles en dicho estado futuro.
- Qlearning+ [18]: nuevamente se comprueba si hay acciones disponibles que no hayan sido ejecutadas. Si se da este caso, realiza la acción sobre uno de los

elementos con los que no se ha interactuado. En el caso contrario selecciona la acción que obtenga una mayor recompensa de la manera que se explicó anteriormente.

- Maxcoverage [19]: trata de explorar la GUI lo máximo posible. Es el algoritmo más novedoso de entre los existentes, con lo que esta será una oportunidad para ponerlo en práctica y estudiar su utilidad.

Seguidamente se ejecuta la acción elegida, tras lo cual se comprueban las posibles condiciones de parada:

- Duración o tamaño de las pruebas. TESTAR requiere que se le indique el número máximo de secuencias y acciones que va a ejecutar y la cantidad de tiempo del que dispone para realizar las pruebas. Cuando se alcancen cualquiera de estos límites TESTAR finaliza las pruebas.
- Error. Se comprueba si ha ocurrido algún error en la ejecución. Pueden existir diversos motivos por los que salten estos errores, como que el SUT no siga funcionando o que haya aparecido una excepción o algún error. Antes de que TESTAR comience la ejecución de las pruebas existe la posibilidad de configurar el parámetro 'ForceToSequenceLength' a true o false. Para las pruebas generadas en este trabajo este parámetro ha estado establecido siempre a true, lo que implica que aunque se produzca un error en el SUT, TESTAR no finalizará la generación de tests y tratará de continuar hasta que se alcance el número de secuencias y acciones establecidas.

Para cada uno de los algoritmos anteriormente mencionados de TESTAR habrá dos maneras de actuar; una forma será utilizando el método denominado getTopWidgets y la otra forma será no utilizarlo. Esto implica que al estar ejecutándose pruebas sobre el SUT, si se utiliza el método getTopWidgets y en el estado actual hay alguna ventana emergente o se ha pulsado sobre algún elemento desplegable, únicamente se permitirá realizar acciones sobre los elementos presentes en dicha ventana emergente o en dicho elemento desplegable. Si no se utiliza el método getTopWidgets y en el estado actual se da alguno de los casos mencionados, TESTAR podrá interactuar con cualquier elemento de la interfaz.

2.4. Cobertura de código

La cobertura de código es una medida porcentual en las pruebas de software que mide el grado en que el código fuente de un programa ha sido comprobado. Sirve para determinar la calidad del test que se lleve a cabo y para determinar las partes del código que no han sido comprobadas y las partes que ya lo fueron. Se dice que una línea de código está cubierta si dicha línea ha sido ejecutada en la realización de su test correspondiente. Al tener una alta cobertura de código se puede afirmar que gran parte del código de la aplicación está siendo probado y por consiguiente se puede tener cierta certeza sobre el correcto funcionamiento de la aplicación. Al mismo

tiempo medir la cobertura puede ser útil para detectar y eliminar código innecesario en la aplicación, ya que es código que no se ejecuta. Sin embargo, si al realizar una prueba se alcanza una cobertura de código del 100% esto no garantiza la ausencia de errores o bugs en el código fuente de la aplicación, simplemente nos indica que el código ha sido cubierto por las pruebas en su totalidad. La cobertura de código es comúnmente utilizada para evaluar la calidad de las pruebas unitarias. Sin embargo en este proyecto la utilizaremos para evaluar las pruebas ejecutadas mediante TESTAR, las cuales podemos catalogar como pruebas de sistema. Entre las diferentes métricas existentes de cobertura de código tenemos:

- Instrucciones: La unidad más pequeña son las simples instrucciones de código de byte. La cobertura de instrucciones da información sobre la cantidad de código que ha sido ejecutada o no.
- Ramas: Esta métrica cuenta el total de ramificaciones (diferentes caminos según sentencias “if”o “switch”) que hay en un método y determina cuantas se han ejecutado y cuantas no. Por lo tanto, este criterio se cumple cuando cada decisión se evalúa a true y false al menos una vez. El manejo de excepciones no se considera en esta métrica.
- Complejidad ciclomática: La complejidad ciclomática es el número mínimo de caminos que pueden, en combinación (lineal), generar todas las rutas posibles a través de un método. Por lo tanto, el valor de la complejidad puede servir como una indicación para el número de casos de prueba de unidad para cubrir completamente una determinada pieza de software.
- Líneas: Una línea se considera ejecutada cuando se ha ejecutado al menos una instrucción asignada a esta línea.
- Métodos: Cada método no abstracto contiene al menos una instrucción. Se considera que un método se ejecuta cuando se ha ejecutado al menos una instrucción.
- Clases: Una clase se considera ejecutada cuando al menos uno de sus métodos ha sido ejecutado. Los constructores así como los inicializadores estáticos son considerados métodos. Como los tipos de interfaz Java pueden contener inicializadores estáticos, tales interfaces también se consideran como clases ejecutables.

3. Objetivo y metodología

Anteriormente hemos recalcado la importancia de poder recolectar y analizar información útil para determinar la calidad de las pruebas realizadas mediante TESTAR. En este proyecto nos concentramos en a la obtención de los registros de ***cobertura de código*** alcanzada durante la ejecución de tests, lo cual no es proporcionado por la heramienta. Por consiguiente, el primer objetivo será concebir y desarrollar una infraestructura reutilizable que permita ejecutar TESTAR sobre diferentes aplicaciones para posteriormente obtener información acerca de la cobertura de código relativa a las pruebas efectuadas.

Una vez que se alcance este objetivo, será de interés poner a prueba los diferentes algoritmos disponibles en la herramienta (descritos en la sección anterior: Random, Random+, Qlearning, Qlearning+ y MaxCoverage), así como realizar pruebas en las que se alterne el uso del método getTopWidgets y la ausencia del mismo. Con este tipo de estudios y análisis de los datos obtenidos se tratará de determinar cuál es la manera más adecuada de configurar los diferentes parámetros de TESTAR para lograr mayores ratios de cobertura. Para la agrupación de estos objetivos se plantean las siguientes preguntas de investigación, las cuales obtendrán respuesta al finalizar el proyecto.

> ***RQ1***: ¿Cómo es posible medir la cobertura de código cuando el testing es realizado a nivel de interfaz gráfica de usuario mediante la herramienta TESTAR?
>
> ***RQ2***: ¿Podemos servirnos de la cobertura de código para obtener conclusiones sobre la calidad de las pruebas realizadas mediante TESTAR?
>
> ***RQ3***: ¿Se obtienen diferencias significativas al variar entre los diferentes algoritmos de TESTAR, así como del uso del método getTopWidgets?
>
> ***RQ4***: ¿Cómo es posible dirigir el testing a nivel de GUI para aumentar los resultados de cobertura?

Para ser capaces de dar respuesta a las preguntas especificadas serán necesarios los requisitos que se listan a continuación:

- Determinar qué es lo que vamos a medir para poder dar respuesta a las preguntas.
- Elegir SUTs open source implementados en Java para testear con TESTAR. Vamos a elegir 3 diferentes aplicaciones.
- Configurar la herramienta TESTAR, la cual se encargará de realizar el testing sobre estas tres aplicaciones software.
- Ser capaces de medir la cobertura de código alcanzada por las pruebas ejecutadas mediante TESTAR sobre los diferentes SUTs.

La metodología empleada durante la elaboración del proyecto ha sido agrupada de la forma que sigue [20]:

- Diseño y definición del experimento.

    Sección 3.1 Variables a medir y set-up de los experimentos.

    Sección 3.2 Selección de los SUTs.

    Sección 3.3. Configuración de TESTAR para los SUTs.

    Sección 3.4. JaCoCo (Java Code Coverage).

    Sección 3.5. Integración de TESTAR, JaCoCo y SUTs.

    Sección 3.6. Comprobación de pruebas iniciales.

- Análisis e interpretación de los resultados obtenidos.

    Sección 4.1. Estadísticos descriptivos.

    Sección 4.2. Reducción del conjunto de datos.

    Sección 4.3. Contraste de hipótesis.

En las siguientes secciones procederemos a describir con mayor amplitud los requisitos expuestos. De igual manera se pondrá de manifiesto los pasos seguidos hasta la obtención final de las conclusiones.

3.1. Variables a medir y set-up de los experimentos

Con el objetivo de responder a las cuestiones anteriormente planteadas haremos uso de una serie de variables dependientes e independientes. Las variables independientes serán las que modifiquemos directamente a lo largo del estudio. Por su parte, las variables dependientes variarán en función de los valores especificados para las variables independientes. Entre las variables independientes a utilizar, y mediante las cuales obtendremos los resultados de las variables dependientes, tenemos:

- Cada uno de los diferentes algoritmos de selección de acciones presentes en la herramienta TESTAR (Random, Random+, Qlearning, Qlearning+ y MaxCoverage).
- El uso, o ausencia del mismo, del método getTopWidgets durante la ejecución de las pruebas.
- Los diferentes SUTs sobre los que realiza la tarea de testing.

Por otra parte, entre las variables dependientes destacaremos:

- Los valores alcanzados de cobertura en cada una de las pruebas realizadas.
- La media de cobertura alcanzada para cada algoritmo teniendo en cuenta si se utilizó el algoritmo anteriormente descrito getTopWidgets o si no se utilizó.
- Se cuantificará para cada SUT el número de pruebas cuyo valor de cobertura esté por encima y por debajo de la media.
- Finalmente presentaremos información relativa a la cobertura de rama alcanzada.

Cada ejecución de TESTAR supondrá la generación de una prueba a nivel de interfaz gráfica, con su correspondiente informe generado por JaCoCo donde se aprecie la cobertura código cubierta. Cada una de estas pruebas estará formada por una secuencia de 1000 acciones sin que exista límite máximo de tiempo. De esta manera tenemos:

*3 SUTS * 5 algoritmos * 2 maneras de determinar acciones = 30 ejecuciones con sus correspondientes informes JaCoCo*

Sin embargo, para evitar que los resultados obtenidos puedan darse debido al azar, y para proporcionar un mayor nivel de confianza a los mismos, cada uno de estas ejecuciones será repetida 10 veces. Somos conocedores de que el nivel de confianza es directamente proporcional al número de repeticiones realizadas, así como la cantidad de acciones que se efectúen en cada una de las secuencias a ejecutar. Sin embargo, debido al tiempo que conlleva la ejecución de las pruebas consideramos oportuno establecer a 10 el número de repeticiones a efectuar y a 1000 el número de acciones que deben realizarse en cada una de las pruebas. Por consiguiente, durante el proyecto se realizarán un total de 300 pruebas:

*3 SUTS * 5 algoritmos * 2 maneras de determinar acciones * 10 repeticiones = 300 ejecuciones con sus correspondientes informes JaCoCo*

3.2. Selección de los SUTs

Como hemos dicho anteriormente, en este trabajo se han utilizado 3 SUTS sobre los cuales TESTAR realizará sus diferentes pruebas, que serán posteriormente analizadas mediante JaCoCo. Por motivos de compatibilidad con esta última tecnología, los SUTS debían de ser open source y estar implementados en Java. Los distintos SUTS utilizados han sido: jEdit, FreeMind y Eclipse Mars.

3.2.1. jEdit

jEdit [21] (versión 5.4.0) es un editor de texto libre, distribuido bajo los términos de la Licencia pública general de GNU. Está escrito en Java y se ejecuta en Windows, GNU/Linux, Mac OS X y otros sistemas operativos que dispongan de la máquina virtual Java. En la Figura 3 podemos apreciar una captura de la interfaz gráfica de jEdit. Este software está compuesto por un total de 1.164 clases organizadas en 35 packages diferentes. A su vez consta de 7.852 métodos y 59.963 líneas de código. La Figura 2 ilustra uno de los posibles estados de la GUI de este software.

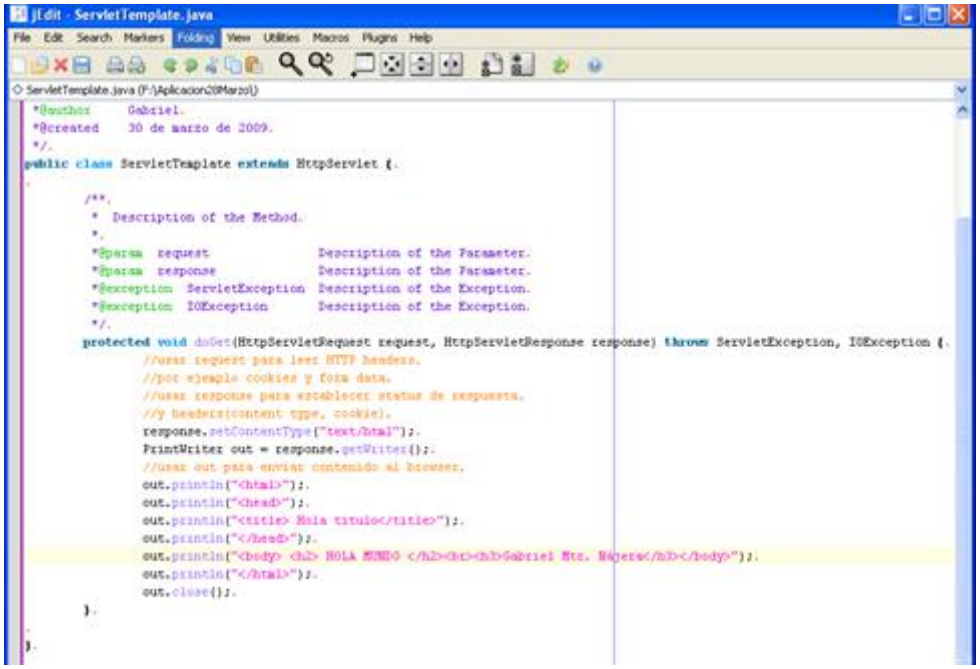


Figura 2: jEdit GUI

3.2.2. FreeMind

FreeMind [22] (versión 1.0.1) es una herramienta para la elaboración y manipulación de mapas conceptuales. Es decir, una herramienta para organizar y estructurar las ideas, los conceptos, su relación entre ellos y su evolución. Puede ser utilizada en cualquier área del ámbito educativo y como mecanismo o forma de plasmar tormentas

de ideas de todo tipo para su posterior reutilización. Es una herramienta de software libre programada en Java. Está bajo licencia GNU General Public License. Funciona en Microsoft Windows, Linux y Mac OS X vía Java Runtime Environment. Este software está compuesto por un total de 1.000 clases organizadas en 43 packages diferentes. A su vez consta de 6.434 métodos y 36.283 líneas de código. A continuación en la Figura 3 podemos observar uno de los estados habituales de la interfaz de esta aplicación.

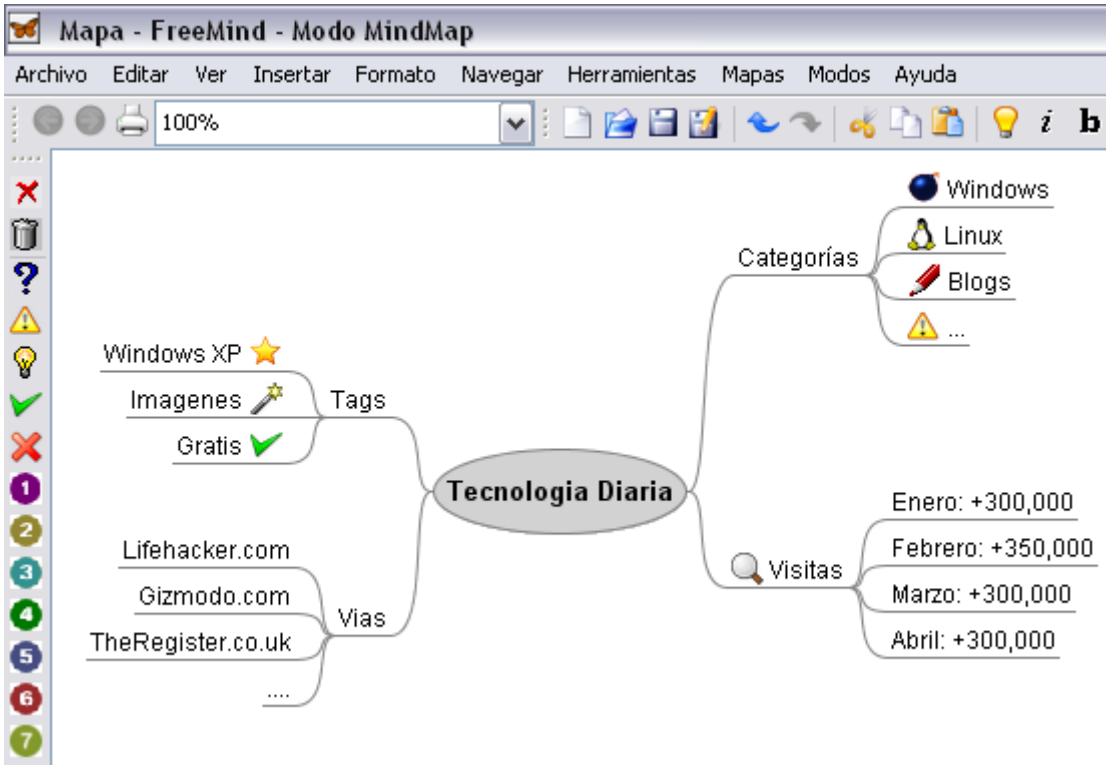


Figura 3: FreeMind GUI

3.2.3. Eclipse Mars.2

Eclipse [23] es uno de los entornos más conocidos y utilizados por los programadores, ya que se trata de un entorno de programación de código abierto y multiplataforma. Está soportado por una comunidad de usuarios lo que hace que tenga muchos plugins de modo que hacen que nos sirva para casi cualquier lenguaje, en este aspecto es de lo mejores. Sirve para Java, C++, PHP, Perl y un largo etcétera. También nos permite realizar aplicaciones de escritorio y aplicaciones web por lo que nos brinda una gran versatilidad. Este software está compuesto por un total de 865 clases organizadas en 40 packages diferentes. A su vez consta de 4.413 métodos y 23.076 líneas de código. En la Figura 4 podemos apreciar diferentes componentes de su interfaz.

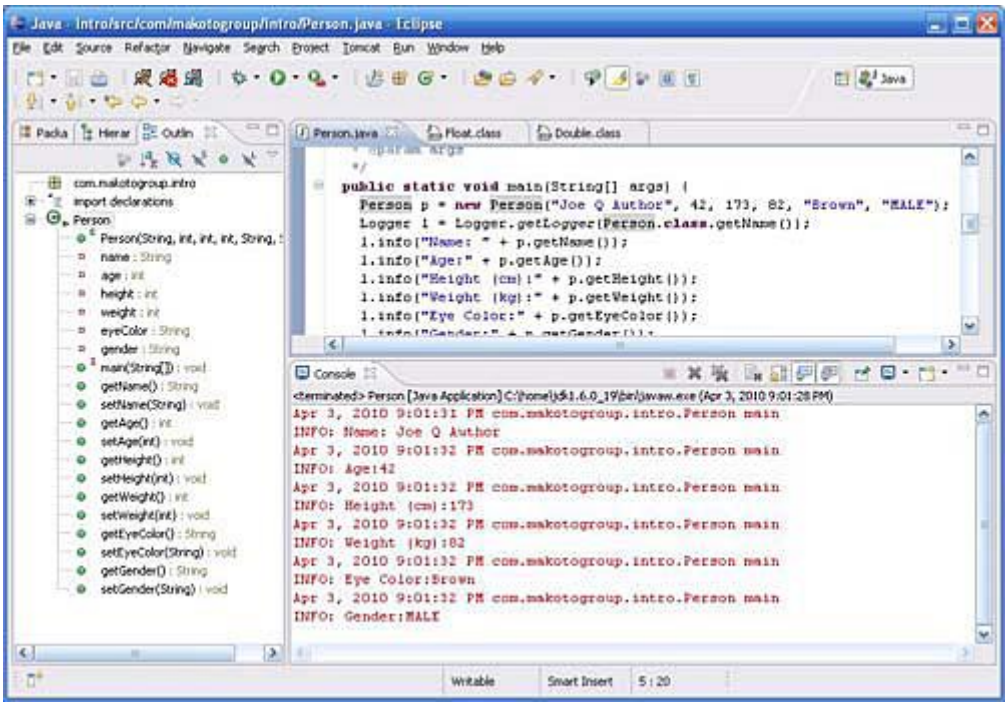


Figura 4: Eclipse GUI

3.3. Configuración de TESTAR para los SUTs

Previamente a la ejecución de las pruebas que se efectúen para resolver las incógnitas planteadas anteriormente es necesario realizar unas pruebas iniciales sobre los diferentes SUTs mediante TESTAR. Con tal fin se siguió el siguiente procedimiento:

- Ejecutar el SUT de manera manual.
- Establecer como SUTConnector SUT_WINDOW_TITLE.
- Ejecutar el modo Spy de TESTAR. Esto permite inspeccionar la interfaz del SUT, lo que nos proporciona información de utilidad para la preparación de las pruebas. Una vez que el SUT se está ejecutando y hemos accionado el modo Spy podremos obtener información como el título de los botones de la interfaz, lo cual nos servirá para evitar la ejecución sobre elementos no deseados de la interfaz. Para poder hacernos una mejor idea de lo que se acaba de explicar tenemos como ejemplo la Figura 5, en la cual se ha ejecutado el modo Spy sobre Eclipse y el cursor se encuentra localizado sobre el elemento 'Window' de la barra de herramientas de la aplicación.

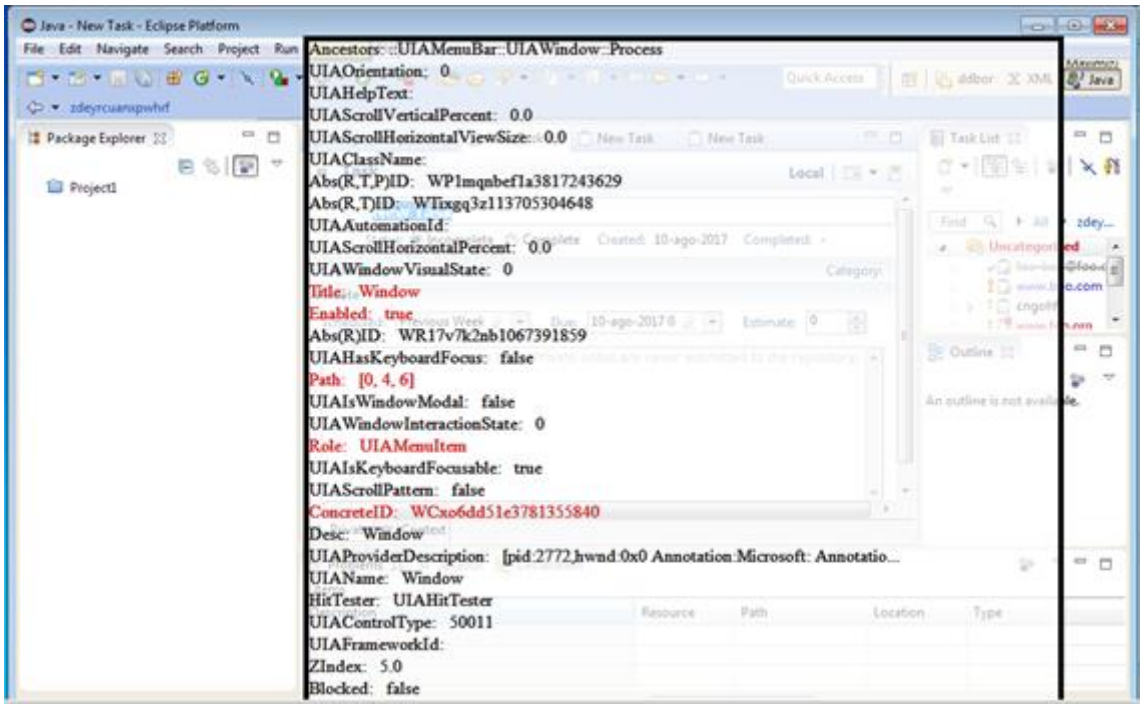


Figura 5: Modo Spy de TESTAR

Mediante este procedimiento se busca identificar qué elementos de la interfaz de la aplicación deseamos que TESTAR no ejecute para no obstaculizar la generación de tests. Por ejemplo se desea evitar los botones cerrar, minimizar, imprimir, etc. Una vez identificados estos elementos, y siendo conocedores de sus títulos identificativos, podemos añadirlos a la pestaña de filtros de TESTAR, asegurándonos de esta manera que no serán pulsados durante las pruebas. En las Figuras 6, 7 y 8 podemos apreciar la manera adecuada en la que se establecen los filtros en TESTAR para jEdit, FreeMind y Eclipse respectivamente. Una vez establecidos estos filtros podemos observar que a continuación es posible filtrar ciertos procesos que puedan tener lugar durante la generación de las pruebas. Para entender la finalidad de esto último debemos de tener en cuenta que durante la ejecución de las pruebas cabe la posibilidad de que se acceda a algún elemento del SUT que inicie una conexión con el navegador de internet que tengamos instalado, hecho que también deseamos evitar. Por si esto ocurre tenemos la posibilidad de matar los procesos relacionados con la ejecución de nuestro navegador.

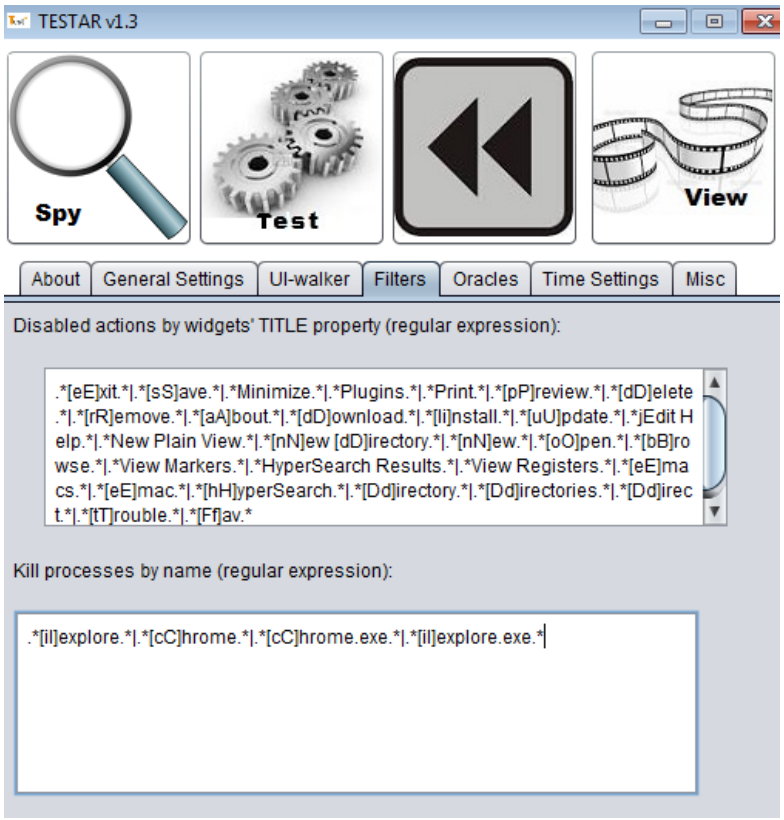


Figura 6: Filters de jEdit

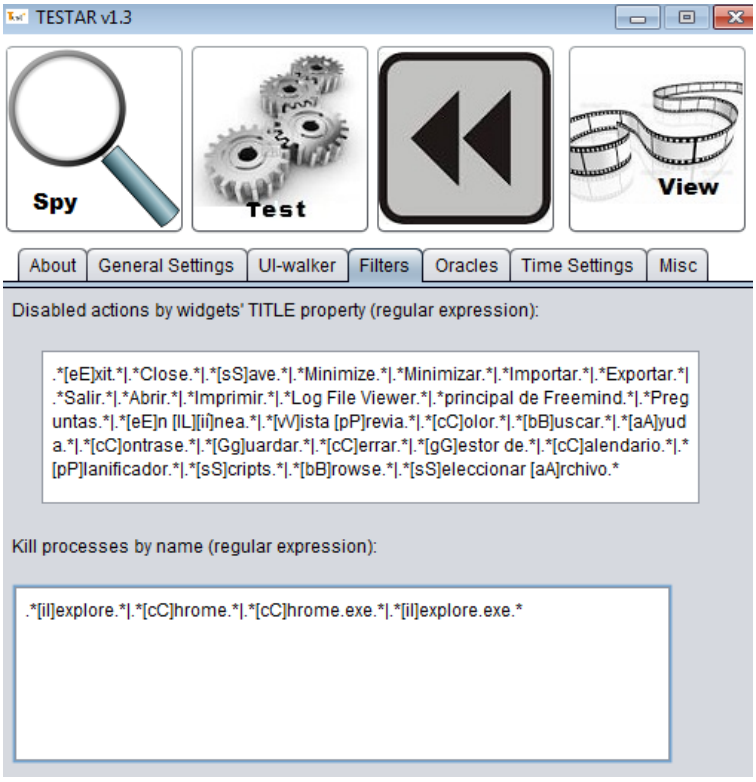


Figura 7: Filters de FreeMind

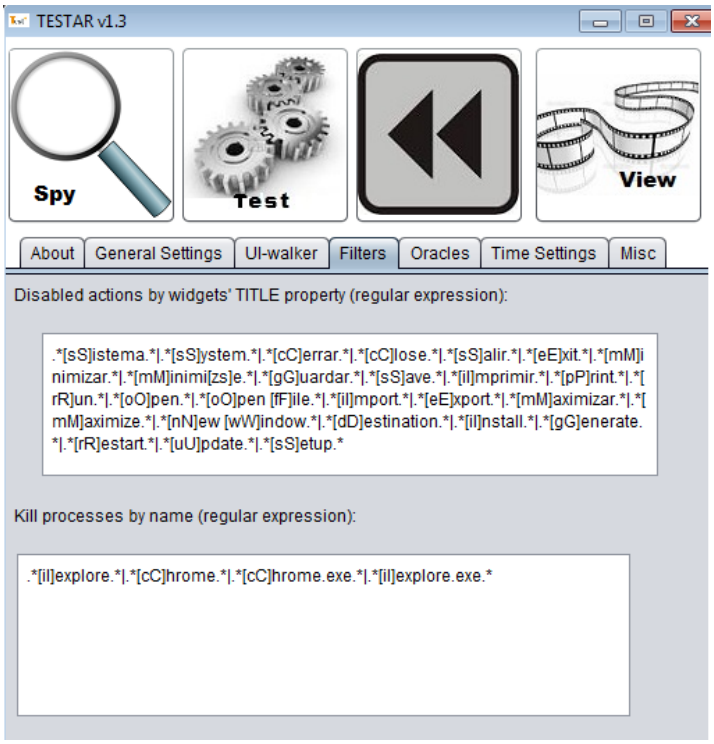


Figura 8:Filters de Eclipse

Por último, debemos establecer cuáles serán los oráculos. Los oráculos son expresiones regulares que nos alertan en el caso de que estén presentes en cualquiera de los títulos identificativos de los elementos de la interfaz. En la Figura 9 podemos ver los oráculos que se han establecido, los cuales son idénticos para los tres SUTs.

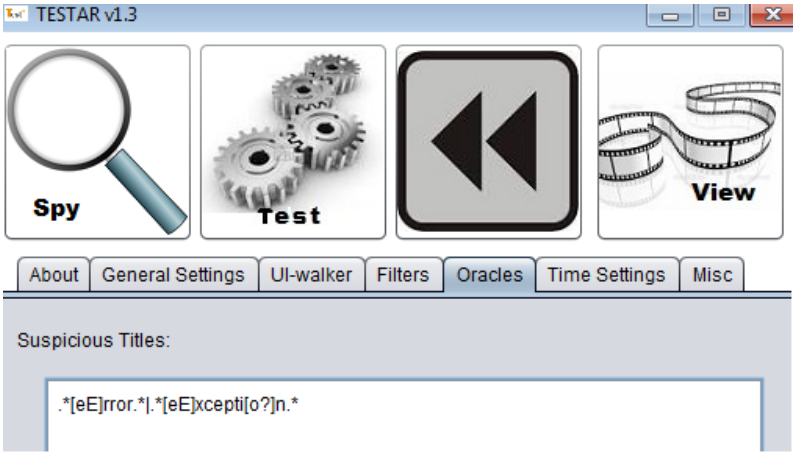


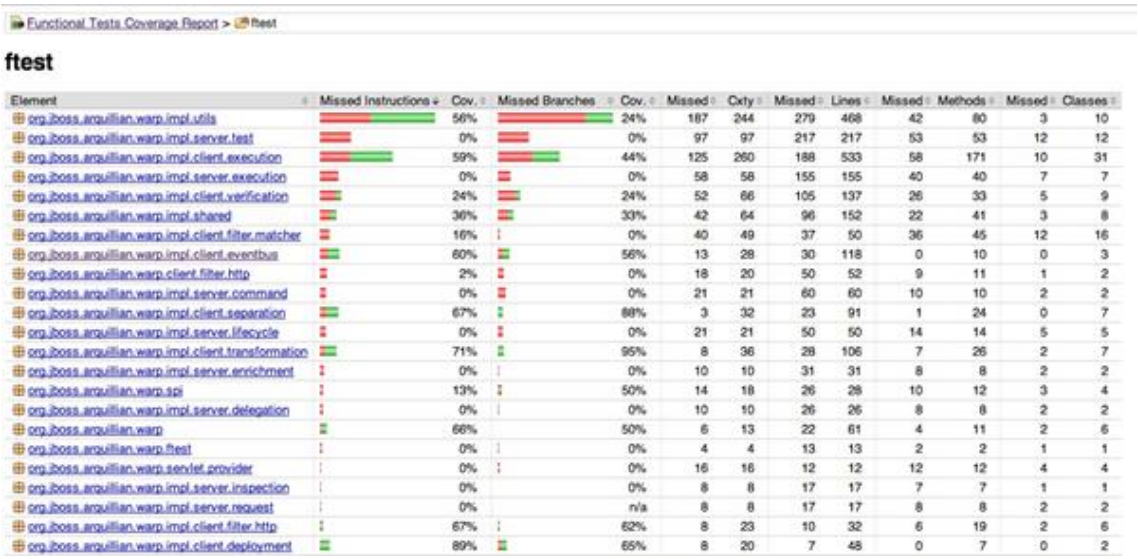
Figura 9: Oráculos de TESTAR

Una vez completado este paso previo se puede dar comienzo a la generación de tests iniciales con un número pequeño de acciones para comprobar el correcto funcionamiento de la herramienta.

3.4. JaCoCo (Java Code Coverage)

JaCoCo es una librería de código abierto que permite analizar la cobertura de código en entornos Java VM. Mediante esta herramienta podremos estudiar la cobertura obtenida en cada una de las pruebas realizadas por los diferentes algoritmos de generación de pruebas existentes en TESTAR. Frente a otras herramientas similares como Emma o Cobertura, JaCoCo sobresale por cuestiones como el mantenimiento bajo el que se encuentra (Emma no se actualiza desde 2005 y Cobertura desde 2010), lo que desemboca en otras cuestiones como la cantidad de tiempo necesario para realizar cálculos, etc [24]. Debido a estas cuestiones JaCoCo ha sido la herramienta empleada a lo largo de este estudio.

Una vez que se finaliza la ejecución de una prueba, y por tanto JaCoCo ya ha realizado el análisis de código cubierto en dicha prueba, se crea un informe en formato html donde se plasman los resultados obtenidos (véase Figura 10). JaCoCo utiliza diferentes contadores para calcular métricas de cobertura (como las que están descritas en la sección 2.4) por medio de información contenida en los archivos class del SUT analizado.


Functional Tests Coverage Report > ftest

ftest

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jboss.arquillian.warp.impl.utils		56%		24%	187	244	279	468	42	80	3	10
org.jboss.arquillian.warp.impl.server.test		0%		0%	97	97	217	217	53	53	12	12
org.jboss.arquillian.warp.impl.client.execution		59%		44%	125	260	188	533	58	171	10	31
org.jboss.arquillian.warp.impl.server.execution		0%		0%	58	58	155	155	40	40	7	7
org.jboss.arquillian.warp.impl.client.verification		24%		24%	52	66	105	137	26	33	5	9
org.jboss.arquillian.warp.impl.shared		36%		33%	42	64	96	152	22	41	3	8
org.jboss.arquillian.warp.impl.client.filter.matcher		16%		0%	40	49	37	50	36	45	12	16
org.jboss.arquillian.warp.impl.client.eventbus		60%		56%	13	28	30	118	0	10	0	3
org.jboss.arquillian.warp.client.filter.http		2%		0%	18	20	50	52	9	11	1	2
org.jboss.arquillian.warp.impl.server.command		0%		0%	21	21	60	60	10	10	2	2
org.jboss.arquillian.warp.impl.client.separation		67%		88%	3	32	23	91	1	24	0	7
org.jboss.arquillian.warp.impl.server.lifecycle		0%		0%	21	21	50	50	14	14	5	5
org.jboss.arquillian.warp.impl.client.transformation		71%		95%	8	36	28	106	7	26	2	7
org.jboss.arquillian.warp.impl.server.enrichment		0%		0%	10	10	31	31	8	8	2	2
org.jboss.arquillian.warp.spi		13%		50%	14	18	26	28	10	12	3	4
org.jboss.arquillian.warp.impl.server.delegation		0%		0%	10	10	26	26	8	8	2	2
org.jboss.arquillian.warp		66%		50%	6	13	22	61	4	11	2	6
org.jboss.arquillian.warp.ftest		0%		0%	4	4	13	13	2	2	1	1
org.jboss.arquillian.warp.servlet.provider		0%		0%	16	16	12	12	12	12	4	4
org.jboss.arquillian.warp.impl.server.inspection		0%		0%	8	8	17	17	7	7	1	1
org.jboss.arquillian.warp.impl.server.request		0%		n/a	8	8	17	17	8	8	2	2
org.jboss.arquillian.warp.impl.client.filter.http		67%		62%	8	23	10	32	6	19	2	6
org.jboss.arquillian.warp.impl.client.deployment		89%		65%	8	20	7	48	0	7	0	2

Figura 10: Informe JaCoCo

3.5. Apache Ant

Apache Ant [25] es una herramienta usada en programación para la realización de tareas mecánicas y repetitivas, normalmente durante la fase de compilación y construcción (build). Es, por tanto, un software para procesos de automatización de

compilación, desarrollado en lenguaje Java y requiere la plataforma Java, por lo que resulta apropiado para la construcción de proyectos Java.

Esta herramienta tiene la ventaja de no depender de las órdenes del shell de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas, siendo idónea como solución multi-plataforma. Ant utiliza XML para describir el proceso de generación y sus dependencias. Por defecto, el archivo XML se denomina build.xml.

Para utilizar ANT basta con disponer de una distribución binaria de ANT y tener instalado la versión 1.4 o superior del JDK. Para ejecutar ANT basta con escribir ant en la línea de comandos. Si se ha especificado la ejecución de un objetivo por defecto en la cabecera del proyecto del fichero build.xml no es necesario el uso de ningún parámetro ya que se ejecutara este por defecto.

En la siguiente sección podremos conocer el papel que desempeña esta herramienta.

3.6. Integración de TESTAR, JaCoCo y SUTs

Para que JaCoCo pueda generar un informe en el que aparezcan reflejados los datos relativos a la cobertura de código son necesarias una serie de cuestiones.

El primer paso es descargar un zip que contiene los archivos que necesitamos a través de la página oficial de JaCoCo [26]. Seguidamente debemos de localizar la ruta en la que se encuentra el archivo .jar que permita iniciar la ejecución de la aplicación que deseamos testear. Manteniendo al margen momentáneamente todo lo relativo a JaCoCo, cuando un usuario tenga localizado el .jar que ejecute la aplicación podrá ejecutar dicha aplicación mediante línea de comandos de la manera:

Java -jar C:\Users\usuario\Desktop\sut\sut.jar

Una vez aclarado esto volvemos a prestar a JaCoCo la atención que merece. El siguiente paso es extraer del fichero zip previamente descargado un archivo denominado jacocoagent.jar para situarlo en el directorio en el que se encuentre el hipotético sut.jar. Teniendo en cuenta que nuestro principal interés reside en recopilar información sobre la cobertura de una aplicación desde el momento que comience la ejecución de la misma, la ejecución del SUT en cuestión y la de JaCoCo debe ser simultánea. Para alcanzar este objetivo el comando anterior debe de verse modificado obteniéndose como resultado:

Java –javaagent:C:\Users\usuario\Desktop\sut\jacocoagent.jar –jar C:\Users\usuario\Desktop\sut\sut.jar

Mediante este comando se iniciará el SUT y se creará un archivo denominado jacoco.exec. Este archivo contendrá finalmente la información relativa a la cobertura una vez se cierre el SUT de manera adecuada. Con la especificación 'de manera adecuada' lo que se quiere decir es que si se interacciona sobre el elemento cerrar del SUT, este informe contendrá la información relativa a la cobertura alcanzada, mientras que si se cierra el SUT de otra manera, como matando su proceso correspondiente, o mediante un error en el propio SUT, no aparecerá ningún registro de cobertura en el archivo.

Hasta el momento hemos especificado la manera en la que obtener informes con JaCoCo sobre un SUT determinado dejando al margen la herramienta TESTAR. Recordemos que anteriormente se expuso la existencia diferentes procedimientos mediante los cuales TESTAR puede conectarse con un determinado SUT. Entre los procedimientos disponibles, el único que puede automatizarse es COMMAND_LINE, por lo que fue el que se utilizó. Teniendo en cuenta lo que se acaba de manifestar, así como los comandos necesarios detallados anteriormente, la Figura 11 ilustra la manera en la que finalmente podremos conectar simultáneamente TESTAR, un SUT determinado y JaCoCo.

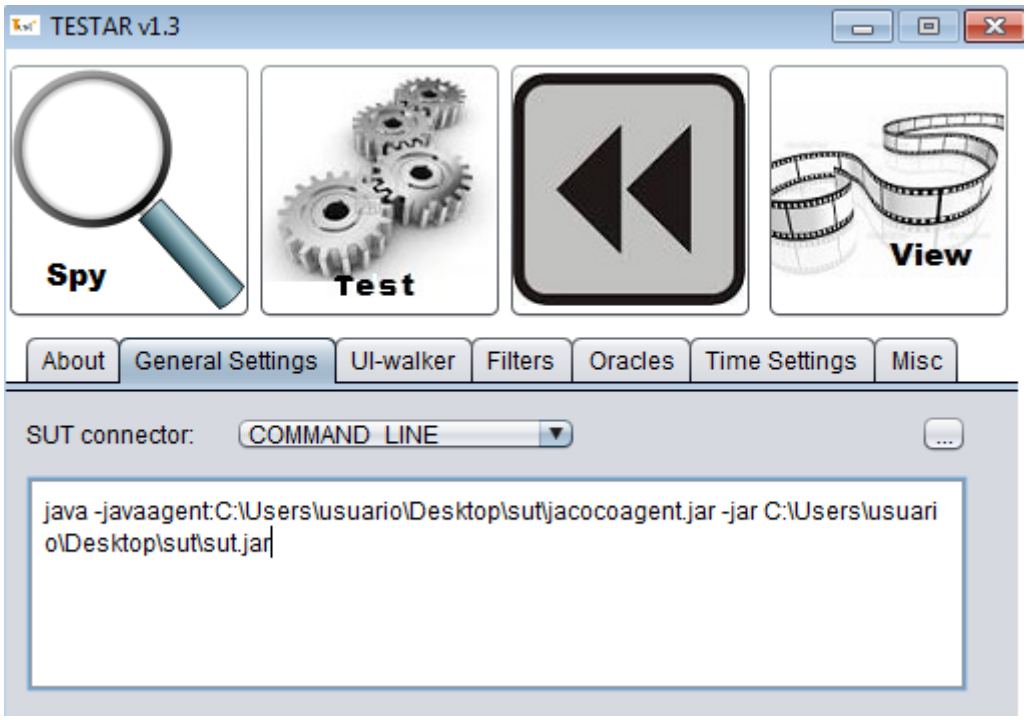


Figura 11: Conexión entre TESTAR, JaCoco y SUT

De esta forma al arrancar TESTAR, este ejecutaría el SUT determinado mientras JaCoCo recopila información acerca de la cobertura. En la carpeta raíz de TESTAR existe un script en batch denominado batchrun.bat el cual permitirá que la herramienta efectúe un número determinado de tests de manera automática tras su ejecución. Un ejemplo de ejecución, mediante línea de comandos y estando localizado en la carpeta en la que se encuentre dicho archivo, sería:

batchrun 10

En este caso las pruebas serían repetidas 10 veces. Dentro de este script se pueden especificar una serie de parámetros (véase figura 12). De especial interés son los siguientes:

- DGT: aquí se deberá especificar el algoritmo deseado de los presentes en TESTAR. Las opciones serán random, random+, qlearning, qlearning+ y maxcoverage.
- DSecuenceLenght: parámetro que refleja el número de acciones sobre la interfaz gráfica de usuario que se efectuarán por cada secuencia.

```
@echo off
if "%1"=="" GOTO MISS
GOTO GO
:MISS
echo Argument missing
echo Use: batchrun number_of_runs
:GO
@for /L %%i in (1,1,%1) do (
    @echo Starting RUN %%i - out of %1 -
    mkdir random\top\%%i
    @run.bat -Dheadless=true -DTG=random -DSequenceLength=1000 -DF2SL=true -DGRA=false -DTT=1 -DUT=true -DSST=0.95
    move "jacoco.exec" "random\top\%%i"
    @echo finished RUN %%i - out of %1 -
)
```

Figura 12: Estructura de batchrun.bat

Una vez finalizadas las pruebas iniciales y habiendo obtenido el archivo jacoco.exec será necesario un último paso para obtener la información presente en dicho archivo. Para este último paso es imprescindible disponer de la herramienta Apache Ant, así como añadir su carpeta 'bin' a la variable de entorno del sistema 'PATH'. Como no podía ser de otra manera, también es necesario disponer del archivo generado por jacoco, el denominado jacoco.exec. El tercer requisito necesario lo obtenemos al extraer del zip de JaCoCo obtenido previamente el archivo denominado jacocoant.jar y situarlo en el mismo directorio en el que se encuentre jacoco.exec. Este directorio no tiene porqué estar localizado en la misma carpeta en la que se encuentra el SUT, puede por ejemplo situarse en una nueva carpeta vacía destinada a albergar los informes de cobertura. El último requisito consiste en diseñar un archivo denominado build.xml cuya estructura aparece representada en la Figura 13. Este archivo es reutilizable para cualquier SUT, si bien es cierto que es necesario modificar los elementos etiquetados mediante <fileset> contenidos en <classfiles>. Dentro de estos elementos <fileset> debe de especificarse la ruta en la que se encuentran los archivos .class del SUT sobre los cuales se analizará la cobertura obtenida. En el momento en que dispongamos de los elementos expuestos, y estén todos situados en el mismo directorio, podemos ejecutar en dicho directorio el comando 'ant' mediante línea de comandos. El tiempo necesario para el proceso de generación de los informes variará en función de la cantidad de clases empleadas. Pasado dicho tiempo, se obtendrá como resultado una carpeta denominada 'reports' que contendrá un informe en

formato html sobre la cobertura alcanzada por el SUT. Puede verse un resumen esquematizado de todo este procedimiento en la Figura 14.

```
<?xml version="1.0"?>
<project name="JaCoCo Project" default="create-report">
    <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
        <classpath path="jacocoant.jar"/>
    </taskdef>
    <target name="create-report" description="Create a coverage report">
        <property name="jacoco.file" value="jacoco.exec"/>
        <property name="report.coverage.dir" value="reports"/>
        <jacoco:report xmlns:jacoco="antlib:org.jacoco.ant">
            <executiondata>
                <file file="${jacoco.file}" />
            </executiondata>
            <structure name="${ant.project.name}">
                <classfiles>
                    <fileset dir="C:\Users\usuario\Desktop\sut\classes"/>
                    <!-- all filesets here -->
                </classfiles>
            </structure>
            <html destdir="${report.coverage.dir}" />
        </jacoco:report>
    </target>
</project>
```

Figura 13: build.xml estructurado

En estudios posteriores a este proyecto en los que se desee nuevamente utilizar el software JaCoCo para medir la cobertura de código alcanzada en las pruebas generadas mediante TESTAR sobre una aplicación determinada basta con seguir las indicaciones descritas en este apartado. Cabe recalcar que, aunque en trabajos anteriores ya se ha utilizado TESTAR para realizar pruebas sobre diferentes aplicaciones, este es el primer proyecto que relaciona la herramienta utilizada con resultados referentes a cobertura de código. De esta forma aspiramos a motivar y facilitar el análisis de la cobertura de código alcanzada en investigaciones venideras.

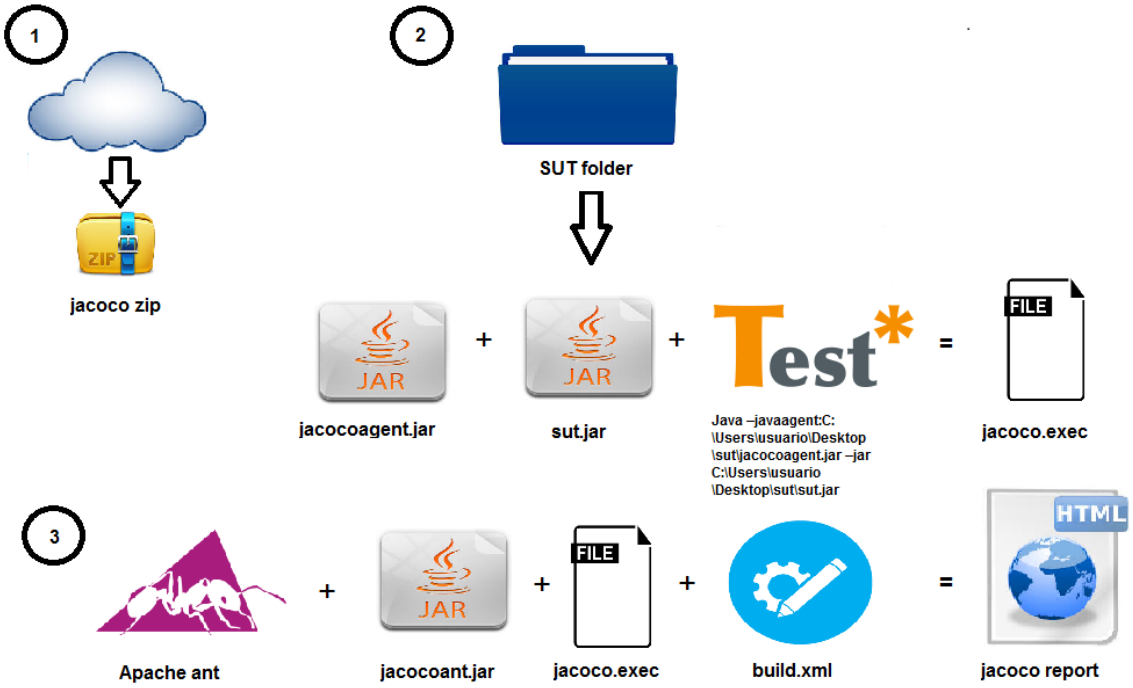


Figura 14: Esquema de pasos a seguir

3.7. Comprobación de pruebas iniciales

Una vez finalizada la ejecución de las primeras pruebas, se realizó una comprobación de los resultados obtenidos para asegurar que los informes se habían generado correctamente. Se vio que este no fue el caso, ya que en todos los informes generados hasta el momento aparecían correctamente todos los packages y clases que forman el SUT, pero no había ni el más mínimo registro de cobertura alcanzada en ellos. Es decir, en todos los casos la cobertura era del 0%. Este fue un hecho llamativo, ya que TESTAR había hecho sus pruebas correctamente sobre el SUT y aunque los informes se habían creado, no había ningún registro de cobertura en ellos. Evidentemente algo se había pasado por alto en algún momento del proceso de generación de tests. Posteriormente se averiguó dónde estaba el fallo. El problema residía en la forma en la que TESTAR cerraba el SUT cuando había terminado de generar el test correspondiente. Como hemos dicho anteriormente, este procedimiento se hizo de manera automática, de manera que le indicábamos a TESTAR el SUT, el algoritmo que debía emplear, el número de acciones que debía realizar cada test y el número de tests que deseábamos obtener. De esta manera TESTAR arranca el SUT, realiza un test, al finalizarlo cierra el SUT, lo vuelve a arrancar para iniciar el siguiente test, y así sucesivamente hasta que alcanza el número de pruebas indicadas. El inconveniente estaba en que TESTAR detiene la ejecución del SUT matando su proceso correspondiente. Esto hace que aunque el informe de JaCoCo se haya creado, no

almacene de manera correcta la información relativa a la cobertura, como se explicó en la sección anterior ‘puesta en marcha de JaCoCo’. La manera de tratar con este problema fue modificar el comportamiento interno de TESTAR. Lo que se hizo fue modificar uno de sus métodos de manera que en el momento en que la secuencia no tuviera más acciones que realizar, antes de cerrar el SUT se buscará por toda la interfaz un elemento cuyo nombre fuera ‘Cerrar’, ‘Salir’, ‘Exit’, etc. Y se ejecutará una acción sobre él. De esta forma se cerraba el SUT y se obtenía el informe JaCoCo correctamente.

4. Resultados

Una vez estudiado el proceso correspondiente a la generación de tests mediante la herramienta TESTAR y la posterior obtención de información relativa a la cobertura de código alcanzada es el momento de ejecutar y analizar los experimentos. Recordemos cuántas ejecuciones de TESTAR hay que efectuar con sus correspondientes algoritmos para cada SUT (véase Figura 15 y Figura 16):

JEdit	FreeMind
Random: con método getTopWidgets = 10 tests	Random: con método getTopWidgets = 10 tests
Random: sin método getTopWidgets = 10 tests	Random: sin método getTopWidgets = 10 tests
Random+: con método getTopWidgets = 10 tests	Random+: con método getTopWidgets = 10 tests
Random+: sin método getTopWidgets = 10 tests	Random+: sin método getTopWidgets = 10 tests
Qlearning: con método getTopWidgets = 10 tests	Qlearning: con método getTopWidgets = 10 tests
Qlearning: sin método getTopWidgets = 10 tests	Qlearning: sin método getTopWidgets = 10 tests
Qlearning+: con método getTopWidgets = 10 tests	Qlearning+: con método getTopWidgets = 10 tests
Qlearning+: sin método getTopWidgets = 10 tests	Qlearning+: sin método getTopWidgets = 10 tests
Maxcoverage: con método getTopWidgets = 10 tests	Maxcoverage: con método getTopWidgets = 10 tests
Maxcoverage: sin método getTopWidgets = 10 tests	Maxcoverage: sin método getTopWidgets = 10 tests
Total = 100 tests	Total = 100 tests

Figura 15: Estructura de las pruebas a realizar (I)

Eclipse
Random: con método getTopWidgets = 10 tests
Random: sin método getTopWidgets = 10 tests
Random+: con método getTopWidgets = 10 tests
Random+: sin método getTopWidgets = 10 tests
Qlearning: con método getTopWidgets = 10 tests
Qlearning: sin método getTopWidgets = 10 tests
Qlearning+: con método getTopWidgets = 10 tests
Qlearning+: sin método getTopWidgets = 10 tests
Maxcoverage: con método getTopWidgets = 10 tests
Maxcoverage: sin método getTopWidgets = 10 tests
Total = 100 tests

Figura 16: Estructura de las pruebas a realizar (II)

4.1. Estadísticos descriptivos

En primer lugar trataremos de obtener una visión general de los resultados obtenidos mediante estadísticos descriptivos [27]. Para ello nos serviremos del software R [28] el cual nos facilitará el procesado y análisis del conjunto de 300 tests. El conjunto de métodos y comandos utilizados para poder realizar este análisis y procesamiento de información está presente en el Anexo de este proyecto, facilitando así la posibilidad de trabajar con un gran volúmen de informes JaCoCo en investigaciones futuras.

4.1.1. jEdit

Para jEdit, al igual que para el resto de SUTs, tenemos un total de 100 tests; 20 por cada algoritmo de los cuales en 10 se ha utilizado getTopWidgets y en los restantes no.

Teniendo en cuenta esta clasificación obtenemos el primer resultado, un diagrama de cajas (véase Figura 17) donde aparece representada la cobertura de instrucción alcanzada por los tests efectuados.

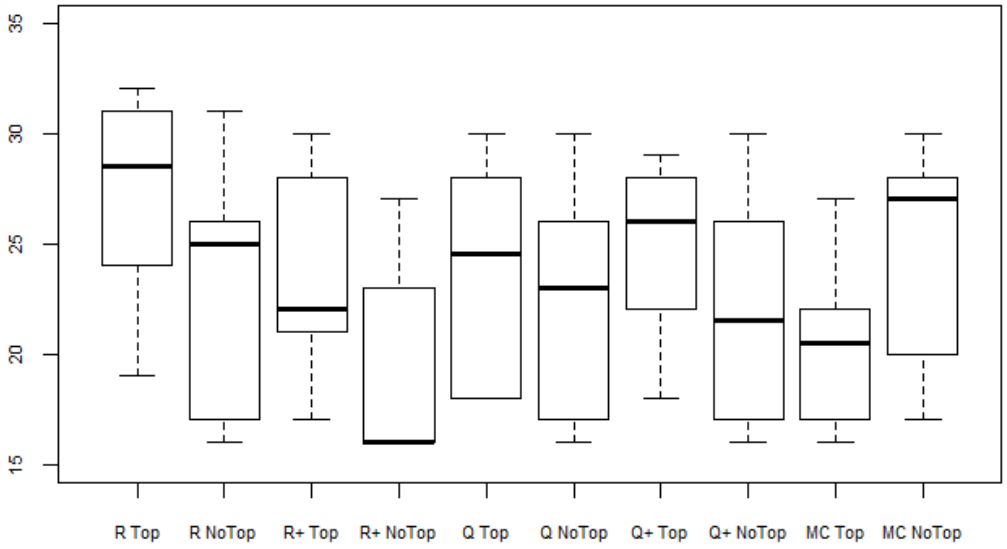


Figura 17: Diagrama cobertura de instrucción jEdit

Este diagrama representa, para cada conjunto de 10 tests, los valores mínimo y máximo alcanzados de cobertura para cada uno de ellos, los cuartiles en los que se encuentran divididos, y nos informa sobre posibles valores atípicos. Cada rectángulo o caja está dividido horizontalmente por el segundo cuartil o mediana, sobre el que se sitúa el segundo cuartil y bajo el cual está representado primer cuartil. Por su parte los bigotes o líneas que se extienden desde la caja representan los valores máximo y

mínimo de cobertura para cada conjunto de pruebas. Todos los valores se encuentran entre el rango de aproximadamente 15% y 33% de cobertura, sin existencia de valores erróneos (por ejemplo -3% o 120%), por lo que no existe motivo para descartar ningún dato.

Una vez realizada una primera visualización de los datos procedemos a dar algunos detalles más específicos de los mismos. En la Figura 18 podemos apreciar los valores medios de cobertura alcanzados por cada uno de los algoritmos empleados, diferenciando si se utilizó o no el método getTopWidgets.

	TopWidgets	Sin TopWidgets
Random	27.3	22.6
Random+	23.5	18.7
Qlearning	23.3	22.5
Qlearning+	25.0	21.6
MaxCoverage	20.3	24.9

Figura 18: Cobertura media jEdit

Si deseamos conocer la media de cobertura obtenida de forma general, es decir, sin diferenciar entre qué algoritmo se empleó y si se utilizó o no getTopWidgets, basta con hacer la media del conjunto de datos reflejados en la Figura 18. Esta media total para el caso de jEdit es de 22.97. Siendo conscientes de estos datos podemos realizar comprobaciones acerca del número de tests efectuados por cada algoritmo que está por encima o por debajo de los resultados globales de la media (véase Figura 19 y Figura 20).

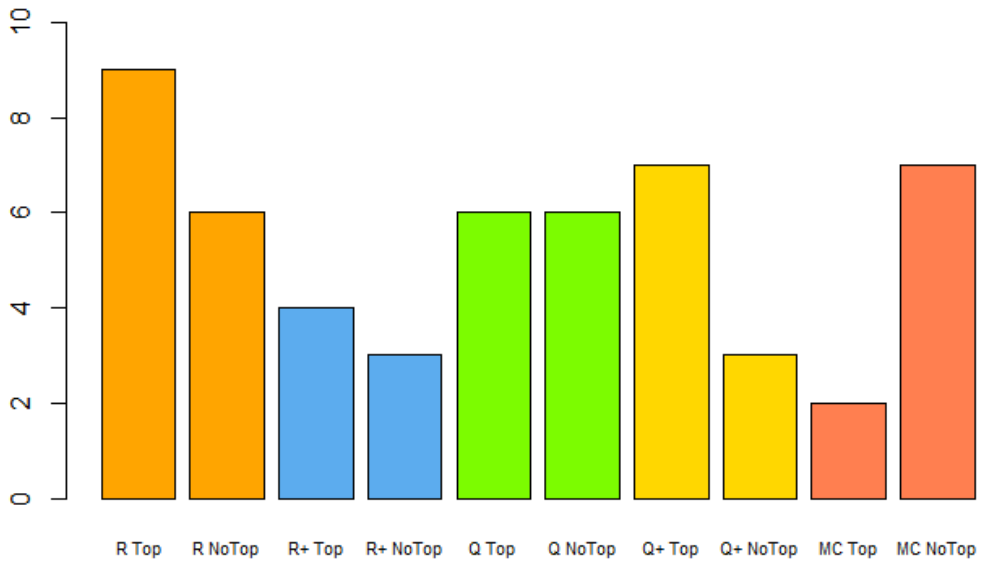


Figura 19: Tests con cobertura por encima de la media (jEdit)

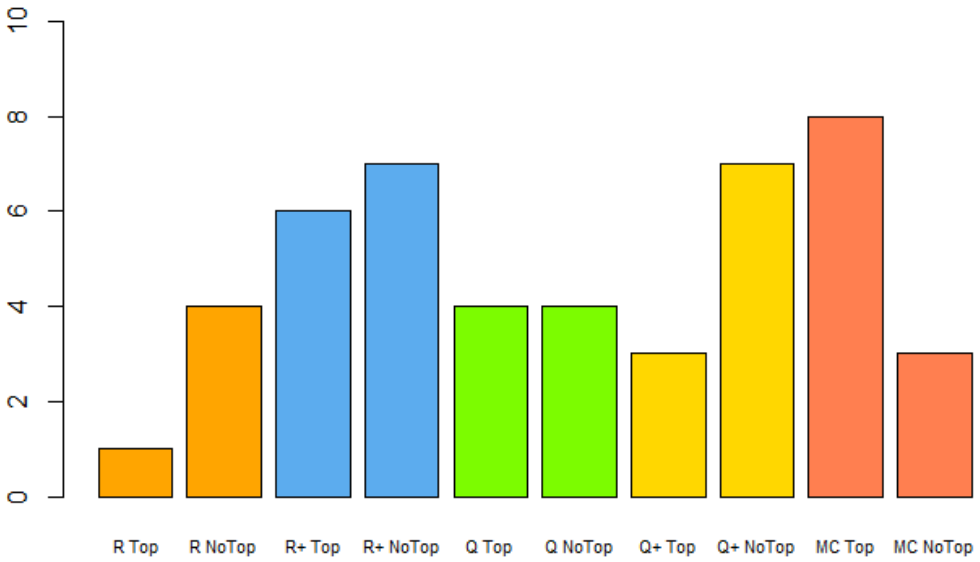


Figura 20: Tests con cobertura por debajo de la media (jEdit)

Como último elemento introductorio a estudiar tenemos los resultados referentes a branch coverage o cobertura de rama (véase Figura 21). Recordemos que este valor representa, sobre el total de ramificaciones existentes (diferentes ejecuciones que puedan realizarse en métodos switch, if, etc.) el porcentaje que ha sido cubierto.

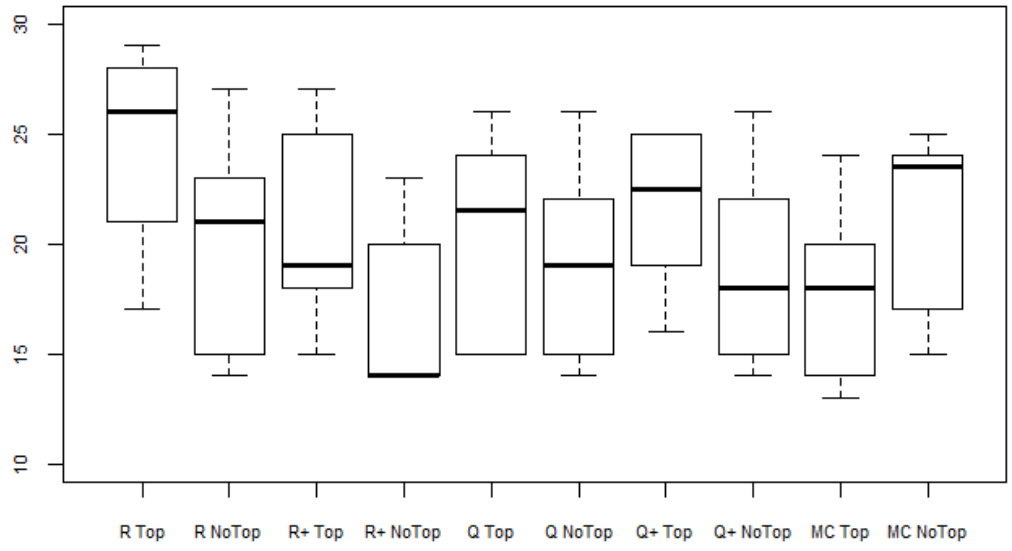


Figura 21: Cobertura de rama jEdit

Atendiendo a los datos presentados podemos apreciar que de manera general parece que utilizando el método getTopWidgets se logran unos mejores resultados que cuando no se utiliza. Esto no es en todos los casos, pues podemos ver que el algoritmo Maxcoverage no se beneficia del uso de dicho método. Las hipótesis formuladas y las pruebas realizadas en la sección posterior nos permitirán esclarecer estas dudas.

4.1.2. FreeMind

Para FreeMind, al igual que para el resto de SUTs, tenemos un total de 100 tests; 20 por cada algoritmo de los cuales en 10 se ha utilizado getTopWidgets y en los restantes no.

Teniendo en cuenta esta clasificación obtenemos el primer resultado, un diagrama de cajas donde aparece representada la cobertura de instrucción alcanzada por los tests efectuados (véase Figura 22).

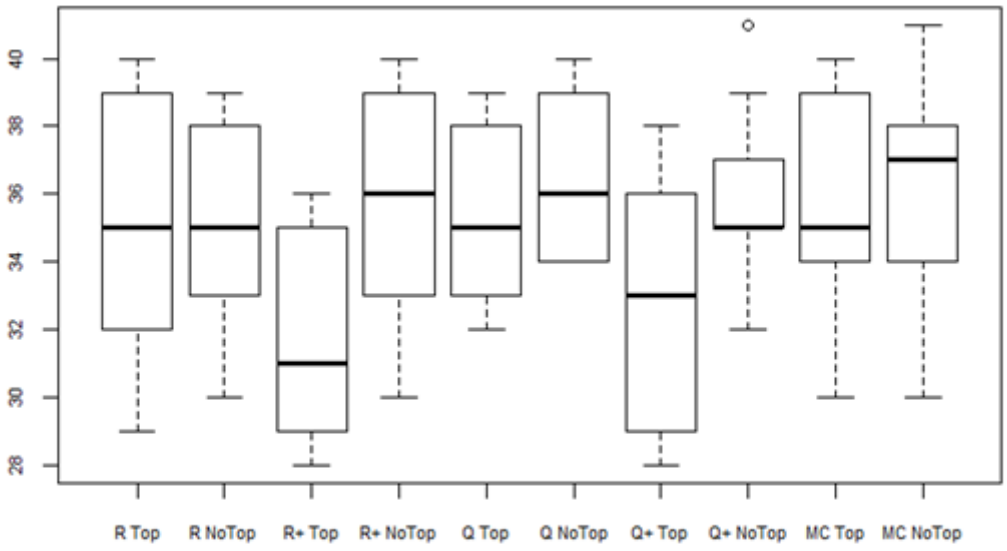


Figura 22: Diagrama cobertura de instrucción FreeMind

Como en el caso anterior, esta figura representa los valores de cobertura alcanzados para cada conjunto de pruebas realizadas con los diferentes algoritmos. En la Figura 23 podemos visualizar la cobertura media alcanzada por cada conjunto de pruebas con mayor exactitud.

	TopWidgets	Sin TopWidgets
Random	34.9	35.0
Random+	31.8	35.7
Qlearning	35.1	36.6
Qlearning+	32.8	35.9
MaxCoverage	35.4	35.9

Figura 23: Cobertura media FreeMind

Como para el SUT anterior, procedemos a hacer una comprobación acerca del número de pruebas cuyo valor de cobertura alcanzado es mayor o menor a la media global, alcanzando esta el valor de 34.91 (véase Figura 24 y Figura 25).

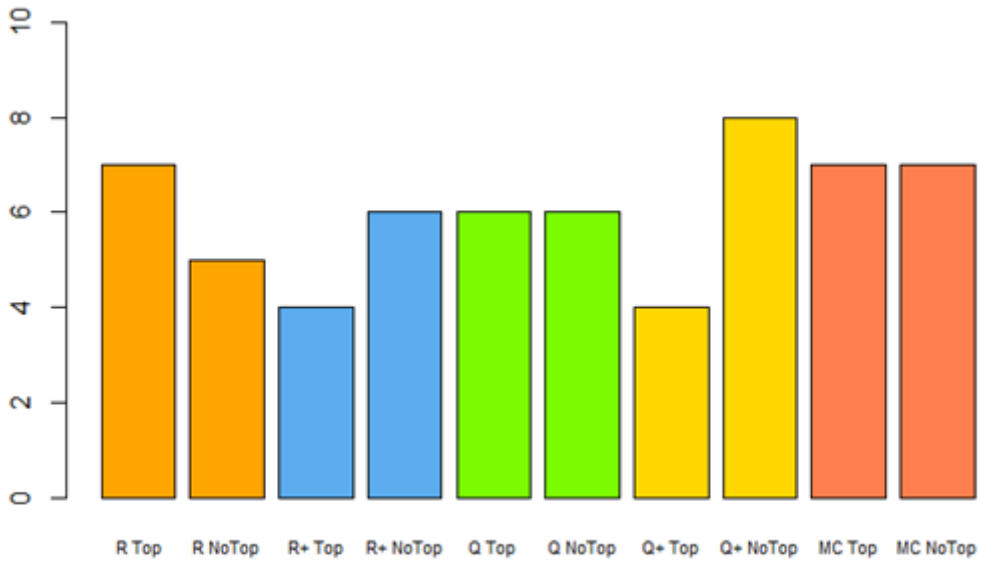


Figura 24: Tests con cobertura por encima de la media (FreeMind)

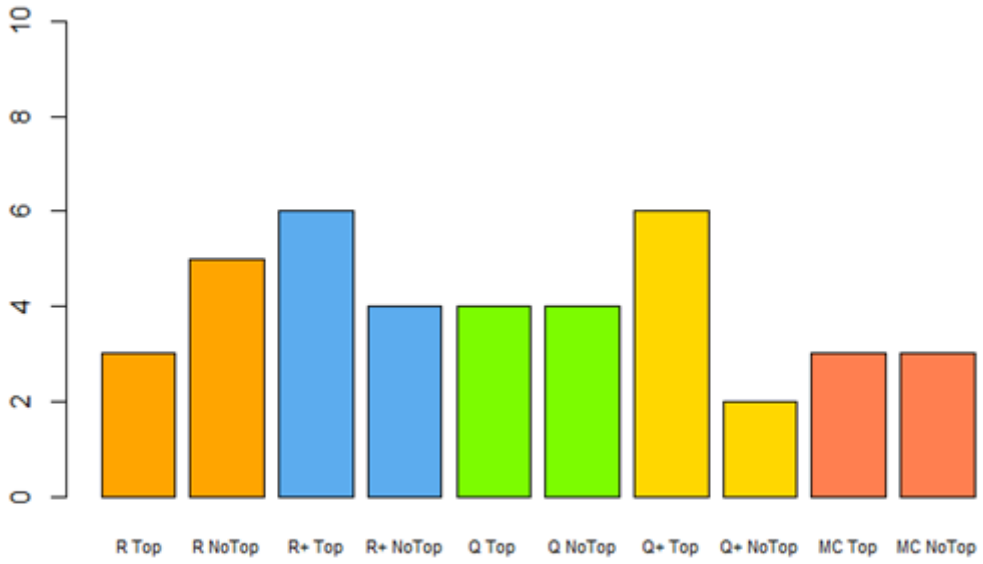


Figura 25: Tests con cobertura por debajo de la media (FreeMind)

Por último, en la Figura 26 podemos apreciar los resultados alcanzados referentes a la cobertura de rama alcanzada.

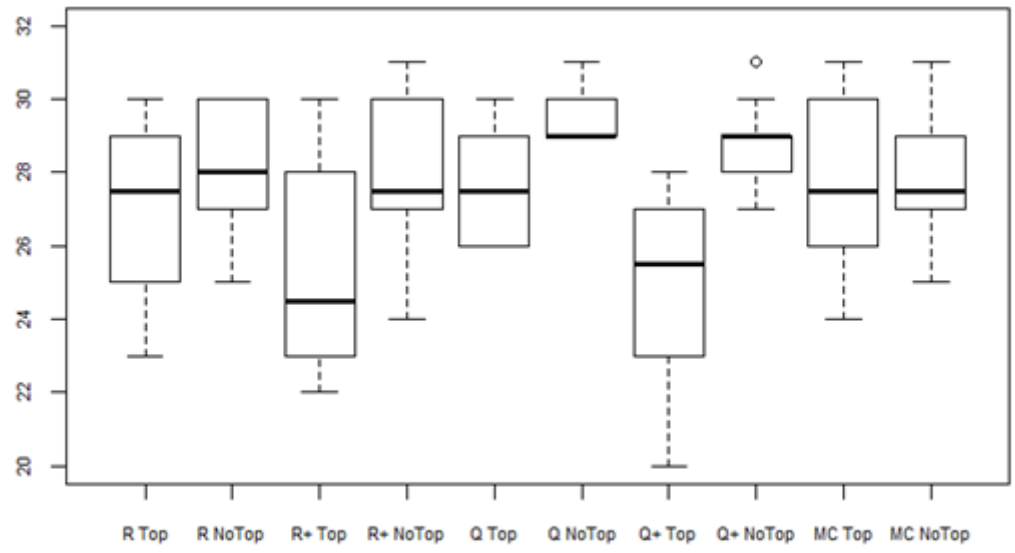


Figura 26: Cobertura de rama FreeMind

Repasando los datos obtenidos, puede parecer que para el caso de FreeMind existe, de manera generalizada, un aumento de la cobertura alcanzada cuando no se utiliza el método getTopWidgets. Posteriormente descubriremos si podemos realizar esta afirmación con rotundidad.

4.1.3. Eclipse

Para Eclipse, al igual que para el resto de SUTs, tenemos un total de 100 tests; 20 por cada algoritmo de los cuales en 10 se ha utilizado getTopWidgets y en los restantes no.

Teniendo en cuenta esta clasificación obtenemos el primer resultado, un diagrama de cajas donde aparece representada la cobertura de instrucción alcanzada por los tests efectuados (véase Figura 27).

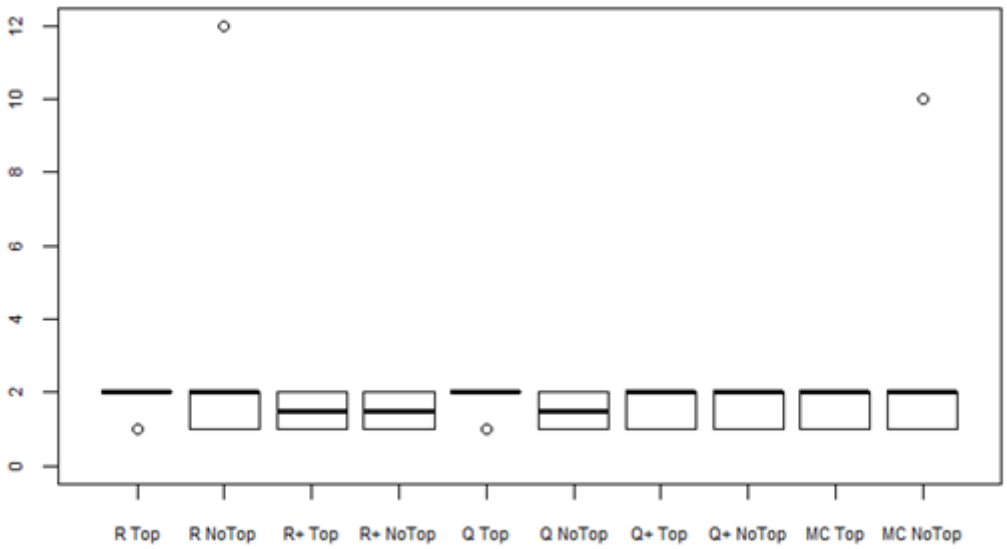


Figura 27: Diagrama cobertura de instrucción Eclipse

En el caso de Eclipse podemos observar que los valores de cobertura alcanzados son muy inferiores respecto a los otros SUTs estudiados, en los que en la mayoría de los casos la cobertura obtenida en cada prueba se sitúa en torno al 1% - 2% habiendo situaciones en las que se supera hasta llegar al 12%. En la Figura 28 podemos visualizar la cobertura media alcanzada por cada conjunto de pruebas con mayor exactitud.

	TopWidgets	Sin TopWidgets
Random	1.8	2.7
Random+	1.5	1.5
Qlearning	1.8	1.5
Qlearning+	1.7	1.6
MaxCoverage	1.7	2.4

Figura 28: Cobertura media Eclipse

La comprobación relativa al número de pruebas cuyo valor de cobertura alcanzado es mayor o menor a la media total (siendo este valor 1.82) podemos visualizarla en la Figura 29 y Figura 30.

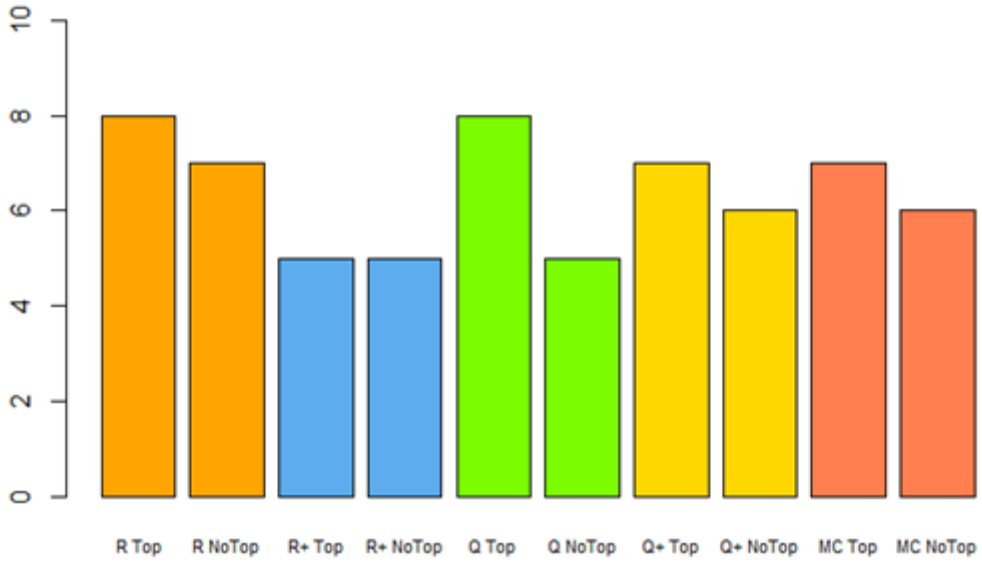


Figura 29: Tests con cobertura por encima de la media (Eclipse)

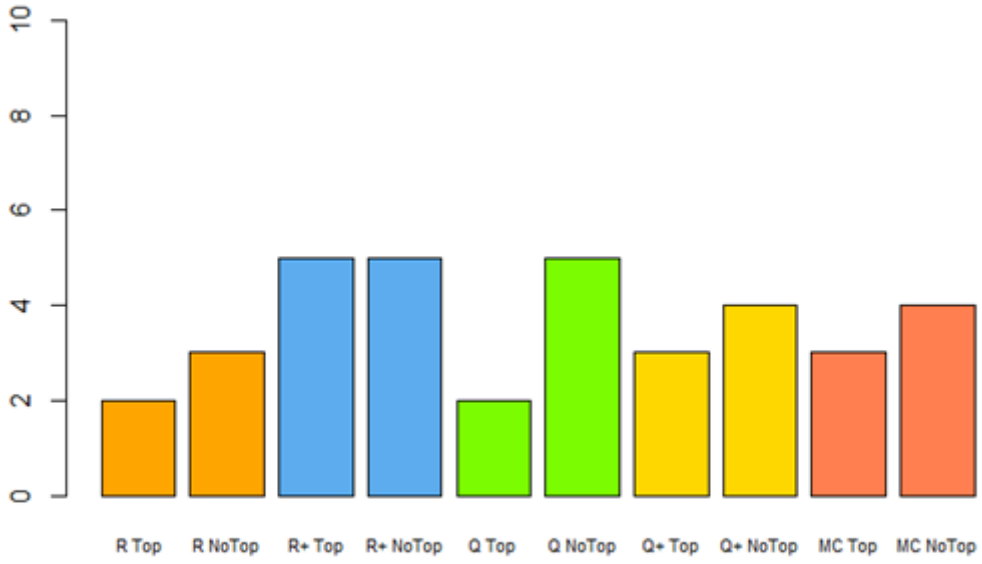


Figura 30: Tests con cobertura por debajo de la media (Eclipse)

Para finalizar con este apartado mostramos en la Figura 31 los resultados alcanzados referentes a la cobertura de rama.

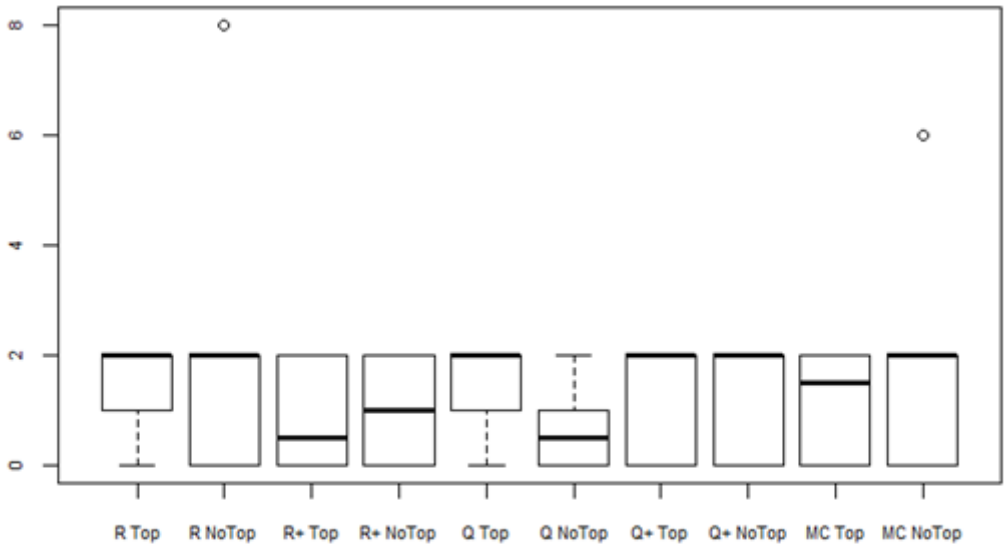


Figura 31: Cobertura de rama Eclipse

Una vez presentada la información referente a los estadísticos descriptivos de los 3 SUTs procedemos a dar comienzo al siguiente apartado: reducción del conjunto de datos.

4.2. Reducción del conjunto de datos

Los únicos datos que se descartaron fueron los asociados a los experimentos fallidos expuestos en la sección '3.6. Comprobación de pruebas iniciales' debido al hecho de que TESTAR finalizaba la ejecución de las pruebas matando el proceso correspondiente al SUT en cuestión. Una vez arreglado este problema todos los demás datos fueron considerados de interés.

4.3. Formulación de hipótesis

En esta sección del proyecto podremos determinar la relación de efectividad para lograr cubrir una mayor cantidad de código entre los diferentes algoritmos. Para lograr este objetivo se formulará una serie de hipótesis. La formulación de hipótesis seguirá la siguiente estructura:

- Habrá una hipótesis H0 que afirmará que es indiferente utilizar un algoritmo u otro ya que todos producen un resultado similar.
- Hipótesis H1, la cual negará la anterior afirmación.

Una vez formuladas se procederá a comprobar cuál de las dos es la veraz. Para esto se realizará la prueba U de Mann-Whitney o Mann-Whitney-Wilcoxon Test [29] sobre los resultados obtenidos con cada algoritmo.

Para cada uno de los SUTs utilizados se realizará una primera hipótesis que tratará de determinar si es indiferente o no utilizar el método getTopWidgets. Seguidamente se procederá a realizar las comparaciones pertinentes entre los resultados de los cinco algoritmos, teniendo en cuenta si se sirvieron del método getTopWidgets o no. El contraste de hipótesis entre estos últimos resultados se realizará de la siguiente manera:

- En primer lugar se realizará la prueba U de Mann-Whitney o Mann-Whitney-Wilcoxon Test sobre el conjunto de datos obtenidos en los que se utilizó el método getTopWidgets. De esta forma habrá que realizar el estudio de cada uno de los algoritmos con los otros cuatro algoritmos restantes.
- En segundo lugar realizaremos lo anteriormente descrito sobre el conjunto de datos en los que no se utilizó el método getTopWidgets.
- Finalmente se efectuará un estudio de la cobertura alcanzada en cada algoritmo, tratando de determinar si para dicho algoritmo es más efectivo utilizar el método getTopWidgets o no.

4.3.1. jEdit

Procedemos a formular las diferentes hipótesis sobre los resultados obtenidos con el SUT jEdit:

1. Hipótesis sobre la eficacia de utilizar getTopWidgets.

- H0: es indiferente utilizar getTopWidgets o no sobre los diferentes algoritmos.
- H1: no es indiferente utilizar getTopWidgets o no sobre los diferentes algoritmos.

Para aceptar o rechazar la hipótesis H0 realizamos la prueba Mann-Whitney-Wilcoxon Test sobre dos conjuntos de valores (véase Figura 32). El primero será el conjunto de valores de cobertura alcanzada utilizando getTopWidgets y el segundo el conjunto de valores de cobertura alcanzada sin utilizar dicho método. Para esta prueba ejecutamos la función wilcox.text en el software R con el objetivo de determinar cuál es la hipótesis correcta correcta. Si el resultado de p-value es inferior a 0.05 se rechaza la hipótesis nula, H0, y se acepta la hipótesis H1.

```
jct <- c(cv_rt,cv_rmt,cv_qt,cv_qmt,cv_mt)
jcn <- c(cv_rn,cv_rmn,cv_qn,cv_qmn,cv_mn)
#diferencia usar getTopWidgets y no usarlo
wilcox.test(jct,jcn)
```

```
        Wilcoxon rank sum test with continuity correction

data:  jct and jcn
W = 1541.5, p-value = 0.04408
alternative hypothesis: true location shift is not equal to 0
```

Figura 32: Resultado Mann-Whitney-Wilcoxon

Como resultado de este valor p-value obtenemos 0.04408, con lo que rechazamos H0 y aceptamos H1. Con este resultado verificamos que no es indiferente utilizar getTopWidgets o no utilizarlo para este SUT. Sin embargo esta prueba no nos dice cuál de las opciones es más efectiva. Para ello simplemente realizamos una comprobación como hacer el sumatorio, o la media entre los resultados obtenidos de las dos maneras a estudiar. Al realizar este cálculo podemos concluir que para los experimentos realizados sobre jEdit es más efectivo utilizar getTopWidgets, si bien es cierto que, como hemos visto en la sección anterior, existen métodos cuya efectividad se ve mermada con el uso de dicho método.

2. Hipótesis comparativa entre los cinco algoritmos utilizando getTopWidgets.

Con el fin de evitar ocupar una gran cantidad de espacio en la formulación de las 10 hipótesis que permitan establecer relaciones de efectividad entre los diferentes algoritmos al utilizar getTopWidgets, a continuación se ilustra una tabla (véase Figura 33) con los resultados de p-value obtenidos en la prueba Mann-Whitney-Wilcoxon. La formulación de la hipótesis resultante adquiere la estructura:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X	Cob. media X	Algoritmo Y	Cob. media Y	P-value	Rechazar H0
Random		Random+		0.0687	No
Random		Qlearning		0.06193	No
Random		Qlearning+		0.1713	No
Random	27.3	Maxcoverage	20.3	0.00451	Sí
Random+		Qlearning		0.939	No
Random+		Qlearning+		0.5435	No
Random+		Maxcoverage		0.09367	No
Qlearning		Qlearning+		0.4215	No
Qlearning		Maxcoverage		0.1284	No
Qlearning+	25.0	Maxcoverage	20.3	0.01875	Sí

Figura 33: Prueba Mann-Whitney-Wilcoxon para jEdit getTopWidgets

Formulando la hipótesis anteriormente planteada y sustituyendo las variables Algoritmo X y Algoritmo Y por las que se ven reflejadas en la Figura 33, podemos concluir que para el SUT jEdit, utilizando getTopWidgets, existe una diferencia significativa entre:

- Utilizar Random o Maxcoverage.
- Utilizar Qlearning+ o Maxcoverage.

Queda pendiente conocer qué algoritmo proporciona un mayor porcentaje de cobertura. Para ello hemos añadido en la ilustración anterior la cobertura media alcanzada en los casos en los que podemos rechazar la hipótesis H0. Volviendo a consultar dicha ilustración podemos concluir que para jEdit utilizando getTopWidgets son más efectivos los algoritmos Random y Qlearning+ frente al algoritmo Maxcoverage.

3. Hipótesis comparativa entre los cinco algoritmos sin utilizar getTopWidgets.

El procedimiento a seguir en este apartado es similar al del caso anterior. Por consiguiente la formulación de las hipótesis adquiere la siguiente estructura:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X	Cob. media X	Algoritmo Y	Cob. media Y	P-value	Rechazar H0
Random	22.6	Random+	18.7	0.04691	Sí
Random		Qlearning		0.8789	No
Random		Qlearning+		0.8484	No
Random		Maxcoverage		0.1578	No
Random+	18.7	Qlearning	22.5	0.0459	Sí
Random+		Qlearning+		0.06036	No
Random+	18.7	Maxcoverage	24.9	0.002877	Sí
Qlearning		Qlearning+		0.5928	No
Qlearning		Maxcoverage		0.1954	No
Qlearning+		Maxcoverage		0.1799	No

Figura 34: Prueba Mann-Whitney-Wilcoxon para jEdit sin getTopWidgets

Formulando las hipótesis anteriormente planteada y sustituyendo las variables Algoritmo X y Algoritmo Y por las que se ven reflejadas en la Figura 34, podemos concluir que para el SUT jEdit, sin utilizar getTopWidgets, existe una diferencia significativa entre:

- Utilizar Random+ o Random.
- Utilizar Random+ o Qlearning.
- Utilizar Random+ o Maxcoverage.

Falta por saber qué algoritmo proporciona un mayor porcentaje de cobertura. Para ello consultamos nuevamente la información relativa a la cobertura media alcanzada para los casos en los que se rechaza la hipótesis H0. Así podemos concluir que para jEdit al no utilizar getTopWidgets son más efectivos los algoritmos Random, Qlearning y Maxcoverage frente al algoritmo Random+.

4. Hipótesis sobre la efectividad del método getTopWidgets en cada uno de los algoritmos.

En la primera hipótesis planteada para este SUT hemos comprobado que es más ventajoso utilizar el método getTopWidgets. Debe de tenerse en cuenta que este beneficio es generalizado para todos los algoritmos, y debemos recordar que no todos alcanzaban una mayor cobertura con el uso de este método. En este apartado trataremos de determinar, para cada algoritmo, si existe una diferencia significativa entre el uso de getTopWidgets y la ausencia del mismo. Las hipótesis planteadas son:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X (con getTopWidgets)	Cob. media X	Algoritmo Y (sin getTopWidgets)	Cob. media Y	P-value	Rechazar H0
Random		Random		0.08753	No
Random+	23.5	Random+	18.7	0.01835	Sí
Qlearning		Qlearning		0.4469	No
Qlearning+		Qlearning+		0.1194	No
Maxcoverage	20.3	Maxcoverage	24.9	0.03966	Sí

Figura 35: Prueba Mann-Whitney-Wilcoxon para jEdit con y sin getTopWidgets

Al formular las hipótesis, sustituir las variables Algoritmo X y Algoritmo Y por las que se ven reflejadas en la Figura 35 y realizar la prueba Mann-Whitney-Wilcoxon, podemos concluir que existen diferencias significativas entre utilizar getTopWidgets o no para los algoritmos Random+ y Maxcoverage. Como en los apartados anteriores, para saber en qué caso se obtiene una mejor cobertura basta con consultar la media de los diferentes conjuntos de datos. Así, podemos determinar que Random+ obtiene mejores resultados si se emplea el método getTopWidgets mientras que Maxcoverage se beneficia de la ausencia de dicho método.

4.3.2. FreeMind

Procedemos a formular las diferentes hipótesis sobre los resultados obtenidos con el SUT FreeMind:

1. Hipótesis sobre la eficacia de utilizar getTopWidgets.

- H0: es indiferente utilizar getTopWidgets o no sobre los diferentes algoritmos.
- H1: no es indiferente utilizar getTopWidgets o no sobre los diferentes algoritmos.

Para aceptar o rechazar la hipótesis H0 realizamos la prueba Mann-Whitney-Wilcoxon Test sobre dos conjuntos de valores como hicimos con el SUT anterior. El primero será el conjunto de valores de cobertura alcanzada utilizando getTopWidgets y el segundo el conjunto de valores de cobertura alcanzada sin utilizar dicho método. Para este tipo de prueba ejecutamos la función wilcox.text para decidir qué hipótesis es la correcta. Recordemos que si el resultado de p-value es inferior a 0.05 se rechaza la hipótesis nula, H0, y se acepta la hipótesis H1.

Como resultado de este valor p-value obtenemos 0. 01907, con lo que rechazamos H0 y aceptamos H1. Con este resultado verificamos que no es indiferente utilizar getTopWidgets o no utilizarlo para el SUT FreeMind, sin embargo esta prueba no nos dice cuál de las opciones es más efectiva. Para ello simplemente hacemos una

comprobación como por ejemplo hacer un sumatorio, o la media entre los resultados obtenidos de las dos maneras a estudiar. Con dicho cálculo concluimos que se alcanza un mejor resultado al no utilizar getTopWidgets. También podemos ver que, así como de manera general es más efectivo usarlo, de manera específica hay ciertos algoritmos que mejoran su resultado sin utilizar getTopWidgets.

2. Hipótesis comparativa entre los cinco algoritmos utilizando getTopWidgets.

Como en el caso anterior, procedemos a ilustrar en la Figura 36 los resultados de p-value obtenidos en la prueba Mann-Whitney-Wilcoxon. Así se podrá justificar la existencia de relaciones de efectividad entre los diferentes algoritmos utilizando getTopWidgets, o ausencia de las mismas. Las hipótesis formuladas son:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X	Cob. media X	Algoritmo Y	Cob. media Y	P-value	Rechazar H0
Random		Random+		0.1124	No
Random		Qlearning		0.9074	No
Random		Qlearning+		0.3219	No
Random		Maxcoverage		0.8463	No
Random+	31.8	Qlearning	35.1	0.0458	Sí
Random+		Qlearning+		0.4678	No
Random+		Maxcoverage		0.05481	No
Qlearning		Qlearning+		0.2233	No
Qlearning		Maxcoverage		0.6445	No
Qlearning+		Maxcoverage		0.1717	No

Figura 36: Prueba Mann-Whitney-Wilcoxon para FreeMind getTopWidgets

Modificando las variables Algoritmo X y Algoritmo Y por las que aparecen representadas en la Figura 36, podemos concluir que para FreeMind no es indiferente utilizar Random+ o Qlearning cuando se emplea getTopWidgets. Consultando el valor de los valores de cobertura alcanzados para estos dos algoritmos podemos afirmar que para este SUT es más beneficioso utilizar Qlearning que Random+.

3. Hipótesis comparativa entre los cinco algoritmos sin utilizar getTopWidgets.

El procedimiento a seguir en este apartado es similar al del caso anterior. La formulación de las hipótesis adquiere la siguiente estructura:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X	Algoritmo Y	P-value	Rechazar H0
Random	Random+	0.6743	No
Random	Qlearning	0.1571	No
Random	Qlearning+	0.5677	No
Random	Maxcoverage	0.5703	No
Random+	Qlearning	0.5162	No
Random+	Qlearning+	0.9392	No
Random+	Maxcoverage	1	No
Qlearning	Qlearning+	0.8777	No
Qlearning	Maxcoverage	0.7304	No
Qlearning+	Maxcoverage	0.7887	No

Figura 37: Prueba Mann-Whitney-Wilcoxon para FreeMind sin getTopWidgets

Formulando las hipótesis anteriormente planteadas y sustituyendo las variables Algoritmo X y Algoritmo Y por las que se ven reflejadas en la Figura 37, podemos concluir que en FreeMind cuando no se utiliza getTopWidgets no existe una diferencia lo suficientemente destacable entre los resultados producidos por los diferentes algoritmos de TESTAR.

4. Hipótesis sobre la efectividad del método getTopWidgets en cada uno de los algoritmos.

En la primera hipótesis planteada para este SUT hemos comprobado que es más ventajoso no utilizar el método getTopWidgets. Debe de tenerse en cuenta que este beneficio es generalizado para todos los algoritmos, y debemos recordar que no todos alcanzaban una mayor cobertura sin el uso de este método. En este apartado trataremos de determinar, para cada algoritmo, si existe una diferencia relevante entre el uso de getTopWidgets y la ausencia del mismo. Las hipótesis planteadas son:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X (con getTopWidgets)	Cob. media X	Algoritmo Y (sin getTopWidgets)	Cob. media Y	P-value	Rechazar H0
Random		Random		1	No
Random+	31.8	Random+	35.7	0.03026	Sí
Qlearning		Qlearning		0.1937	No
Qlearning+		Qlearning+		0.1272	No
Maxcoverage		Maxcoverage		0.8188	No

Figura 38: Prueba Mann-Whitney-Wilcoxon para FreeMind con y sin getTopWidgets

Al formular las hipótesis y sustituyendo las variables Algoritmo X y Algoritmo Y por las que se ven reflejadas en la Figura 38, podemos concluir que existen diferencias significativas entre utilizar getTopWidgets o no para el algoritmos Random+. Nuevamente, para saber en qué caso se obtiene una mejor cobertura basta con hacer la media de los diferentes conjuntos de datos. Podemos concluir que en el SUT FreeMind Random+ obtiene mejores resultados si no se emplea el método getTopWidgets.

4.3.3. Eclipse

Procedemos a formular las diferentes hipótesis sobre los resultados obtenidos con el SUT Eclipse:

1. Hipótesis sobre la eficacia de utilizar getTopWidgets.

- H0: es indiferente utilizar getTopWidgets o no sobre los diferentes algoritmos.
- H1: no es indiferente utilizar getTopWidgets o no sobre los diferentes algoritmos.

Para aceptar o rechazar la hipótesis H0 realizamos la prueba Mann-Whitney-Wilcoxon Test sobre dos conjuntos de valores como hicimos con el SUT anterior. El primero será el conjunto de valores de cobertura alcanzada utilizando getTopWidgets y el segundo el conjunto de valores de cobertura alcanzada sin utilizar dicho método. Para esta prueba ejecutamos la función wilcox.text con el fin de determinar qué hipótesis es la correcta. Si el resultado de p-value es inferior a 0.05 se rechaza la hipótesis nula, H0, y se acepta la hipótesis H1.

Como resultado de este valor p-value obtenemos 0. 3506, por lo que no podemos rechazar la hipótesis H0 y demostrar mejora entre utilizar getTopWidgets y no utilizarlo.

2. Hipótesis comparativa entre los cinco algoritmos utilizando getTopWidgets.

Como en el caso anterior, procedemos a ilistrar en la Figura 39 los resultados de p-value obtenidos en la prueba Mann-Whitney-Wilcoxon. Así se podrá justificar la existencia de relaciones de efectividad entre los diferentes algoritmos utilizando getTopWidgets, o ausencia de las mismas. Las hipótesis formuladas son:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X	Algoritmo Y	P-value	Rechazar H0
Random	Random+	0.1851	No
Random	Qlearning	1	No
Random	Qlearning+	0.6506	No
Random	Maxcoverage	0.6506	No
Random+	Qlearning	0.1851	No
Random+	Qlearning+	0.398	No
Random+	Maxcoverage	0.398	No
Qlearning	Qlearning+	0.6506	No
Qlearning	Maxcoverage	0.6506	No
Qlearning+	Maxcoverage	1	No

Figura 39: Prueba Mann-Whitney-Wilcoxon para Eclipse getTopWidgets

Alternando las variables Algoritmo X y Algoritmo Y por las que aparecen representadas en la Figura 39, podemos concluir que para Eclipse no existen datos lo suficientemente equidistantes para poder afirmar que utilizando getTopWidgets algún método sea más beneficioso que otro.

3. Hipótesis comparativa entre los cinco algoritmos sin utilizar getTopWidgets.

El procedimiento a seguir en este apartado es similar al del caso anterior. La formulación de las hipótesis adquiere la siguiente estructura:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X	Algoritmo Y	P-value	Rechazar H0
Random	Random+	0.3017	No
Random	Qlearning	0.3017	No
Random	Qlearning+	0.5107	No
Random	Maxcoverage	0.7024	No
Random+	Qlearning	1	No
Random+	Qlearning+	0.6934	No
Random+	Maxcoverage	0.5505	No
Qlearning	Qlearning+	0.6934	No
Qlearning	Maxcoverage	0.5505	No
Qlearning+	Maxcoverage	0.8296	No

Figura 40: Prueba Mann-Whitney-Wilcoxon para Eclipse sin getTopWidgets

Formulando las hipótesis anteriormente planteadas y sustituyendo las variables Algoritmo X y Algoritmo Y por las que se ven reflejadas en la Figura 40, podemos concluir que en Elcipse cuando no se utiliza getTopWidgets no existe una diferencia lo suficientemente destacable entre los resultados producidos por los diferentes algoritmos de TESTAR.

4. Hipótesis sobre la efectividad del método getTopWidgets en cada uno de los algoritmos.

En la primera hipótesis planteada para este SUT hemos comprobado que no es posible afirmar que el hecho de usar o no usar getTopWidgets de forma general para los cinco algoritmos produzca mejores resultados de cobertura. En este apartado trataremos de determinar de forma específica para cada algoritmo si existe una diferencia relevante entre el uso de getTopWidgets y la ausencia del mismo. Las hipótesis planteadas son:

- H0: es indiferente utilizar el algoritmo X o el algoritmo Y.
- H1: no es indiferente utilizar el algoritmo X o el algoritmo Y.

Algoritmo X (con getTopWidgets)	Algoritmo Y (sin getTopWidgets)	P-value	Rechazar H0
Random	Random	0.9624	No
Random+	Random+	1	No
Qlearning	Qlearning	0.1851	No
Qlearning+	Qlearning+	0.6809	No
Maxcoverage	Maxcoverage	0.9301	No

Figura 41: Prueba Mann-Whitney-Wilcoxon para Eclipse con y sin getTopWidgets

Al formular las hipótesis y sustituyendo las variables Algoritmo X y Algoritmo Y por las que se ven reflejadas en la Figura 41, no podemos concluir que existan diferencias significativas entre el uso y no uso de getTopWidgets en ninguno de los cinco algoritmos.

A la vista de los resultados, no podemos afirmar que exista una mejora significativa entre los distintos algoritmos de TESTAR para el SUT Eclipse. Tampoco se ha podido probar la ventaja o desventaja que supone el hecho de usar o no el método getTopWidgets.

5. Conclusiones y trabajo futuro

Como se ha manifestado en varias ocasiones anteriormente, el objetivo de este trabajo ha sido tratar dar respuesta las preguntas de investigación planteadas en el apartado anterior. Con ese fin, en primer lugar se ideó una infraestructura reutilizable que permitiera ejecutar el software TESTAR sobre diferentes aplicaciones para posteriormente obtener información referente a la cobertura de código alcanzada en dichas pruebas. Seguidamente se generaron 100 secuencias de 1000 acciones mediante TESTAR sobre cada una de las interfaces gráficas de las aplicaciones a estudiar o SUTs. Estas 100 secuencias totales para cada SUT han estado organizadas o divididas entre los diferentes algoritmos de selección de acciones de los que dispone TESTAR; random, random+, qlearning, qlearning+ y maxcoverage. De esta manera obtenemos 20 pruebas realizadas por cada uno de los algoritmos, y recordemos que de estas 20 pruebas, 10 se realizaron sirviéndose del método getTopWidgets y las 10 restantes sin utilizarlo. En el momento en el que el conjunto total de las 300 pruebas finalizaron se procedió a realizar el análisis comparativo entre los resultados obtenidos para cada SUT, teniendo que:

- Para jEdit existe una diferencia considerable entre el hecho de utilizar o de no utilizar el método getTopWidgets a la hora de seleccionar acciones sobre la interfaz, lográndose mayores porcentajes de cobertura al emplear dicho método. Posteriormente, analizando los resultados obtenidos al utilizar y al no utilizar getTopWidgets se pudo determinar que al emplear dicho método, los algoritmos Random y Qlearning+ son más efectivos que el algoritmo Maxcoverage. Por otra parte, en los experimentos en los que no se utilizó getTopWidgets los resultados indican que es preferible utilizar los algoritmos Random, Qlearning y Maxcoverage frente al algoritmo Random+. Finalmente, estudiando las diferencias para cada uno de los algoritmos entre el uso y el no uso de getTopWidgets pudimos concluir que para Random+ es más efectivo utilizar dicho método, mientras que para Maxcoverage ocurre lo contrario.
- Para FreeMind, al igual que para jEdit, también podemos apreciar diferencias en los resultados obtenidos en función de si se hizo uso o no del método getTopWidgets. A diferencia del SUT anterior, en este caso se alcanzaba una mayor cobertura al no utilizar el método. Prestando atención a las diferencias entre el uso de los diferentes algoritmos de TESTAR para FreeMind pudimos determinar que al utilizar getTopWidgets conviene usar Qlearning en vez de Random+. También pudimos conocer el hecho de que para este último algoritmo (Random+) se alcanzan mejores resultados si no se emplea el método getTopWidgets.
- Lamentablemente, una vez analizados los resultados obtenidos con Eclipse no es posible determinar el posible beneficio que supondría utilizar getTopWidgets o no utilizarlo, así como tampoco determinar qué algoritmo supondría una mejora sobre los otros. Achacamos esta incapacidad de establecer relaciones entre los resultados a los bajos resultados obtenidos de cobertura de código. Debido a la frustración que afloró a causa de estos bajos indices de cobertura, realizamos una reflexión acerca de qué se podría hacer para poder aumentar la

cantidad de código cubierta, tras lo cual se procedió a la ejecución de una prueba un tanto particular sobre este SUT. La singularidad de esta prueba residía en que, a diferencia de las que se efectuaron, no se estableció un límite máximo de acciones a efectuar, sino que este límite estuvo marcado por la duración de la prueba. De esta forma, se configuró TESTAR para que realizara pruebas sobre Eclipse durante el plazo de un día entero. Tras la finalización de dicho tiempo, se analizaron los resultados obtenidos, llegando en este caso a alcanzar el 24% de cobertura de código. Con este resultado podemos concluir que para Eclipse los resultados de cobertura están directamente ligados a la duración de las pruebas que se ejecuten. Sin embargo, nos ha sido imposible repetir 100 veces este experimento debido a la gran cantidad de tiempo que hubiera consumido realizar pruebas de tan elevada duración.

Teniendo en cuenta lo que se acaba de manifestar procedemos a resolver las preguntas de investigación.

- RQ1: ¿Cómo es posible medir la cobertura de código cuando el testing es realizado a nivel de interfaz gráfica de usuario mediante la herramienta TESTAR?

Esto ha podido realizarse gracias a la herramienta JaCoCo, la cual permite la generación de informes que presentan información relativa a la cantidad de código cubierto durante la interacción de TESTAR con la aplicación bajo testeo. En la sección 3.1 Diseño y definición del experimento, aparece detalladamente el proceso para montar la infraestructura necesaria que permite realizar pruebas mediante TESTAR sobre un SUT determinado con el posterior fin de poder obtener información relativa a la cobertura de código alcanzada. Como se dijo anteriormente, este proceso no es intrínseco a los SUTs estudiados en este proyecto, sino que podrá aplicarse a diferentes aplicaciones software, permitiendo a lo largo del tiempo y en sucesivas investigaciones conocer la evolución de algoritmos y métodos futuros de la herramienta TESTAR .

- RQ2: ¿Podemos servirnos de la cobertura de código para obtener conclusiones sobre la calidad de las pruebas realizadas mediante TESTAR?

Sí podemos utilizar la cobertura de código obtenida para establecer conclusiones sobre las pruebas generadas. Los resultados correspondientes a la cobertura de código alcanzada por una prueba son directamente proporcionales a la calidad de la misma, como hemos explicado al comienzo del documento. Recordando los resultados máximos de cobertura alcanzada para cada SUT; jEdit (33%) , FreeMind (41%) y Eclipse (12%) podemos afirmar que la cobertura alcanzada, y por tanto la calidad de las pruebas, es mayor para las aplicaciones jEdit y FreeMind.

- RQ3: ¿Se obtienen diferencias significativas al variar entre los diferentes algoritmos de TESTAR, así como del uso del método getTopWidgets?

A la vista de los resultados obtenidos podemos concluir que con los experimentos realizados no podemos afirmar que exista un método de los 5 disponibles que siempre vaya a proporcionar una cobertura considerablemente mayor que los demás. A pesar de la inexistencia de un algoritmo que en todas las pruebas sea considerablemente

superior a los demás, hemos visto que para jEdit y FreeMind sí que existen casos en los que supone mayor beneficio utilizar un determinado algoritmo frente a otro. De esta forma, hemos comprobado que para estos dos SUTs se dan casos en los que utilizar un determinado algoritmo supone una ventaja o inconveniente (véase Figura 42 y Figura 43).

Random	Random+
Mejor resultado frente a Maxcoverage utilizando getTopWidgets (jEdit)	Peor resultado frente a Random, Qlearning y Maxcoverage sin getTopWidgets(jEdit)
Mejor resultado frente a Random+ sin utilizar getTopWidgets (jEdit)	Peor resultado frente a Qlearning utilizando getTopWidgets (FreeMind)

Figura 42: Resultados algoritmos (I)

Qlearning	Qlearning+	Maxcoverage
Mejor resultado frente a Random+ sin getTopWidgets (jEdit)	Mejor resultado frente a Maxcoverage utilizando getTopWidgets (jEdit)	Mejor resultado frente a Random+ sin getTopWidgets (jEdit)
Mejor resultado frente a Random+ con getTopWidgets (FreeMind)		Peor resultado frente a Qlearning+ con getTopWidgets (jEdit)

Figura 43: Resultados algoritmos (II)

Recordando una vez más que no es posible determinar un algoritmo que siempre obtenga unos resultados de cobertura considerablemente mejores que el resto, con esta información podemos sospechar que Random y Qlearning son más efectivos que el resto de los algoritmos disponibles en TESTAR.

Lo que sí que podemos determinar basándonos en los resultados de cobertura alcanzada para los SUTs jEdit y FreeMind es que sí que existe una diferencia a tener en cuenta entre los resultados obtenidos con las pruebas en las que se utilizó el método getTopWidgets y en las que no se utilizó. A pesar de que queda demostrada la existencia de esta diferencia para estos SUTs, es cierto que para jEdit se obtiene una mejora de la cobertura de código cubierto si se utiliza el método, mientras que para FreeMind se llega a más parte del código fuente de la aplicación si no se utiliza. Teniendo en cuenta esto último junto con el hecho de que las interfaces gráficas de las aplicaciones son únicas, no sería nada descabellado pensar que la efectividad del uso del método getTopWidgets o ausencia del mismo esté condicionada por cada interfaz particular.

- RQ4: ¿Cómo es posible dirigir el testing a nivel de GUI para aumentar los resultados de cobertura?

Como se mencionó en la sección en la que se explicaba TESTAR, esta herramienta permite establecer una serie de filtros. Estos filtros son expresiones regulares que permiten restringir acciones sobre la GUI de un SUT determinado. De esta manera añadiendo una expresión regular como 'Imprimir' evitaremos que TESTAR interaccione con cualquier elemento que lleve la expresión 'Imprimir' en su título. Sin embargo, aunque de esta manera se puede dirigir el testing sobre la GUI, realmente lo que se hace es acotar las posibles interacciones sobre el SUT. Este hecho lleva a que no se ejecute el código correspondiente a la acción de pulsar sobre el botón impirmir, con lo que ese código nunca será cubierto. Por consiguiente entendemos que la única manera de poder dirigir el testing a nivel de GUI es mediante el refinamiento de los protocolos de la herramienta.

Teniendo en mente todo lo manifestado a lo largo del documento, consideramos que este proyecto servirá de gran ayuda en futuras investigaciones en las que se desee realizar estudios sobre la cobertura de código lograda por la herramienta TESTAR. Hemos estudiado cómo poder ejecutar pruebas sobre diferentes aplicaciones software para posteriormente averiguar qué cantidad del SUT ha sido testeado, hecho que hasta la fecha no se había realizado con el uso de TESTAR. Hemos puesto a disposición del lector una serie de métodos y algoritmos en R que permitirán procesar y analizar de manera rápida y eficaz la información previamente obtenida del software JaCoCo. En este trabajo hemos analizado 3 SUTs distintos, sobre los que hemos efectuado un total de 100 pruebas a cada uno alternando entre los diferentes algoritmos presentes en la herramienta y la presencia o ausencia del método getTopWidgets. Cada una de estas pruebas estaba formada por una secuencia de 1000 acciones sobre la interfaz gráfica del SUT. Para poder obtener información futura sobre la cobertura alcanza con TESTAR será deseable estudiar diferentes aplicaciones en las que se realicen pruebas con el mayor número posible de secuencias, así como del número de acciones efectuadas en cada una de las mismas. Los trabajos actuales de TESTAR [17][18] están investigando cómo usar algoritmos evolutivos o de machine learning para mejorar la selección de acciones sobre la interfaz de una aplicación dada. Sería interesante contemplar la posibilidad de que esta selección estuviera basada en la cobertura obtenida, y así poder obtener reglas que den lugar a una mayor cobertura durante la ejecución de pruebas.

6. Reflexión personal

Para finalizar me gustaría realizar una reflexión personal de lo que ha supuesto la realización de este trabajo. Al comienzo del mismo, el poco contacto que había tenido con el mundo del testing se había limitado a realizar pruebas unitarias sobre diferentes programas Java. Previamente había leído información acerca de la automatización de pruebas, algo que consideraba interesante ya que permitiría ahorrar gran cantidad de tiempo a la hora de testear una aplicación. De lo que nunca había tenido constancia era de la posibilidad de realizar pruebas sobre la interfaz gráfica de las aplicaciones de manera automatizada, hecho que me pareció aun más llamativo. Por esos motivos me resultó en un principio tan interesante TESTAR, así como el hecho de que mi Trabajo de Fin de Máster pudiera estar relacionado con el uso de esta herramienta. Como se ha dicho a lo largo de este documento, TESTAR no proporciona información sobre la cobertura de código obtenida en la ejecución de sus pruebas, por lo que la propuesta de mi tutora de medir y analizar este dato en los tests efectuados sobre diferentes aplicaciones me pareció un tema llamativo a la par que novedoso. A partir de la definición de los objetivos comenzó el proceso de estudio de cómo conseguir crear la infraestructura necesaria para conectar TESTAR con JaCoCo y los tres SUTs estudiados. El hecho de obtener el primer informe con la información relativa a la cobertura obtenida en las pruebas generadas con TESTAR suspuso una gran alegría, y trajo consigo el deseo de poder automatizar el proceso necesario para poder obtener el número total de informes que se había establecido. Una vez logrado, comenzó el proceso de análisis, en el que se vio que los resultados obtenidos con jEdit y FreeMind eran útiles para poder determinar el beneficio de utilizar el método getTopWidgets, o de no utilizarlo, así como se pudo observar el beneficio que suponía utilizar determinados algoritmos frente a otros. Esto constituyó otro de los placeres experimentados durante la realización de este trabajo. Por otra parte, con el análisis de los resultados de Eclipse llegó uno de los peores momentos, ya que eran bastante bajos y poco útiles para establecer relaciones de efectividad entre los algoritmos de TESTAR así como del método getTopWidgets. Como se comentó anteriormente, se realizó una prueba de un día de duración en la que pudimos apreciar que la cobertura alcanzada era mucho mayor. Con esto al menos encontramos una explicación de porqué no se había logrado alcanzar una cobertura mayor en las 100 pruebas efectuadas durante el experimento. El problema era que no disponíamos de tiempo suficiente para realizar 100 pruebas de 24 horas, así como el hecho de que considerábamos adecuado que todas las pruebas efectuadas sobre los diferentes SUTs tuvieran la misma duración, o en su defecto el mismo número de acciones. Sabiendo esto, surgió el deseo de, una vez acabada la realización de este trabajo, poder realizar pruebas de mayor duración y así poder extraer información útil de la cobertura alcanzada, sin que exista una fecha límite en la que tengan que estar preparados los

resultados y poder realizar así esta tarea con mayor tranquilidad. Para finalizar, con la infraestructura ideada en el proyecto que permite combinar TESTAR con JaCoCo y un SUT, surgió un nuevo sentimiento de bienestar al pensar que esta infraestructura será útil no solo en este trabajo, sino en futuras investigaciones que quieran estudiar la cobertura para evaluar las pruebas realizadas con TESTAR. De hecho, pocos días atrás recibí un correo de una chica holandesa estudiante de máster pidiéndome información sobre el trabajo que había realiazado, ya que consideraba que podía serle de utilidad en la realización de su propio proyecto. Este correo trajo consigo una inyección de orgullo y jovialidad, ya que en mis años como estudiante he proporcionado ayuda a mis compañeros de aula cuando me la han solicitado, pero nunca hasta la fecha pensé que el trabajo que desempeño podría ser de interés y servir de ayuda a estudiantes ajenos a mi entorno.

7. Agradecimientos

Quisiera dedicar unas palabras de agradecimiento a las personas involucradas en la realización de este trabajo. En primer lugar a mi tutora Tanja E.J. Vos por la ayuda prestada, sus consejos y la forma de guiarme durante todo el proyecto.

A mi cotutora, Anna I Esparcia, por la experiencia que posee, la cual ha servido para mejorar diferentes elementos clave de esta investigación.

También me gustaría dejar constancia de la ayuda prestada por Urko Rueda Molino. El entendimiento de la naturaleza de TESTAR no hubiera sido posible sin sus explicaciones, así como la resolución de los problemas relativos a la herramienta que surgieron durante la realización de este proyecto.

Por último, a las personas que no han estado directamente involucradas en la realización de este Trabajo de Fin de Máster. Estoy hablando cómo no, de mi familia y amigos. He tenido el apoyo constante de todos ellos durante los momentos de dificultad del proyecto, así como su compañía a la hora de celebrar las metas alcanzadas.

A todos vosotros, gracias.

8. Bibliografía

[1] "The Importance of Testing In Software Development - Axis Technical Group", Axis Technical Group. [Online]. Available: http://www.axistechnical.com/importance-testing-software-development/. [Accessed: 12- Aug- 2017].

[2] 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.

[3] Msdn.microsoft.com. (n.d.). *Windows Automation API (Windows)*. [online] Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/ff486375(v=vs.85).aspx [Accessed 7 Jun. 2017].

[4] 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.

[5] 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.

[6] 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.

[7] 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

[8] C. Pfaller, S. Wagner, J. Gericke and M. Wiemann, "Multi-Dimensional Measures for Test Case Quality," *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, Lillehammer, 2008, pp. 364-368.

[9] Jacoco.org. (n.d.). *JaCoCo - Documentation*. [online] Available at: http://www.jacoco.org/jacoco/trunk/doc/ [Accessed 10 Jun. 2017].

[10] "The Importance of Testing In Software Development - Axis Technical Group", Axis Technical Group. [Online]. Available: http://www.axistechnical.com/importance-testing-software-development/. [Accessed: 12- Aug- 2017].

[11] Cillero, M. (n.d.). Pruebas - manuel.cillero.es. [online] manuel.cillero.es. Available at: https://manuel.cillero.es/doc/metrica-3/tecnicas/pruebas/ [Accessed 7 Jul. 2017].

[12] Pmoinformatica.com. (2017). Pruebas de caja negra: Ejemplos. [online] Available at: http://www.pmoinformatica.com/2017/02/pruebas-de-caja-negra-ejemplos.html [Accessed 2 Aug. 2017].

[13] Es.wikipedia.org. (2017). Pruebas de caja blanca. [online] Available at: https://es.wikipedia.org/wiki/Pruebas_de_caja_blanca [Accessed 2 Aug. 2017].

[14] E. Figueiredo Collins and V. Ferreira de Lucena, "Software test automation practices in agile development environment: an industry experience report", Proceeding AST '12 Proceedings of the 7th International Workshop on Automation of Software Test, pp. 57-63, 2016.

[15] A. Memon, "A comprehensive framework for testing graphical user interfaces", Doctoral Dissertation, University of Pittsburgh, 2001.

[16] Esparcia-Alcázar A.I., Almenar F., Rueda U., Vos T.E.J. (2017) Evolving Rules for Action Selection in Automated Testing via Genetic Programming - A First Approach. In: Squillero G., Sim K. (eds) Applications of Evolutionary Computation. EvoApplications 2017. Lecture Notes in Computer Science, vol 10200. Springer, Cham

[17] Bauersfeld, Sebastian and Tanja E. J. Vos. “User Interface Level Testing with TESTAR; What about More Sophisticated Action Specification and Selection?” SATToSE (2014).

[18] Anna I. Esparcia-Alcázar, F. Almenar, M. Martínez, U.Rueda, and Tanja E.J. Vos, “Q-learning strategies for action selection in the TESTAR automated testing tool”, Proceedings of 6th International Conference on Metaheuristics and Nature Inspired Computing META’16, Marrakech, Morocco, Oct 27-31, 2016.

[19] TESTAR user manual, version from August 2017, testar.org.

[20] C. Wohlin, Experimentation in software engineering. Boston [u.a.]: Kluwer, 2002, pp. 123-140.

[21] Jedit.org. (2017). jEdit - Programmer's Text Editor -overview. [online] Available at: http://www.jedit.org/ [Accessed 4 Jul. 2017].

[22] Freemind.sourceforge.net. (n.d.). Main Page - FreeMind. [online] Available at: http://freemind.sourceforge.net/wiki/index.php/Main_Page [Accessed 14 Jul. 2017].

[23] Anon, (n.d.). [online] Available at: https://eclipse.org/mars/ [Accessed 6 Jul. 2017].

[24] "Code Coverage Tools (JaCoCo, Cobertura, Emma) Comparison in Sonar", Only Software matters. [Online]. Available: https://onlysoftware.wordpress.com/2012/12/19/code-coverage-tools-jacoco-cobertura-emma-comparison-in-sonar/. [Accessed: 14- Aug- 2017].

[25] MacNeill, C. and Bodewig, S. (n.d.). Apache Ant - Welcome. [online] Ant.apache.org. Available at: http://ant.apache.org/ [Accessed 11 Jun. 2017].

[26] "EclEmma - JaCoCo Java Code Coverage Library", Eclemma.org. [Online]. Available: http://www.eclemma.org/jacoco/. [Accessed: 05- Jun- 2017].

[27] C. Wohlin, M. Höst and K. Henningsson, "Empirical Research Methods in Software Engineering", In Lecture Notes in Computer Science: Empirical Methods and Studies in

Software Engineering: Experiences from ESERNET, edited by A. I. Wang and R. Conradi, Springer Verlag, 2003.

[28] R-project.org. (n.d.). R: The R Project for Statistical Computing. [online] Available at: https://www.r-project.org/ [Accessed 9 Aug. 2017].

[29] R-tutor.com. (n.d.). Mann-Whitney-Wilcoxon Test | R Tutorial. [online] Available at: http://www.r-tutor.com/elementary-statistics/non-parametric-methods/mann-whitney-wilcoxon-test [Accessed 10 Aug. 2017].

9. Anexo

En este apartado presenta una guía para poder utilizar los métodos desarrollados en R que permitan en estudios posteriores obtener información sobre los informes generados mediante JaCoCo. Recordemos que dichos informes son generados en formato html. Necesitaremos una conversión a otro formato, por ejemplo csv, para poder trabajar con ellos en R. Para ello proponemos una manera sencilla, la cual consiste en abrir en Excel el archivo html, ir a la pestaña Archivo, Guardar como, y elegir el formato CSV (delimitado por comas).

Una vez hecho esto ya podemos trabajar con la información obtenida. Para ejemplificar las explicaciones (véase Figura 44), utilizaremos por ejemplo uno de los informes obtenidos para FreeMind (concretamente para el algoritmo random utilizando getTopWidgets).

```
frt_1 <- read.csv("freemind/random/top/index1.csv", header = FALSE, sep = ";")

frt_1 <- frt_1[6:49,]

nombres_columnas <- c("Element",  "Missed Instructions",  "Cov.", "Missed Branches",
"Cov.2",  "Missed"  ,"Cxty" ,"Missed2"  ,"Lines", "Missed3",  "Methods" ,"Missed4"
,"Classes")

colnames(frt_1) <- nombres_columnas

frt1 <- data.frame(frt_1)
```

Figura 44: Lectura informe en R

En primer lugar debemos cargar el archivo correspondiente a dicho informe, especificando la ruta en la que se encuentre. Debido a la estructura inicial de este documento debemos de eliminar alguna de sus filas y trabajar con las que son de nuestro interés. Las columnas aparecen en este momento sin su título correspondiente, por lo que se lo añadimos para que tenga la misma estructura que tenía en el documento html inicial.

Este procedimiento puede repetirse con todos los informes JaCoCo con los que se desee trabajar, almacenando cada uno de ellos en variables diferentes para evitar pérdida de información. Una vez hecho esto existen diferentes métodos y funciones que nos permitirán obtener información de los informes:

Cobertura media alcanzada, Figura 45. De esta forma podremos agrupar unas variables en otras de manera que una de estas variables almacene la información relativa a la cobertura media alcanzada para un determinado SUT, con un determinado algoritmo, utilizándose o no getTopWidgets.

```
cov_average <- function(x)
{
  return (as.numeric(sub("%", "", x[1,]$Cov.)))
}

fcv_rt <- c(cov_average(frt1),cov_average(frt2),cov_average(frt3),cov_average(frt4),
cov_average(frt5),cov_average(frt6),cov_average(frt7),cov_average(frt8),cov_average(
frt9),cov_average(frt10))
```

Figura 45: Método cobertura R

Una vez agrupados todos los datos de cobertura en función del algoritmo utilizado podremos obtener representación visual de las mismas mediante diagrama de cajas (véase Figura 46):

```
boxplot(fcv_rt,fcv_rn,fcv_rmt,fcv_rmn,fcv_qt,fcv_qn,fcv_qmt,fcv_qmn,fcv_mt,fcv_mn,
names = c("R Top", "R NoTop","R+ Top","R+ NoTop", "Q Top","Q NoTop","Q+ Top","Q+
NoTop","MC Top","MC NoTop"), cex.axis = 0.7)
```

Figura 46: Código diagram cajas R

Para obtener una mayor precisión sobre las medias de cobertura alcanzadas, en la Figura 47 se aprecia el código necesario para lograr una tabla que lo refleje.

```
tf <- (rbind(cbind(mean(fcv_rt),mean(fcv_rn))))
tf <- rbind(tf, cbind(mean(fcv_rmt),mean(fcv_rmn)))
tf <- rbind(tf, cbind(mean(fcv_qt),mean(fcv_qn)))
tf <- rbind(tf, cbind(mean(fcv_qmt),mean(fcv_qmn)))
tf <- rbind(tf, cbind(mean(fcv_mt),mean(fcv_mn)))
colnames(tf) <- c("TopWidgets", "Sin TopWidgets")
row.names(tf) <- c("Random", "Random+","Qlearning","Qlearning+","MaxCoverage")
```

Figura 47: Tabla media cobertura R

Con el fin de obtener una representación mediante diagramas del número de tests que superan la cobertura media alcanzada tenemos la función codificada en la Figura 48:

```
More_mean <- function(sut_mean,rt,rn,rmt,rmn,qt,qn,qmt,qmn,mt,mn)
{
  array_mj <- c(0,0,0,0,0,0,0,0,0,0)
  for(i in 1:10)
  {
    if(rt[i] > sut_mean){array_mj[1] <- array_mj[1] + 1}
    if(rn[i] > sut_mean){array_mj[2] <- array_mj[2] + 1}
    if(rmt[i] > sut_mean){array_mj[3] <- array_mj[3] + 1}
    if(rmn[i] > sut_mean){array_mj[4] <- array_mj[4] + 1}
    if(qt[i] > sut_mean){array_mj[5] <- array_mj[5] + 1}
    if(qn[i] > sut_mean){array_mj[6] <- array_mj[6] + 1}
    if(qmt[i] > sut_mean){array_mj[7] <- array_mj[7] + 1}
    if(qmn[i] > sut_mean){array_mj[8] <- array_mj[8] + 1}
    if(mt[i] > sut_mean){array_mj[9] <- array_mj[9] + 1}
    if(mn[i] > sut_mean){array_mj[10] <- array_mj[10] + 1}
  }
  return(array_mj)
}
```

Figura 48: Método More_mean R

Para invocar el método anterior es necesario pasarle como parámetro la media obtenida en el SUT a estudiar y las variables que almacenan los registros de cobertura de cada algoritmo (véase Figura 49).

```
fmean <- mean(c(fcv_rt,fcv_rn,fcv_rmt,fcv_rmn,fcv_qt,fcv_qn,fcv_qmt,fcv_qmn,fcv_m
t,fcv_mn))

barplot(More_mean(fmean,fcv_rt,fcv_rn,fcv_rmt,fcv_rmn,fcv_qt,fcv_qn,fcv_qmt,fcv_q
mn,fcv_mt,fcv_mn), names = c("R Top", "R NoTop","R+ Top","R+ NoTop", "Q Top","Q
NoTop","Q+ Top","Q+ NoTop","MC Top","MC NoTop"), cex.names = 0.7 , ylim =
c(0,10), col = c("orange","orange","steelblue2","steelblue2","lawngreen","lawngre
en","gold","gold","coral","coral"))
```

Figura 49: Llamada a More_mean R

Igual que podemos saber para un determinado SUT el número de tests situados por encima de la media para cada algoritmo, podemos saber los que se encuentran por debajo de dicha media (véase Figura 50).

```
Less_mean <- function(sut_mean,rt,rn,rmt,rmn,qt,qn,qmt,qmn,mt,mn)
{
  array_mj <- c(0,0,0,0,0,0,0,0,0,0)
  for(i in 1:10)
  {
    if(rt[i] < sut_mean){array_mj[1] <- array_mj[1] + 1}
    if(rn[i] < sut_mean){array_mj[2] <- array_mj[2] + 1}
    if(rmt[i] < sut_mean){array_mj[3] <- array_mj[3] + 1}
    if(rmn[i] < sut_mean){array_mj[4] <- array_mj[4] + 1}
    if(qt[i] < sut_mean){array_mj[5] <- array_mj[5] + 1}
    if(qn[i] < sut_mean){array_mj[6] <- array_mj[6] + 1}
    if(qmt[i] < sut_mean){array_mj[7] <- array_mj[7] + 1}
    if(qmn[i] < sut_mean){array_mj[8] <- array_mj[8] + 1}
    if(mt[i] < sut_mean){array_mj[9] <- array_mj[9] + 1}
    if(mn[i] < sut_mean){array_mj[10] <- array_mj[10] + 1}
  }
  return(array_mj)
}

barplot(Less_mean(fmean,fcv_rt,fcv_rn,fcv_rmt,fcv_rmn,fcv_qt,fcv_qn,fcv_qmt,fcv_q
mn,fcv_mt,fcv_mn), names = c("R Top", "R NoTop","R+ Top","R+ NoTop", "Q Top","Q
NoTop","Q+ Top","Q+ NoTop","MC Top","MC NoTop"), cex.names = 0.7 , ylim =
c(0,10), col = c("orange","orange","steelblue2","steelblue2","lawngreen","lawngre
en","gold","gold","coral","coral"))
```

Figura 50: Método Less_mean R

Para obtener información relativa a la cobertura de rama obtenida, o branch coverage, definimos una función que nos permita obtener este dato para posteriormente poder representar el conjunto total mediante diagramas de barras (véase Figura 51).

```
Branch <- function(x)
{
  return (as.numeric(sub("%", "", x[1,]$Cov.2)))
}

fb_rt <- c(Branch(frt1),Branch(frt2),Branch(frt3),Branch(frt4),Branch(frt5),Branc
h(frt6),Branch(frt7),Branch(frt8),Branch(frt9),Branch(frt10))
boxplot(fb_rt,fb_rn,fb_rmt,fb_rmn,fb_qt,fb_qn,fb_qmt,fb_qmn,fb_mt,fb_mn, names =
c("R Top", "R NoTop","R+ Top","R+ NoTop", "Q Top","Q NoTop","Q+ Top","Q+
NoTop","MC Top","MC NoTop"), cex.axis = 0.7 , ylim = c(20,32))
```

Figura 51: Método branch coverage R

Para realizar la Prueba Mann-Whitney-Wilcoxon sobre dos conjuntos de datos en los que uno de ellos pueda ser, por ejemplo, los resultados obtenidos con el método getTopWidgets y el segundo conjunto, los resultados que se obtuvieron sin dicho método, podemos ejecutar los comandos ilustros en la Figura 52.

```
fct <- c(fcv_rt,fcv_rmt,fcv_qt,fcv_qmt,fcv_mt)
fcn <- c(fcv_rn,fcv_rmn,fcv_qn,fcv_qmn,fcv_mn)
#diferencia usar GetTopWidgets y no usarlo
wilcox.test(fct,fcn)
```

Figura 52: Prueba Mann-Whitney-Wilcoxon en R