Proyecto Final de Carrera

Diseño e implementación de un sistema para la monitorización de aplicaciones móviles

Autor:

Irene García Flores

Departamento de Arquitectura de Computadores

Universitat Politècnica de Catalunya (UPC)

Director:

Jordi Riera Molina

Director ponente:

Beatriz Otero Calviño

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Agradecimientos

A mis padres, a quienes les debo todo.

Colaboraciones

NexTReT, S.L.

Índice general

1. Resumen **7**
1.1. Castellano . . . 9
1.2. Català . . . 10
1.3. English . . . 11

2. Introducción **13**
2.1. Contexto del proyecto . . . 15
2.1.1. Usos más representativos . . . 15
2.2. Objetivos del proyecto . . . 16
2.2.1. Objetivos generales . . . 16
2.2.2. Objetivos específicos . . . 16
2.3. Estructura de la memoria . . . 17

3. Descripción global del sistema **19**
3.1. Introducción . . . 21
3.2. Descripción de las características funcionales . . . 21
3.2.1. Objetivo . . . 21
3.2.2. Módulos funcionales . . . 21
3.3. Descripción de las características técnicas . . . 25
3.3.1. Introducción . . . 25
3.3.2. Diagrama de la arquitectura . . . 25
3.4. Alcance del proyecto . . . 27
3.4.1. Actividades fuera del alcance del proyecto . . . 28

4. Módulo de intercomunicación entre agente y móvil **29**
4.1. Introducción . . . 31
4.2. Fase de investigación . . . 31
4.2.1. Objetivos . . . 31
4.2.2. Desarrollo de la búsqueda . . . 32
4.2.3. Conclusiones de la búsqueda . . . 46

5. Agente **47**
5.1. Fase de diseño . . . 49
5.1.1. Diagramas UML . . . 49
5.2. Resultado del desarrollo . . . 54
5.2.1. Estructura del Proyecto . . . 54
5.2.2. Análisis del código . . . 54

6. Web Service **71**
6.1. Introducción . . . 73
6.2. Resultado del desarrollo . . . 74
6.2.1. Agente: función *sendURLAndGetNodeResponse* de la clase *WebService.java* 74
6.2.2. *Web Service*: cómo interpretar una petición . . . 74
6.2.3. Agente y *Web Service*: funciones relacionadas con la inicialización del agente 74

6.2.4. Agente y *Web Service*: funciones relacionadas con el envío de resultados a la Base de Datos . . . 77

7. Base de datos **93**
7.1. Introducción . . . 95
7.2. Diagrama de base de datos . . . 95

8. Evaluación de la solución **99**
8.1. Control de errores . . . 101
8.2. Explotación de resultados . . . 102

9. Conclusiones finales **111**
9.1. Puntos fuertes del proyecto . . . 113
9.2. Trabajos futuros . . . 114

Glosario de términos y acrónimos **117**

Índice de figuras **119**

Bibliografía **121**

Anexos **123**

A. MonkeyRunner: *script dump.py* **125**

B. MonkeyRunner: Función *dump* del *script ViewClient.py* **131**

C. Agente: Diagrama de flujo **137**

D. Agente: Diagrama de clases **141**

E. Agente: Contenido archivo *ConfigAgente.properties* **145**

F. Diagrama de la Base de Datos **149**

G. Procedimiento MASM: Plataforma Robot **153**

H. Procedimiento MASM: Requisitos Terminal Móvil **165**

I. Procedimiento MASM: Grabar Tests Sikuli **179**

J. Procedimiento MASM: Añadir calendario a un test **187**

Capítulo 1

Resumen

1.1. Castellano

Hoy en día los teléfonos móviles son un complemento indispensable en nuestras vidas. Utilizamos todo tipo de aplicaciones, pero entre todas ellas existen unas muy concretas que debido a su naturaleza son más propensas a tener problemas: las aplicaciones interactivas, aquellas que se comunican con un servidor del que se descargan parte del contenido que muestran. Gracias a ellas consultamos la cuenta bancaria, descargamos cupones de descuento o compramos entradas de cine.

Curiosamente en las situaciones en las que utilizamos esas aplicaciones necesitamos finalizar la gestión con celeridad y, además, no disponemos de ningún ordenador, con lo que dependemos totalmente de la aplicación móvil. En consecuencia, si ésta no responde, el resultado es adverso tanto para el usuario como para la entidad que distribuye la aplicación, que tendrá pérdidas inmediatas y a corto plazo: inmediatas debidas a la cantidad de usuarios que hayan dejado de generar ingresos durante el corte del servicio y a corto plazo si usuarios frustrados dejan de utilizar la aplicación.

En ese contexto, NexTReT, la empresa en la que actualmente trabaja la autora de este PFC, vio la manera de ofrecer a esas entidades una solución con la que mantener operativas sus aplicaciones móviles el máximo tiempo posible: monitorizándolas. Para ello la mejor opción era disponer de un producto propio, con lo que se planteó a la autora de este proyecto la oportunidad de diseñarlo y desarrollarlo. El resultado de ese trabajo es el servicio que hoy en día NexTReT comercializa con el nombre de MASM.

MASM es un sistema de monitorización de aplicaciones móviles que comprueba la disponibilidad de las mismas desde el punto de vista del usuario. El funcionamiento global de la solución consiste en seguir de forma cíclica los siguientes pasos:

1. Realizar una serie de pasos concretos en la aplicación móvil (al conjunto de pasos le llamaremos "test" o "circuito"). Esos pasos podrían ser, por ejemplo: abrir la aplicación (paso 1), escoger una opción dentro de la aplicación (paso 2), introducir unas credenciales para acceder a una vista concreta (paso 3), comprobar que la vista ha cargado correctamente (paso 4) y cerrar la aplicación (paso 5).
2. Enviar a una página web, a la que llamaremos *Reporting Web*, todos los resultados obtenidos de esa ejecución.
3. En caso de que el test no finalice correctamente (por ejemplo, el paso 4 falla porque la vista no ha terminado de cargar en el tiempo máximo establecido), enviar una alerta al cliente informando con detalle el tipo de error que está teniendo la aplicación.

Para conseguir ese resultado, en primer lugar se realizó un diseño funcional que se tuvo en cuenta para el posterior diseño técnico. En esas fases previas se definió el diagrama completo de la arquitectura, cuyos elementos principales son los siguientes:

- Agente: *core* del sistema. Se encarga de ejecutar los tests sobre las aplicaciones móviles para después enviar los resultados a la Base de Datos.
- Web Service: intermediario entre agente y Base de Datos.
- Base de Datos: almacena toda la información relativa a los tests, a sus ejecuciones y a los usuarios de la web.
- *Reporting Web*: página web a la que el cliente puede acceder en todo momento para conocer el estado de sus aplicaciones móviles tanto en tiempo real como de forma histórica.
- Alertas: encargadas de avisar al cliente en caso de que la aplicación no funcione.

Tal y como se proyectó el sistema, la idea es que en caso de añadir futuras funcionalidades éstas recaigan en el agente, aplicación de escritorio Java J2SE. Por ese motivo, su desarrollo se centró en obtener un código eficiente, escalable y modular.

1.2. Català

Avui dia els telèfons mòbils són un complement indispensable en les nostres vides. Utilitzem tot tipus d'aplicacions, però entre totes elles existeixen unes molt concretes que a causa de la seva naturalesa són més propenses a tenir problemes: les aplicacions interactives, aquelles que es comuniquen amb un servidor del que es descarreguen part del contingut que mostren. Gràcies a elles consultem el compte bancari, descarreguem cupons de descompte o comprem entrades de cinema.

Curiosament en les situacions en les quals utilitzem aquestes aplicacions necessitem finalitzar la gestió amb celeritat i, a més, no disposem de cap ordinador, amb el que depenem totalment de l'aplicació mòbil. En conseqüència, si aquesta no respon, el resultat és advers tant per a l'usuari com per a l'entitat que distribueix l'aplicació, que tindrà pèrdues immediates i a curt termini: immediates degudes a la quantitat d'usuaris que hagin deixat de generar ingressos durant el tall del servei i a curt termini si usuaris frustrats deixen d'utilitzar l'aplicació.

En aquest context, NexTReT, l'empresa en la qual actualment treballa l'autora d'aquest PFC, va veure la manera d'oferir a aquestes entitats una solució amb la qual mantenir operatives les seves aplicacions mòbils el màxim temps possible: monitoritzant-les. Per fer-ho la millor opció era disposar d'un producte propi, amb el que es va plantejar a l'autora d'aquest projecte l'oportunitat de dissenyar-ho i desenvolupar-ho. El resultat d'aquest treball és el servei que avui dia NexTReT comercialitza amb el nom de MASM.

MASM és un sistema de monitoratge d'aplicacions mòbils que comprova la disponibilitat de les mateixes des del punt de vista de l'usuari. El funcionament global de la solució consisteix a seguir de forma cíclica els següents passos:

1. Realitzar una sèrie de passos concrets en l'aplicació mòbil (al conjunt de passos li anomenarem "test" o "circuit"). Aquests passos podrien ser, per exemple: obrir l'aplicació (pas 1), escollir una opció dins de l'aplicació (pas 2), introduir unes credencials per accedir a una vista concreta (pas 3), comprovar que la vista ha carregat correctament (pas 4) i tancar l'aplicació (pas 5).
2. Enviar a una pàgina web, a la qual anomenarem *Reporting Web*, tots els resultats obtinguts d'aquesta execució.
3. En cas que el test no finalitzi correctament (per exemple, el pas 4 falla perquè la vista no ha acabat de carregar en el temps màxim establert), enviar una alerta al client informant amb detall el tipus d'error que està tenint l'aplicació.

Per aconseguir aquest resultat, en primer lloc es va realitzar un disseny funcional que es va tenir en compte per al posterior disseny tècnic. En aquestes fases prèvies es va definir el diagrama complet de l'arquitectura, els elements principals de la qual són els següents:

- Agent: *core* del sistema. S'encarrega d'executar els tests sobre les aplicacions mòbils per després enviar els resultats a la Base de Dades.
- Web Service: intermediari entre agent i Base de Dades.
- Base de dades: emmagatzema tota la informació relativa als tests, a les seves execucions i als usuaris de la web.
- *Reporting Web*: pàgina web a la qual el client pot accedir en tot moment per conèixer l'estat de les seves aplicacions mòbils tant en temps real com de forma històrica.
- Alertes: encarregades d'avisar al client en cas que l'aplicació no funcioni.

Tal com es va projectar el sistema, la idea és que en cas d'afegir futures funcionalitats aquestes recaiguin en l'agent, aplicació d'escriptori Java J2ES. Per aquest motiu, el seu desenvolupament es va centrar a obtenir un codi eficient, escalable i modular.

1.3. English

Today mobile phones are an essential accessory to our lives. We use all kinds of applications, but among them there are some very specific that due to their nature are more likely to have problems: interactive applications, those that communicate with a server which content showing is downloaded. Thanks to them we are able to consult the bank account, download discount coupons, or buy cinema tickets.

Curiously, in the situations in which we use these applications, we need to finish the process quickly and also we do not have any computer, so we totally depend on the mobile application. Consequently, if it does not respond, the result is both adverse for the user and the entity that distributes the application, which will have immediate and short-term losses: due to the amount of users who did not generate income during the cutting of the service and short-term if frustrated users stop using the application.

In this context, NexTReT, the company where the author of this PFC currently works, saw the way to offer a solution to those entities to keep operating their mobile applications as long as possible: by monitoring them. To do this, the best option was to have an own product. This is how the author of this project started to design and develop it. The result of this work is a service that NexTReT currently marketed under the name of MASM.

MASM is a monitoring system of mobile applications that checks the availability of them from the point of view of the user. The overall working of the solution is to follow cyclically the following steps:

1. To carry out a series of concrete steps in the mobile application (we will call "test" or "circuit" to the set of steps). These steps could be, for example, open the application (step 1), choose an option within the application (step 2), enter credentials to access a concrete view (step 3), make sure the view is correctly loaded (step 4) and close the application (step 5).
2. To send to a website, which we will call *Reporting Web*, all the results of that execution.
3. If the test does not complete successfully (for example, step 4 fails because the view has not finished loading in maximum time set), to send an alert to the client reporting in detail type of error that is having the application.

To achieve this result, firstly a functional design which was taken into account for the subsequent technical design was designed. In these previous phases the complete diagram of the architecture is defined, which the main elements are:

- Agent: *core* system. Is responsible for executing the tests on mobile applications and then send the results to the Database.
- Web Service: intermediary between agent and Database.
- Database: stores all information relative to the tests, to their executions and to the users of the website.
- *Reporting website*: web page that the client can access at any time to know the status of their mobile applications both in real time and historical form.
- Alerts: responsible to notify the client if the application does not work.

According to how the system was designed, the idea is that if future features are added, this fall on the agent, Java J2SE desktop application. For that reason, its development was focused on obtaining an efficient, scalable and modular code.

Capítulo 2

Introducción

2.1. Contexto del proyecto

En los últimos años hemos visto cómo los "smartphones" se han incorporado a nuestras vidas hasta el punto que los utilizamos para cualquier situación cotidiana como comprar entradas de cine, descargar cupones de descuento, consultar la cuenta bancaria, realizar una transferencia, etc. Ese tipo de uso requiere de una disponibilidad inmediata de la aplicación móvil correspondiente, ya que de no ser así esperaríamos a realizar la operación en un ordenador. En consecuencia, las aplicaciones móviles deben funcionar en el momento en el que las necesitamos ya que sino dejaríamos de usarlas, provocando así una pérdida en el negocio de quien las proporciona. No obstante, para que eso sea posible, las empresas que distribuyen las aplicaciones deberían conocer su disponibilidad en tiempo real para actuar en caso necesario, pero el problema actual que tienen es precisamente la falta de control al respecto.

Existen numerosas soluciones que monitorizan la disponibilidad de servidores, *routers*, *switches*, anchos de banda, bases de datos, etc., pero que esa monitorización no detecte ninguna anomalía no asegura que el usuario final de una página web o una aplicación móvil disponga del recurso que está solicitando. Para ello es necesario disponer de una monitorización adicional que compruebe precisamente ese uso final. Para el caso de las páginas webs se pueden encontrar muchas soluciones con las que monitorizar su disponibilidad, ya que este sector ya ha sido explotado durante años, pero el caso de las aplicaciones móviles es totalmente diferente.

Si bien es cierto que se pueden encontrar distintas herramientas con las que realizar tareas de *testing* durante el desarrollo de una aplicación móvil, no ocurre lo mismo con la posibilidad controlar su comportamiento una vez implementada. Esto último es realmente importante y necesario en el caso de aplicaciones móviles interactivas (aquellas que se comunican con un servidor del que descargan parte del contenido que muestran) ya que, por muy bien que se hayan programado, pueden fallar por muchos motivos debido a su naturaleza.

Precisamente ese tipo de aplicaciones móviles son las que utilizamos en situaciones como las comentadas al inicio, con lo que, si por un lado se ha visto la necesidad de que el usuario no perciba un problema en ese tipo de aplicación y, por otro lado, la no despreciable probabilidad de que eso ocurra, la necesidad de monitorizar esas aplicaciones es evidente.

En ese contexto se inició este Proyecto Final de Carrera (PFC). La empresa NexTReT, donde actualmente trabaja la autora de este proyecto, tras analizar las soluciones existentes en el mercado vio una oportunidad de abrirse camino en la monitorización de aplicaciones móviles. Las opciones de la competencia, además de escasas, requerían de un esfuerzo económico considerable debido a los complejos sistemas que utilizaban, con lo que, con una opción alternativa de precio competitivo, NexTReT podría conseguir su objetivo. Así fue como a la autora de este proyecto le llegó la oportunidad de diseñar y desarrollar una solución que desempañara esa función. Se aceptó el reto y tras muchos meses de trabajo se llegó a lo que hoy en día NexTReT está ofreciendo con el nombre comercial de MASM.

2.1.1. Usos más representativos

El sistema de monitorización de aplicaciones móviles que se ha desarrollado en este PFC va dirigido a cualquier tipo de organización, nacional o internacional, que preste sus servicios *online* a través de plataformas de movilidad. Se pueden destacar las siguientes:

- Administración pública en todos sus niveles.
- Compañías de *eCommerce* de todos los sectores.
- Sector financiero.
- Compañías de nicho de Internet con soluciones particularmente adaptadas a la movilidad.

2.2. Objetivos del proyecto

2.2.1. Objetivos generales

Desde el punto de vista académico, al inicio de este PFC el objetivo era diseñar e implementar un sistema de monitorización de aplicaciones móviles robusto y eficaz, centrándose en conseguir la optimización de cada uno de los elementos a desarrollar.

Por otro lado, desde el punto de vista comercial, la finalidad de iniciar este proyecto fue conseguir una solución de monitorización de aplicaciones móviles orientada a mejorar la competitividad de las empresas en el ámbito del comercio electrónico. Para ello esta solución debía detectar errores y caídas de servicios móviles en tiempo real con el objetivo de avisar a los responsables de la misma. De esa forma, la actuación es inmediata y la afectación a los usuarios reales de la aplicación, mínima.

2.2.2. Objetivos específicos

Teniendo en cuenta que la autora de este PFC ha estudiado Ingeniería Superior de Telecomunicaciones, plantear un proyecto de programación ha sido todo un reto, puesto que la experiencia en diseño y desarrollo de aplicaciones Java, así como en diseño de Bases de Datos relacionales era nula. Así, a cada uno de los objetivos que se mencionarán a continuación se tendría que sumar que la autora se tuvo que familiarizar en primer lugar con las plataformas.

Dejando a un lado los primeros objetivos fundamentales de la autora, para lograr las ambiciosas aspiraciones iniciales, se debían alcanzar cada uno de los siguientes objetivos:

- Enumerar todas las características funcionales del sistema.
- Especificar los requisitos de cada uno de los módulos funcionales.
- Diseñar el diagrama de la arquitectura necesario (tanto *hardware* como *software*) para satisfacer las funcionalidades anteriores.
- Ordenar por preferencia las funcionalidades definidas para decidir con qué prioridad implementarlas.
- Estudiar las particularidades de cada uno de los elementos necesarios:
  - Móvil:
    * Decidir a qué Sistemas Operativos móviles se les dará prioridad a la hora de encontrar una solución compatible: definir los indispensables y los complementarios.
    * Estudiar una solución para mantener el móvil operativo las 24h del día.
    * Analizar las opciones de conexión a Internet del móvil.
  - Comunicación entre ordenador y aplicación móvil:
    * Investigar las herramientas existentes.
    * Estudiar la compatibilidad de esas herramientas con el proyecto a desarrollar.
    * Decidir la tecnología a utilizar.
  - Agente:
    * Decidir con qué lenguaje de programación desarrollarlo.
    * Diseñar un diagrama de flujo de la aplicación en el que detallar todos los pasos que debe realizar.
    * Diseñar un diagrama de clases de la aplicación en el que detallar todas las clases y funciones necesarias.
    * Definir toda la información que el agente debe reportar.
    * Realizar un estudio de las librerías necesarias teniendo en cuenta las funcionalidades de la aplicación.

    * Diseñar y desarrollar la parte gráfica del agente en base a las necesidades de comunicación del usuario con el agente.
    * Integrar la tecnología de comunicación escogida entre ordenador y aplicación móvil.
    * Establecer un método de comunicación entre agente y *Web Service*.
    * Estudiar cómo controlar los posibles errores de la aplicación, tanto los de código como los de comunicación con el resto de elementos de la arquitectura.
  - *Web Service*:
    * Definir cómo comunicarse con el agente.
    * Definir cómo comunicarse con la Base de Datos.
    * Averiguar cómo desplegar la aplicación en un servidor.
  - Base de Datos:
    * Escoger el gestor de la Base de Datos.
    * Definir las tablas relacionales necesarias.
    * Definir los campos necesarios en cada tabla.
    * Definir qué campos serán auto-incrementales, así como cuáles tendrán la posibilidad de ser nulos y cuáles no.
    * Diseñar un diagrama de la Base de Datos en el que relacionar todos los elementos pensados.

Como se verá en el capítulo 3, el sistema completo de monitorización de aplicaciones móviles contiene más elementos de los anteriormente comentados. No obstante, tal y como se menciona en las secciones 3.2.2.1 y 3.3.2.1, la autora de este PFC no ha realizado todos ellos, de ahí que los objetivos se centren en las implementaciones de la autora.

2.3. Estructura de la memoria

Este documento se ha estructurado en 9 capítulos, tras los cuales hay una serie de apartados que complementan toda la información anterior. A continuación se razona la estructura de la memoria:

- Capítulo 1: Resumen.

  El primer capítulo de este documento resume en 3 idiomas, castellano, catalán e inglés, el objetivo de realizar este PFC, el trabajo desempeñado para conseguirlo y los resultados finales obtenidos.

- Capítulo 2: Introducción.

  En este capítulo se inicia realmente la explicación, ya que el capítulo 1 tan sólo resume toda la información que se detalla en el documento. Así, el capítulo 2 constituye la introducción de la memoria, razonando en primer lugar el porqué realizar este proyecto en la situación actual, para, a continuación, enumerar los objetivos propuestos para que el proyecto tenga sentido.

- Capítulo 3: Descripción global del sistema.

  En este capítulo se describe el sistema global a desarrollar. Así, el lector se puede hacer un esquema general de todo lo necesario en un sistema de monitorización de aplicaciones móviles, de forma que el seguimiento de los siguientes capítulos es más sencillo.

- Capítulo 4: Módulo de intercomunicación entre agente y móvil.

  Tras describir de forma general el sistema de monitorización, empiezan una serie de capítulos en los que se detallan algunos de los elementos de dicho sistema. El capítulo 4 concretamente se centra en la investigación que se hizo para encontrar una tecnología con la que comunicarse con aplicaciones móviles.

- Capítulo 5: Agente.

  Una vez se halló el método de comunicación con aplicaciones móviles, debía desarrollarse un agente que fuera el encargado de ejecutar circuitos continuamente sobre esas aplicaciones y, tras ello, enviar los resultados. Todo ese procedimiento se detalla en el capítulo 5.

- Capítulo 6: Web Service.

  Como intermediario entre los agentes y la Base de Datos se implementó un *Web Service*. En el capítulo 6 se detalla su desarrollo.

- Capítulo 7: Base de datos.

  Para almacenar toda la información relativa a la monitorización es necesario disponer de una Base de Datos. En el capítulo 7 se explica la lógica del diagrama de Base de Datos relacional que se diseñó.

- Capítulo 8: Evaluación de la solución.

  Para que el sistema de monitorización goce de alta disponibilidad, es necesario disponer de elementos que así lo faciliten. En el capítulo 8 se explican las medidas preventivas tomadas. Por otro lado, este capítulo de evaluación de la solución también describe la explotación de resultados que se lleva a cabo a partir de los datos obtenidos por los agentes.

- Capítulo 9: Conclusiones finales.

  Tras explicar todo el trabajo realizado en este proyecto, en el capítulo 9 se resalta el valor de la solución obtenida, así como los trabajos futuros que mejorarían el sistema.

- Anexos.

  En la sección de anexos se adjuntan documentos que complementan la información descrita en los anteriores capítulos. La mayoría de anexos se mencionan desde la propia memoria, no obstante, los anexos G, H, I y J son procedimientos que la autora ha realizado para que otras personas de NexTReT dispongan de documentación gestionar el servicio de MASM. Puesto que explican detalles que en la memoria no se mencionan, se ha considera interesante incluirlos como anexos.

Capítulo 3

Descripción global del sistema

3.1. Introducción

Como se ha comentado en el capítulo anterior, al inicio del presente proyecto el desconocimiento sobre cómo conseguir el sistema deseado era total, con lo que se pensó cómo diseñarlo a nivel global para después pensar en cómo realizar cada módulo individualmente. Esta primera fase fue la más importante del proyecto, ya que de ella dependía que la solución fuera escalable y fiable.

En primer lugar, se ideó un diagrama de flujo funcional en el que se detalló lo que debía realizar el sistema. A partir de éste, se diseñó un diagrama de la arquitectura de la solución con el que conseguir las funcionalidades descritas en el diagrama anterior. Para ello se debía tener en cuenta que la arquitectura debía ser escalable, es decir, capaz de albergar tantos terminales móviles y circuitos como fuera necesario.

En este capítulo se detallan los primeros pasos de este PFC, los que corresponden a las fases de análisis funcional y diseño técnico.

3.2. Descripción de las características funcionales

3.2.1. Objetivo

Al realizar este proyecto se buscaba una solución con la que medir la experiencia de usuario de aplicaciones móviles mediante transacciones sintéticas, emulando un usuario que continuamente estuviera usando la aplicación para comprobar su funcionamiento.

Para conseguir esta monitorización, el sistema debe realizar continuamente una serie de acciones predeterminadas en la aplicación móvil, comprobando si la respuesta obtenida es la esperada y, en caso de no serlo, enviar una alerta notificando el problema. Todos los resultados, además, se deben mostrar en una página web.

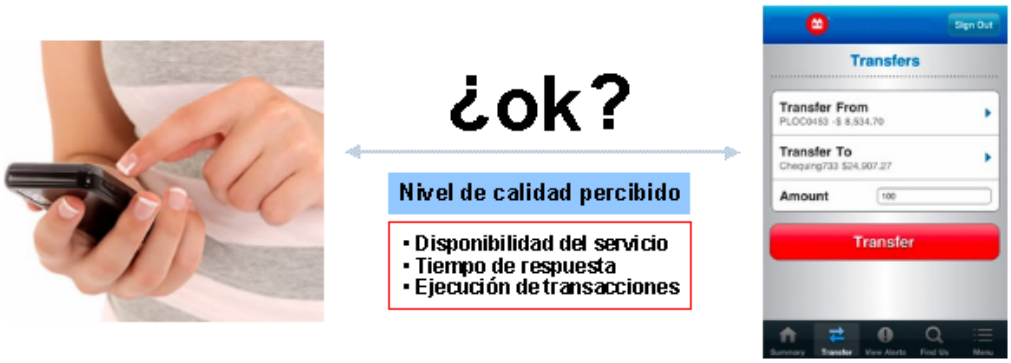


Figura 3.1: Realizar acciones sobre una aplicación móvil para comprobar su funcionamiento

De forma esquemática el sistema debería seguir el diagrama de flujo funcional que se muestra en la imagen 3.2.

3.2.2. Módulos funcionales

Desde un punto de vista funcional, la plataforma debería presentar una estructura de módulos y funcionalidades como la que se expone en la imagen 3.3, donde los cuadros con discontinuidad serían opcionales.

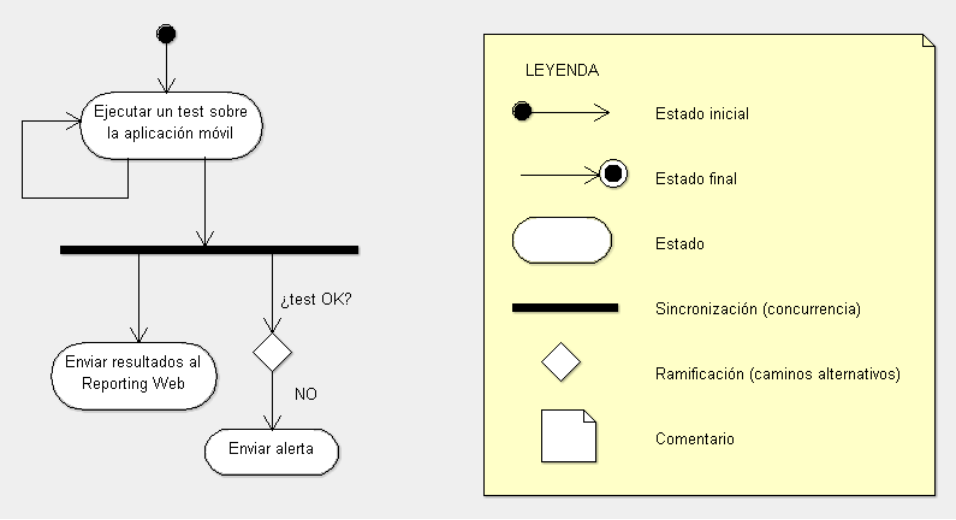


Figura 3.2: Diagrama de flujo funcional de MASM.

A continuación se detalla cada uno de ellos tal y como se pensaron al iniciar el proyecto y, posteriormente, se comentará cuáles se han realizado y cuáles han quedado para trabajos futuros.

Estación de monitorización

Este módulo es el corazón del sistema, gestiona a otros módulos permitiendo la reproducción de los tests y la captación de medidas. Ejecuta automáticamente la secuencia de todos los tests a partir del calendario definido en el *Scheduler* (ver imagen 3.3) y los repite cíclicamente, tomando mediciones y generando alarmas.

El sistema no requiere de ningún conocimiento del funcionamiento de la aplicación móvil más allá del que pueda tener un usuario, no es necesario conocer su código fuente ni ningún dato confidencial de la misma, tan sólo se requiere tener la aplicación instalada.

Durante la ejecución de un circuito se comprueba el resultado de un paso antes de pasar al siguiente. Para ejecutar un paso es necesario que el anterior se haya completado correctamente, en caso contrario, el test acaba donde ha fallado y captura la imagen del móvil en ese instante. Las capturas de error son una parte fundamental en la explotación de resultados.

Grabador de scripts

Es el encargado de la grabación y configuración de los tests. Dicha grabación se realiza interactuando manualmente con el móvil, de forma que los pasos que se hacen manualmente (clicar, seleccionar, desplazarse, etc.) se guardan automáticamente en forma de código. Este código se va creando en tiempo real a medida que se navega en la aplicación móvil y de forma transparente al usuario. Así, esta utilidad es de tipo *User Friendly* ya que permite crear nuevos circuitos sin necesidad de conocer la forma con la que MASM interactúa con el móvil, con lo que la grabación de tests es un paso trivial dentro del sistema de monitorización. Una vez creados, los circuitos se pueden modificar para adaptarlos a las necesidades del entorno cambiante.

Al crear un nuevo test el usuario, además de realizar las acciones en la aplicación móvil que definen dicho test, debe seleccionar el *robot*(o *robots*) en el que se ejecutará el test y el cliente al que corresponde. De esta forma, el grabador organiza los scripts creados en una estructura lógica a diferentes niveles de agrupación permitiendo que el módulo *Data Warehouse* obtenga esa información. La representación gráfica de esa estructura se muestra en la imagen 3.4.

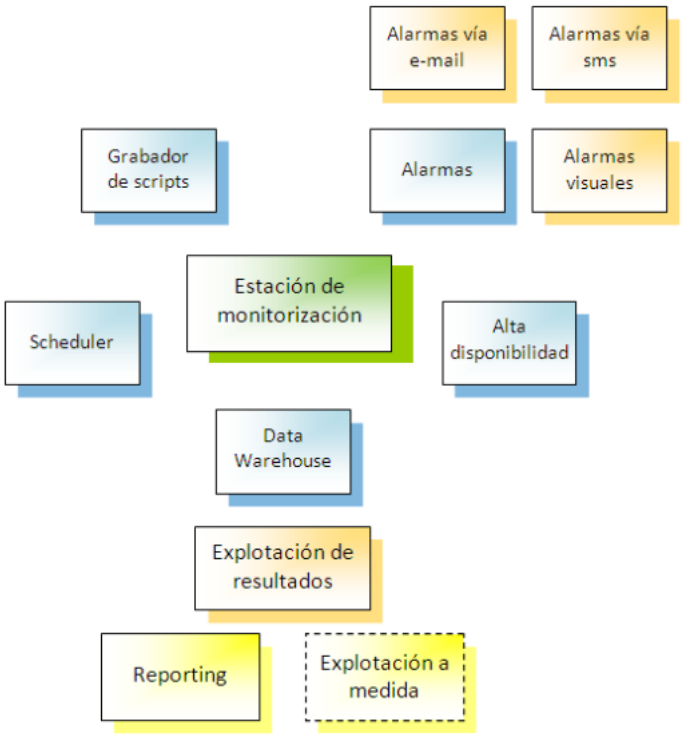


Figura 3.3: Módulos funcionales de MASM.

Scheduler

Permite seleccionar el calendario de ejecución de los tests estableciendo las franjas temporales de funcionamiento. De esta forma, siguiendo el orden de una lista de reproducción , el *scheduler* controla si el test está dentro o no del periodo de validación y, si lo está, empieza a ejecutarse, mientras que en caso contrario no se ejecuta y pasa a mirarse el siguiente test de la lista.

Por defecto se asigna que los tests se reproduzcan todos los días del año a todas horas y con la frecuencia máxima que permita la capacidad del robot considerando los tests que tenga definidos. De forma opcional se puede configurar un calendario personalizado: días concretos del año, del mes, de la semana, horas concretas del día, periodo mínimo entre ejecuciones, etc.

Alarmas

Alertan ante cualquier error producido en las transacciones de la aplicación móvil o al detectar una baja calidad de la aplicación según los niveles definidos en el SLA. Existen varios modos de enviar esa información:

- Mensajes emergentes.

  Se muestran en la estación de monitorización.

- Envío de e-mail o SMS.

  Este tipo de alarmas son configurables por el usuario, permitiéndole elegir el rango horario en el que desea que ser informado de las incidencias.

Data Warehouse

El módulo *Data Warehouse* permite guardar los datos en un volumen de información razonable agrupada en función del tiempo y tipo de test.

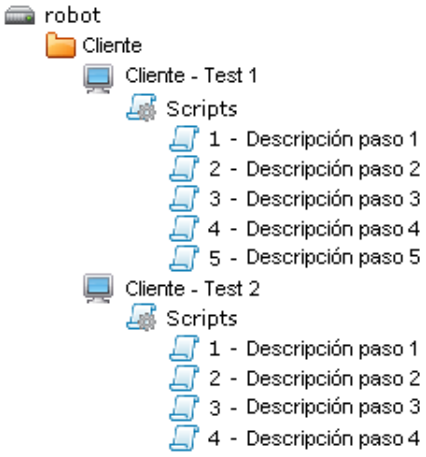


Figura 3.4: Estructura lógica con la que se organizan los tests de MASM.

Alta disponibilidad

Frente a una caída de la conexión al servidor de base de datos remoto, este módulo graba los resultados en una base de datos local y al recuperar la conexión los traspasa al servidor de base de datos. Con esto se consigue no perder resultados en circunstancias anómalas.

Reporting

El componente de *reporting online* es de utilidad tanto para el cliente como para NexTReT como proveedor del servicio de monitorización. La explotación de los datos se muestra en diversos formatos, de forma que la web tiene varios apartados principales:

- Panel de estado: Es lo primero que ve un usuario al acceder a la web. El panel de estado se trata de una lista de los tests contratados con unos indicadores visuales que resumen el resultado de los mismos en las últimas ejecuciones. La visualización es del estilo a la que se muestra en la imagen 3.5.
- Desde el propio panel de estado podrá accederse a un test en concreto haciendo clic sobre él. De esta forma obtenemos información más detallada sobre el mismo tanto en forma de gráfica como de tabla. Los datos se muestran en diversos formatos como gráficas, tablas, pdfs, etc. y se clasifican de distintas formas: tiempo visual, disponibilidad en muestras, disponibilidad en tiempo, lista de errores, capturas de errores, etc.
- Desde el Reporting se podrá consultar, administrar y definir las alarmas para todos los usuarios dados de alta en nuestra aplicación.
- Permitirá crear gráficos predefinidos con las consultas más usuales que se realicen, configurando los parámetros que sean de más utilidad para cada usuario.
- Incluirá una ayuda que facilitará el aprendizaje y la familiarización con la aplicación web.

3.2.2.1. Funcionalidades realizadas en el proyecto

Las funcionalidades mencionadas en la sección anterior son las que en su día se pensaron como deseables, pero de momento no todas ellas se han realizado. A continuación se detalla el estado actual de los módulos funcionales y cuáles de ellos los ha realizado la autora de este PFC:

- Estación de monitorización.

  Este módulo lo ha realizado íntegramente la autora de este PFC.

Test	Estado	Última Ejecución	Tiempo Visualización	Disponibilidad Última hora	Disponibilidad Hoy	Disponibilidad Mes
Cliente - Test 1	✔	19-07-2013 17:28:49	5.44	100%	100%	100%
Cliente - Test 2	✔	19-07-2013 17:29:47	18.90	100%	98.44%	98.85%
Cliente - Test 3	✔	19-07-2013 17:30:01	7.52	90.91%	97.37%	96.76%
Cliente - Test 4	✘	19-07-2013 17:16:15	-	0%	3.74%	2.33%

Figura 3.5: Panel de estado del Reporting web de MASM.

- Grabador de scripts.

  Este módulo no se ha realizado, queda pendiente como trabajo futuro.

- Scheduler

  Este módulo lo ha realizado íntegramente la autora de este PFC.

- Alarmas.

  Este módulo está hecho, pero no lo ha realizado la autora de este PFC.

- Data Warehouse.

  Este módulo lo ha realizado íntegramente la autora de este PFC.

- Alta disponibilidad.

  Este módulo no se ha realizado, queda pendiente como trabajo futuro.

- Reporting.

  Este módulo está hecho, pero no lo ha realizado la autora de este PFC.

3.3. Descripción de las características técnicas

3.3.1. Introducción

Para conseguir las funcionalidades deseadas se ha creado un software, al que llamamos agente, que es el encargado de ejecutar tests sobre dispositivos móviles. A partir de estas ejecuciones se recogen métricas de disponibilidad y tiempo de respuesta de la aplicación móvil desde el punto de vista del usuario. Estas métricas se envían a un *Web Service* para ser almacenadas en la Base de Datos. Mediante otro módulo se analizan los datos obtenidos y en caso de haberse producido algún problema en el uso de la aplicación, automáticamente se envía una alerta al responsable del servicio. Esta herramienta tiene, además, la capacidad de generar resúmenes automatizados e informes a medida, con lo que se puede consultar *online* el estado general del servicio a través de los últimos resultados disponibles.

3.3.2. Diagrama de la arquitectura

Para conseguir los objetivos mencionados se pensó en una arquitectura basada en el diagrama simplificado que se muestra en la imagen 3.6.

A continuación se mencionan brevemente algunos elementos de la arquitectura.

- Los terminales móviles, objeto de automatización, acceden a las aplicaciones en red tanto por WiFi como por 3G.

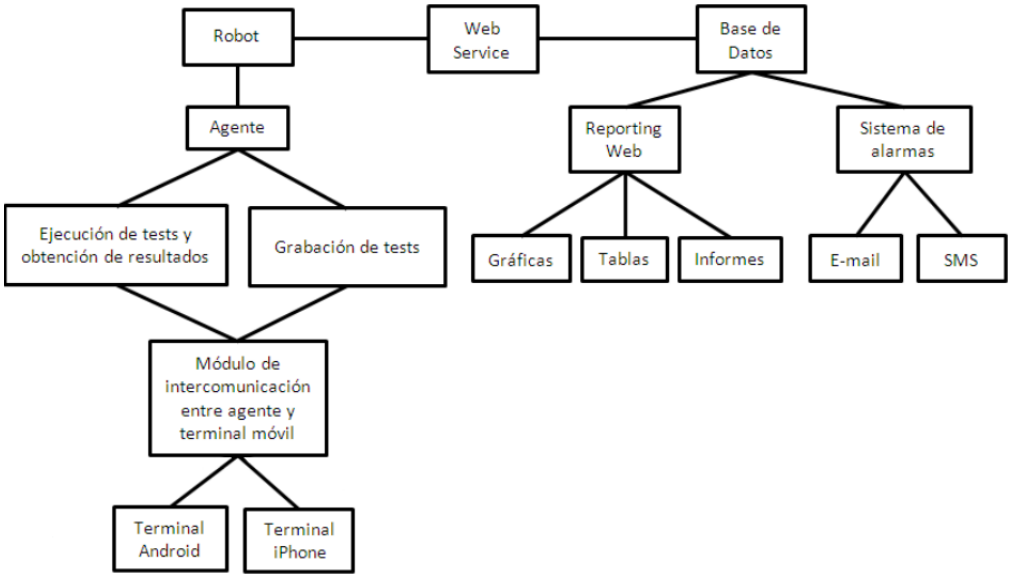


Figura 3.6: Diagrama de la arquitectura MASM.

- Se requiere de servidores físicos en los que se virtualiza un cierto número de *robots* en cada uno. En estos servidores se tienen conectados los dispositivos móviles objeto de las pruebas. Como se explica más adelante, el número de robots a desplegar en cada servidor puede verse limitado según el tipo de conexión por la que se opte entre el robot y el terminal móvil.

- Cada robot es un Sistema Operativo Windows virtual en el que se despliega un solo agente. En la fase de explotación de la herramienta se ve a estos robots *en la nube* y se accede a ellos remotamente.

- El *core* del sistema es una aplicación de escritorio desarrollada en Java J2SE llamada agente que actúa de modo sistemático ejecutando sobre terminales móviles diferentes circuitos previamente establecidos y guardados como scripts. En estos circuitos están definidos los pasos a realizar dentro de la aplicación móvil.

- La comunicación entre agente y terminal móvil puede realizarse por USB o por WiFi, mientras que la interacción con la aplicación móvil se hace por reconocimiento de imagen.

- La arquitectura también debe disponer de un *Web Service* y una *Base de Datos*. En cada ejecución de un test, el agente comprueba los resultados del mismo y recoge una serie de métricas con los tiempos de respuesta y/o los errores producidos. Estos datos se envían al *Web Service*, que es el encargado de recibir y tratar peticiones de la aplicación y realizar las tareas pertinentes contra la Base de Datos, donde se almacena la información relativa a los tests. El *Web Service* se ha desarrollado en Java J2EE, mientras que para el almacenamiento de datos se ha utilizado una BBDD SQL en la que se almacenan los datos con un *Stored Procedure* que se encarga de agrupar los resultados con temporalidad hora / día / semana / mes para su posterior explotación.

- Mediante otro módulo se analizan los datos obtenidos para, en caso de detectarse cualquier anomalía en la aplicación, enviar en tiempo real una alerta informando de la indisponibilidad. El sistema de alarmas es configurable y puede escogerse qué tipo de aviso se prefiere para recibir la alarma (correo electrónico, SMS, etc.). En la alerta se incorpora un texto en el que se detalla qué servicio concreto está fallando y con qué error, lo que facilita la recuperación del mismo. Este servicio de gestión y envío de alertas se ha desarrollado en Java.

- Otra parte fundamental del sistema completo de monitorización es la herramienta de *reporting web*, desarrollado en Java J2EE, que permite consultar *online* los datos obtenidos por el agente. Se desarrollará en Java J2EE.

3.3.2.1. Componentes realizados en el proyecto

En la sección anterior se ha visto los elementos de la arquitectura necesarios para realizar el sistema de monitorización. Obviamente se han realizado todos ellos ya que sino no habría monitorización, pero a continuación se detalla cuáles los ha realizado la autora de este PFC:

- Terminales móviles.

  La preparación de los terminales móviles (instalación del software necesario) la ha realizado íntegramente por la autora de este PFC.

- Módulo de intercomunicación entre agente y terminal móvil.

  Realizado íntegramente por la autora de este PFC.

- Agente.

  Realizado íntegramente por la autora de este PFC.

- Robot.

  La preparación de las máquinas virtuales (instalación del software necesario) la ha realizado íntegramente por la autora de este PFC.

- Web Service

  Realizado íntegramente por la autora de este PFC.

- Base de Datos.

  Realizado íntegramente por la autora de este PFC.

- Reporting.

  No lo ha realizado la autora de este PFC.

- Alarmas.

  No lo ha realizado la autora de este PFC.

3.4. Alcance del proyecto

El objetivo a alcanzar era obtener una solución de monitorización de aplicaciones móviles robusta y eficaz, centrándose en conseguir los requisitos funcionales ya mencionados. Para ello se llevaron a cabo una serie de tareas encaminadas a cumplir cumplir la totalidad de los requerimientos. Las actividades que se realizaron para el desarrollo e implantación del sistema se describen a continuación:

Análisis funcional

Durante la fase funcional se recogieron los requisitos exactos del sistema, incluyendo el contenido visual tanto del agente como del *Reporting Web*. Para ello, se realizaron reuniones con los compañeros de NexTReT para definir a partir de los requisitos generales, los detalles de cada funcionalidad.

Diseño técnico

Una vez definidos los requisitos, fue preciso realizar el diseño técnico del sistema concretando la solución técnica que se aplicaría para cada funcionalidad.

Investigación

Debido a las características de este PFC, fue necesario documentarse e investigar sobre cómo resolver los retos tecnológicos que suponía el sistema que se quería realizar.

Desarrollo

La fase de desarrollo constituyó las actividades necesarias para conseguir las funcionalidades detalladas en la fase de análisis, aplicando para ello el diseño técnico definido. Para el desarrollo del agente, *core* del sistema, se fueron incorporando nuevas funcionalidades de forma progresiva llevando a cabo las validaciones necesarias, de forma que en su finalización se tuviera la garantía de que se cumplían los objetivos funcionales definidos.

3.4.1. Actividades fuera del alcance del proyecto

Los procesos descritos anteriormente son aquellos que formaron parte de la creación del sistema de monitorización de aplicaciones móviles, es decir, los que se han realizado en este PFC.

Las actividades que se describen a continuación son las que NexTReT tiene previsto realizar a partir de tener las primeras ventas de este sistema. Se considera de interés para el lector comentarlas.

Despliegue en producción

El proceso de puesta en funcionamiento consistirá en la implantación en un entorno de producción del nuevo sistema con todas las funcionalidades desarrolladas. Durante el despliegue se contemplan como mínimo dos entornos diferentes:

- Un entorno de preproducción para cada cliente con datos reales de ejecuciones de un solo circuito. Se creará un usuario para que el cliente pueda acceder al *Reporting Online* y así visualizar los resultados en todos sus formatos.
- Un entorno de producción (independiente por cliente), real.

Ampliación de funcionalidades

Se prevé que el sistema esté en constante evolución. Como primer objetivo, se estudiará ampliar la versatilidad a un mayor número de terminales móviles. Esto es, obtener accesibilidad de interacción con Sistemas Operativos móviles no tan comunes como Windows Mobile y BlackBerry.

Capítulo 4

Módulo de intercomunicación entre agente y móvil

4.1. Introducción

Con la arquitectura de la solución ya pensada, destacaba un módulo especialmente crítico por ser el más innovador de este PFC, el que debía realizar la automatización de acciones sobre el terminal móvil. Así, la siguiente fase consistió en averiguar cómo resolverlo. Este capítulo se centra precisamente en exponer el resultado de ese estudio.

La fase de investigación de este proyecto fue una etapa complicada ya que se fijaron unos objetivos muy ideales que, a medida que el estudio avanzaba, parecían menos asequibles. Era importante no descartar ninguna opción a la ligera, puesto que de la decisión que se tomara en esta fase dependería tanto la fiabilidad como el mantenimiento de la herramienta. Es por ello que se dedicó un tiempo considerable a analizar las distintas posibilidades para estar seguros de escoger la más adecuada.

4.2. Fase de investigación

4.2.1. Objetivos

El propósito de la fase de investigación era encontrar una forma de realizar acciones sobre aplicaciones móviles cumpliendo los requisitos que se citan a continuación:

- Indispensables:
  - Tecnología compatible con terminales iOS.
    Debido a una estrategia comercial de NexTReT, desde un primer momento el proyecto se enfocó en conseguir una solución para terminales iOS.
  - Factible sin conocer el código de la aplicación móvil.
    Este proyecto se inició con la finalidad de obtener un sistema capaz de monitorizar aplicaciones móviles de terceros, con lo que no se tendría acceso al código de la aplicación.
- Deseables:
  - Tecnología compatible tanto con terminales iOS como Android.
    Si bien es cierto que la búsqueda debía centrarse en encontrar una solución para terminales iOS, también se estudiaron las opciones para Android, ya que a corto plazo la herramienta se expandiría a Android y la situación ideal sería encontrar una solución compatible con ambos Sistemas Operativos.
  - Factible sin realizar ninguna acción en el terminal móvil.
    Puesto que de conseguirlo sería un elemento diferenciador frente a soluciones existentes, inicialmente se buscaba una tecnología con la que no hiciera falta modificar el terminal móvil, es decir, que pudiera ejecutarse en terminales venidos de fábrica, sin necesitar acceso *root* ni realizar *jailbreak*.
  - Interacción con la aplicación móvil por objeto.
    Inicialmente se prefería una tecnología basada en objetos, frente a la detección por imagen, puesto que los objetos siempre son accesibles aunque no se vean visualmente en la pantalla del dispositivo.
  - Sencilla creación y modificación de tests.
    Pensando en el futuro mantenimiento de la herramienta, sería conveniente encontrar una forma con la que realizar tests de forma sencilla, es decir, que cualquier persona ajena al proyecto y sin conocimientos técnicos pudiera hacerlos siguiendo un pequeño manual.

4.2.2. Desarrollo de la búsqueda

En la investigación concretamente se buscaba una API, un *framework* o una *librería* que sirviera de capa de abstracción entre el terminal móvil y el Java. A continuación se presentarán las herramientas más relevantes que se encontraron, se compararán y se expondrán las razones de la elección de una de ellas.

4.2.2.1. UI Automation

Instruments es una aplicación de escritorio para Mac OS X que proporciona *Apple* como parte de *Xcode* (IDE de desarrollo de aplicaciones iOS). *Instruments* es una herramienta de rendimiento, análisis y *testing* de código OS X e iOS. En la versión de *Xcode 4* se añadió una nueva funcionalidad a *Instruments* llamada *Automation* que permite realizar *testing* GUI en simuladores y dispositivos iOS. Con la incorporación de esta herramienta, se pueden automatizar casos de prueba escritos en *JavaScript* utilizando las APIs de *UI Automation* que proporciona *Apple*.

Automation viene con un editor de código y dispone de un grabador que facilita la tarea de *scripting*. La interfaz gráfica es tal como se muestra en la imagen 4.1.

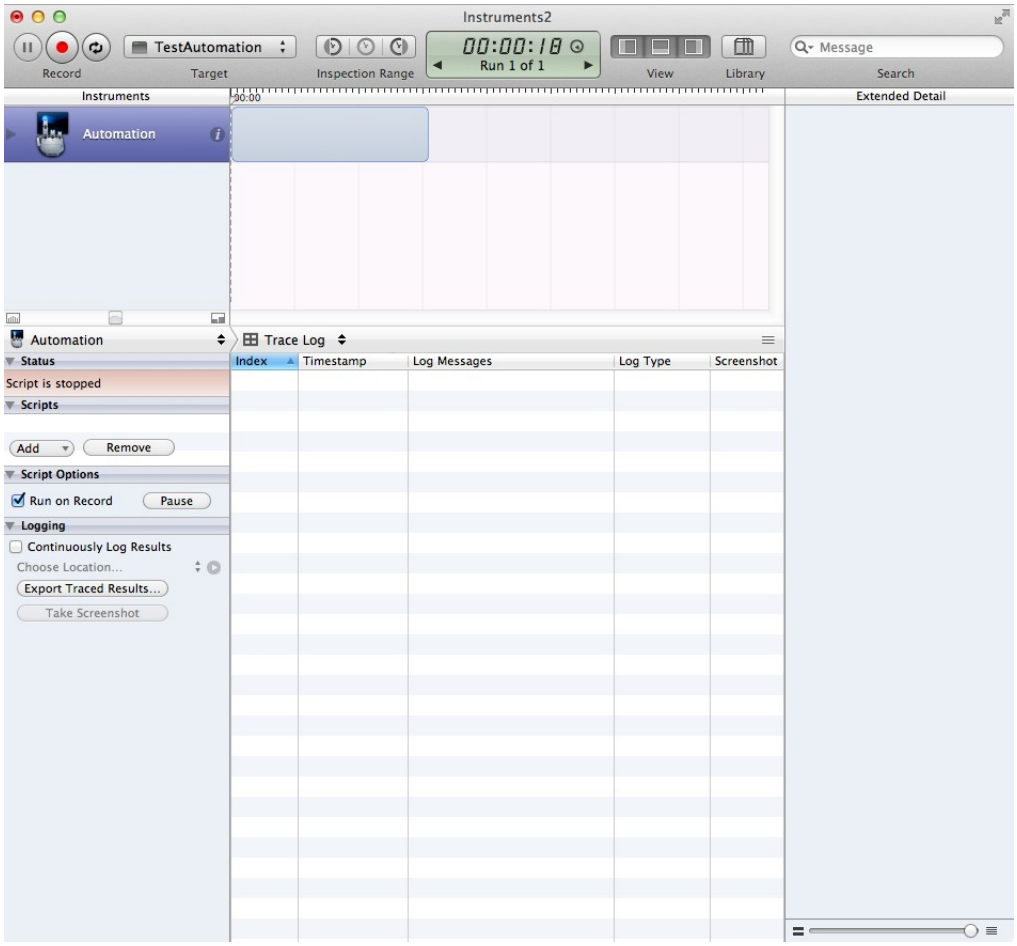


Figura 4.1: UI Automation: Interfaz gráfica de *Automation*.

Análisis de los *scripts*

Los *scripts* de esta herramienta se escriben utilizando las clases *JavaScript* que proporciona Apple y cuya información puede encontrarse en la referencia bibliográfica [1]. Basándose en la

presencia de ciertos elementos UI o su valor (*id* del objeto), estos *scripts* se pueden usar para comprobar si la aplicación se está comportando de acuerdo a las expectativas.

El primer paso es tener una referencia de la aplicación a la que se quiere acceder. Para ello deben añadirse las siguientes líneas de código al inicio del test:

target = UIATarget.localTarget();
application = target.frontMostApp();

En la primera línea, *UIATarget* es la vista principal de la aplicación móvil que se ejecuta en el dispositivo o el simulador. Actúa como una especie de *proxy*, es el objeto con el que se interactúa para realizar acciones sobre el terminal como jugar con los controles de volumen, mover el dispositivo o realizar gestos de usuario. En la segunda línea, el objeto *application* da acceso a la estructura de alto nivel de la aplicación, a cosas como la barra de navegación, las pestañas y la ventana principal.

En la imagen 4.2 puede verse un *script* GUI muy simple que genera eventos aleatorios en una aplicación de iPad. Básicamente simula tocar distintas posiciones de la pantalla del terminal, donde las posiciones vienen determinadas por una función matemática que genera números aleatorios.

```
// #### Random Event Generator for iPad####
var width = 1024;
var height = 768;
varmTarget = UIATarget.localTarget();

// tap randomly on screen
vartapTest = function()
{
varyPos = Math.floor(Math.random()*width +1);
varxPos = Math.floor(Math.random()*height +1);

varimageRect = mTarget.frontMostApp().mainWindow().rect();
if(xPos> (imageRect.origin.x + imageRect.size.width))
xPos = imageRect.size.width/2;
if(yPos> (imageRect.origin.y + imageRect.size.height))
yPos = imageRect.size.height/2;

try{
mTarget.tap({x:xPos,y:yPos});
mTarget.delay(1.0); // waiting for 1 second
    }
catch(error){
UIALogger.logError(error);
mTarget.logElementTree();
UIALogger.logFail("Tapping failed @ x: " + xPos + ", y: "+yPos);
    }
mTarget.logElementTree(); //logs elements of the current Tree {mTarget}
}

tapTest();
```

Figura 4.2: UI Automation: Ejemplo de *script* que genera eventos aleatorios.

En el ejemplo anterior se simula *tocar* posiciones de pantalla (*mTarget.tap(x:xPos,y:yPos)*), pero también se puede interactuar con objetos concretos si la aplicación tiene botones u otros elementos. Para ello, se obtiene el elemento y entonces se interactúa con él. Por ejemplo, el código de la imagen 4.3 *clicaría* sobre el elemento llamado "Equation library", accesible en la vista "rootView" de la aplicación.

Con esta herramienta se pueden simular una variedad de movimientos y gestos, por ejemplo:

- tap()
- tapAndHold(Number duration)

```
// #### Getting Element ####

rootView.buttons()["Equation library"].tap();
```

Figura 4.3: UI Automation: Ejemplo de línea de *script* que obtiene los elementos de una aplicación.

- doubleTap()
- twoFingerTap()
- touchWithOptions(Object options)
- dragInsideWithOptions(Object options)
- flickInsideWithOptions(Object options)
- scrollToVisible()

Automatización de *scripts*

Esta herramienta puede servir de capa de abstracción de forma sencilla, ya que los *scripts* pueden ejecutarse desde la línea de comandos. La nomenclatura del comando es la siguiente:

instruments -*w* numeroSerieTerminalReal -*t* rutaAutomationTemplate rutaAplicación -*e UIASCRIPT* rutaScript -*e UIARESULTSPATH* rutaResultados

Donde:

- *instruments*: comando que hace referencia al programa *Instruments*.
- -*w*: Con este argumento se especifica que el *script* se va a ejecutar sobre un dispositivo real. A continuación, separado por un espacio en blanco, se escribe el número de serie del terminal.
- -*t*: Con este argumento se especifica la funcionalidad de *Instruments* que se va a utilizar, en este caso *Automation*. A continuación, separado por un espacio en blanco, se escribe la ruta donde se encuentra el archivo "Automation.tracetemplate" en el Mac.
- En el siguiente argumento se especifica la ruta de la aplicación dentro del terminal móvil.
- -*e UIASCRIPT*: Con este argumento se especifica la ruta de ordenador donde está el *script* a ejecutar.
- -*e UIARESULTSPATH*: Con este argumento se especifica la ruta de ordenador donde dejar los *logs* del *script*.

Para comprobar si esta herramienta era útil para el proyecto, se disponía de un Mac OS X 10.7 y un iPad 3 para realizar las pruebas. Se encontraron las correspondientes rutas y se lanzó el comando. La ruta de Mac donde se encuentra el archivo "Automation.tracetemplate" depende de la versión de *Xcode* instalada, mientras que para encontrar la ruta en la que se encontraban las aplicaciones del iPad, se instaló un programa que permitía explorar el contenido del terminal. En este caso pretendía ejecutarse un *script* sobre la aplicación de *La Caixa* y las rutas eran las siguientes:

- rutaAutomationTemplate: */Applications/Xcode.app/Contents/Applications/Instruments.app/Contents/PlugIns/AutomationInstrument.bundle/Contents/Resources/Automation.tracetemplate*
- rutaAplicación: */laCaixa.app/laCaixa*

Al ejecutar el comando se consiguió abrir la aplicación, pero no se realizaba ninguna de las acciones que se habían escrito en el *script*. De hecho, si se ejecutaba el comando sin el parámetro *-e UIASCRIPT*, es decir, sin especificar el *script* que se quería ejecutar, ocurría exactamente lo mismo, se abría la aplicación y nada más. Viendo esto, parecía que por algún motivo el *script* no se comunicaba con la aplicación móvil.

Dicho *script* se había escrito manualmente, con lo que otra prueba que se hizo fue intentar crearlo con el grabador del que dispone *Automation*. Con este grabador, si todo va bien, mientras se realizan acciones manualmente sobre la aplicación móvil, en tiempo real se van creando líneas de código *JavaScript* que equivalen a los gestos que se hacen. Esto, en cambio, no ocurría en nuestro caso. De nuevo se veía un problema de comunicación con la aplicación móvil.

Se hizo una última prueba que verificó las sospechas anteriores. Se realizó una pequeña aplicación iOS con *Xcode*, se compiló, se instaló en el iPad y se comprobó que sobre esa aplicación no había ningún problema. El grabador realizaba su función correctamente y los *scripts* que se *lanzaban* se ejecutaban con normalidad.

Así, tras el estudio realizado, se llegó a la conclusión de que la herramienta *UI Automation* sólo es útil en el caso de disponer del código de la aplicación móvil. Se investigó al respecto y se vio el porqué: Apple hace firmar las aplicaciones iOS con un certificado y, para que los *scripts* de *UI Automation* se puedan comunicar con una aplicación, se debe ejecutar un *build* especial que *linque* el certificado del que se dispone con el certificado con el que creó la aplicación.

Pros y Contras de la herramienta

- Pros:
  - El *JavaScript* es sencillo de aprender.
  - Aplicable a simulador y terminal real.
  - Soporta todo tipo de gestos.
  - Tiene soporte de la comunidad de Apple.
  - Permite ejecutar comandos nativos como el comando de *shell ImageMagick* así como otros *scripts* para comparar imágenes, con lo que puede combinarse con detección de imagen.
- Contras:
  - Sólo disponible en Mac OS X.
  - Sólo disponible para aplicaciones iOS.
  - Para ser utilizado en un terminal iOS físico se necesita tener el código de la aplicación.

Compatibilidad con el PFC

Esta herramienta es muy potente, pero no se puede usar si el código de la aplicación es de un tercero, lo cual es un requisito indispensable para este proyecto. Por tanto, se descarta.

4.2.2.2. Monkey

Monkey es una herramienta de línea de comandos disponible en el SDK de Android capaz de generar movimientos de usuario pseudo-aleatorios tanto en emuladores como en terminales Android. Incluye bastantes opciones que se pueden clasificar como sigue:

- Opciones de configuración básica, como determinar el número de eventos a realizar.
- Limitaciones operacionales, como restringir el test a un sólo *package*, es decir, a una sola aplicación móvil.

- Tipos de eventos y frecuencia de los mismos.
- Opciones de depuración.

Cuando se ejecuta *Monkey*, genera y envía eventos al sistema. También *observa* el sistema sobre el que está ejecutándose y busca tres condiciones:

- Si *Monkey* se limita a uno o varios *packages* previamente fijados, vigila si hay intentos de navegar a otros *packages* y los bloquea.
- Si la aplicación sobre la que se ejecuta *Monkey* se *cuelga* o recibe algún tipo de excepción no controlada, *Monkey* para y reporta el error.

Uso básico de la herramienta

Como se ha comentado, es una herramienta que se ejecuta desde la línea de comandos. La sintaxis básica es la siguiente:

adb shell monkey [opciones] <NúmeroEventos>

Si no se especifica ninguna opción, *Monkey* envía eventos a cualquier (y a todos) los *packages* instalados en el dispositivo/emulador. A continuación se muestra un ejemplo típico con el que se abrirá una aplicación concreta y sobre ella se ejecutarán 500 eventos aleatorios:

adb shell monkey -p nombre.de.package -v 500

Puede encontrarse más información sobre la herramienta en la referencia bibliográfica [2].

Pros y Contras de la herramienta

- Pros:
  - Sencillez del uso.
- Contras:
  - Sólo útil para Android.
  - Sólo genera eventos aleatorios, no pueden especificarse cuáles.

Compatibilidad con el PFC

Esta herramienta suele utilizarse para realizar pruebas de estrés a aplicaciones móviles en fase de desarrollo. En ese ámbito reside su utilidad, pero no sirve para el objetivo del proyecto, con lo que queda descartada.

4.2.2.3. MonkeyRunner

El SDK de Android consiste en una serie de herramientas para desarrollar y realizar pruebas sobre aplicaciones móviles Android. Una de estas herramientas es *MonkeyRunner*, capaz de realizar acciones UI programadas sobre terminales Android sin necesidad de acceso *root*.

El Sistema Operativo móvil Android dispone de una aplicación pre-instalada llamada *Monkey*. *MonkeyRunner* se ejecuta desde el ordenador, crea una conexión *socket* con *Monkey* con la ayuda de Android Debug Bridge (ADB) y envía comandos a *Monkey*, quien es el que realmente ejecuta las acciones en el dispositivo.

Análisis de la herramienta

La herramienta *MonkeyRunner* puede usarse con *Python* o con *Java*. Durante la investigación principalmente se estudió su versión en *Python*, pero son equivalentes.

MonkeyRunner con *Python*

Para ejecutar un *script* de *MonkeyRunner*, tan sólo debe enviarse la siguiente línea de comandos:

monkeyrunner nombreScript.py

A continuación se comentarán las líneas de código más importantes de estos *scripts* y el estudio realizado con respecto a esta herramienta.

En primer lugar es necesario definir el *target* del *script*, es decir, sobre qué terminal se va a ejecutar y concretamente en qué aplicación. Para ello se escriben las siguientes líneas de código:

```
device = MonkeyRunner.waitForConnection()
package = nombrePackageAplicación
activity = nombreActivityAplicación
runComponent = package + '/' + activity
device.startActivity(runComponent)
```

La primera línea sirve para conectarse con el terminal Android enchufado por USB al ordenador (más adelante se hará un inciso al respecto) y las siguientes líneas sirven para seleccionar la aplicación con la que se quiere interactuar. Para ello se debe escribir el nombre del *package* y de una *Activity* de la aplicación. Es importante mencionar que las aplicaciones Android tienen un único *package*, pero pueden tener varias *Activity*s, es decir, varias ventanas. Para conocer esos nombres, existe una aplicación Android llamada *Package Manager Ad* con la que se puede ver el *package* y las *Activity*s de las aplicaciones instaladas en el móvil. Durante el estudio se realizaron pruebas sobre la aplicación de *La Caixa*, así que las las líneas de código quedaron de esta forma:

```
device = MonkeyRunner.waitForConnection()
package = 'es.lacaixa.mobile.android.newwapicon'
activity = 'es.lacaixa.mobile.android.newwapicon.SplashActivity'
runComponent = package + '/' + activity
device.startActivity(runComponent)
```

Cabe destacar que en la primera línea se asume que sólo hay un terminal conectado. En caso de tener varios, podría utilizarse el comando *adb devices* disponible en el SDK de Android para escoger uno de ellos. Un ejemplo de cómo hacerlo sería el código que se muestra en la imagen 4.4, en el que se escoge el primer terminal que aparece en la lista. Obviamente esto se podría personalizar y escoger el terminal, por ejemplo, por el número de serie.

Una vez definido el *target* del *script*, ya se puede comenzar a interactuar con él. Algo común es simular la acción de *clicar* sobre un objeto de la aplicación móvil. Para ello habría dos formas de hacerlo:

- Enviando las coordenadas (x,y) de la pantalla del móvil, por ejemplo:
  device.touch(225, 182, MonkeyDevice.DOWN_AND_UP)
- Especificando el tipo de objeto, por ejemplo:
  device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)

El *string MonkeyDevice.DOWN_AND_UP* es el tipo de acción que se va a realizar, mientras que el *string 'KEYCODE_MENU'* del segundo caso es el tipo de objeto con el que se va a interactuar.

```
# Obtener los terminales conectados

devices = \
commands.getoutput('adb devices').strip().split('\n')[1:]

if len(devices) == 0:
    print "Ningún terminal conectado"
    sys.exit(1)

# Conectarse al primero de la lista

Actual_device = devices[0].split('\t')[0]
device = MonkeyRunner.waitForConnection(5, Actual_device)
```

Figura 4.4: MonkeyRunner: Cómo escoger uno de los móviles conectados por USB.

Al contrario de lo que ocurría con *UI Automation*, con *MonkeyRunner* sí podemos comunicarnos con los objetos de la aplicación a pesar de no tener el código de la misma. Para ello se debe conocer el tipo de objeto con el que se quiere interactuar, pero eso no es trivial. A priori el tipo de objeto se conoce realizando *prueba y error*, con el tiempo que ello conlleva, sobretodo teniendo en cuenta todos los tipos de objeto que existen (en la referencia bibliográfica [3] pueden verse todos ellos). Es evidente que este camino no es viable, con lo que se buscaron otras formas de encontrar los objetos.

1. Mediante *uiautomatorviewer*.

   El SDK de Android dispone de una herramienta gráfica llamada *uiautomatorviewer* con la que se puede *escanear* y analizar los componentes UI de una aplicación Android. En la imagen 4.5 se muestra el resultado de la prueba que se realizó de esta herramienta sobre la aplicación de *La Caixa*. Como puede verse, este programa tiene dos partes diferenciadas: la vista de la pantalla del dispositivo y los elementos UI hallados en esa vista. Si se selecciona uno de los elementos (parte derecha de la imagen), se resalta dicho elemento con un marco rojo (parte izquierda de la imagen).

   Como se aprecia en la imagen, en este caso la aplicación tiene una *WebView* (elemento resaltado en la imagen) que *uiautomatorviewer* ve *como un todo*. Con esto pudo comprobarse que *uiautomatorviewer* es incapaz de discernir entre los elementos dentro de *WebViews*. Con lo cual, puesto que esta herramienta no puede diferenciar todos los objetos que pueden tener las aplicaciones, quedó descartada como utilidad para el proyecto.

2. Mediante *AndroidViewClient*.

   Existe una extensión de *MonkeyRunner* llamada *AndroidViewClient* que le proporciona una visión de más alto nivel para que el *scripting* sea más sencillo. Se descargó la extensión y se realizaron pruebas.

   En dicha extensión se encontró un *script* interesante llamado *dump.py* (Anexo A) que detecta los elementos UI de la pantalla del móvil y les asigna un nombre a cada uno (*'id/no_id/1'*,*'id/no_id/2'*,*'id/no_id/3'*, etc.). Con este *script* se abría una posibilidad: a pesar de no tener el código de una aplicación, se podría ejecutar este *script*, *parsear* su salida para conocer el nombre que le ha asignado a un objeto X que nos interesara y así interactuar con dicho objeto a partir del nombre asignado.

   Para comprobar si realmente el *script* era de utilidad, se *lanzó* con la pantalla del móvil en el mismo punto en el que se ve en la imagen 4.5. El resultado fue el que se muestra en la imagen 4.6.

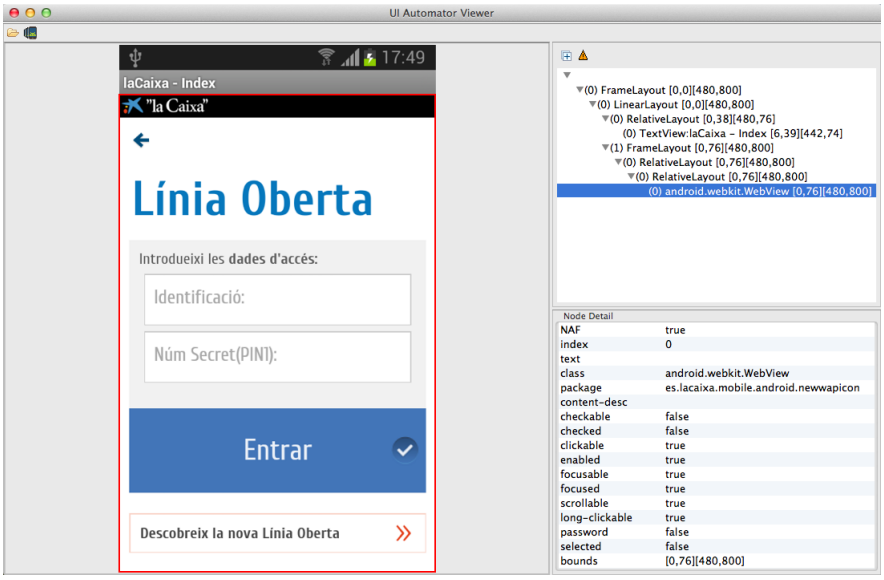


Figura 4.5: MonkeyRunner: Interfaz de Usuario capturada por *uiautomatorviewer*.

```
MacBook-Pro-de-Irene:examples Garcia$ ./dump.py
android.widget.FrameLayout id/no_id/1
   android.widget.LinearLayout id/no_id/2
      android.widget.RelativeLayout id/no_id/3
         android.widget.TextView id/no_id/4 laCaixa - Index
      android.widget.FrameLayout id/no_id/5
         android.widget.RelativeLayout id/no_id/6
            android.widget.RelativeLayout id/no_id/7
               android.webkit.WebView id/no_id/8
```

Figura 4.6: MonkeyRunner: Resultado ejecución *script dump.py*.

Comparando las imágenes 4.5 y 4.6 se llega a la conclusión de que el *script dump.py* extrae exactamente los mismos elementos que la herramienta *uiautomatorviewer*. Esto es debido a que, como puede verse en la línea 127 del anexo A, el *script dump.py* utiliza la función *dump()* de la clase *ViewClient* que, como puede verse en la línea 29 del anexo B, a su vez utiliza el comando *uiautomator dump* del SDK de Android. Dicho comando es el mismo sobre el que se sustenta la herramienta *uiautomatorviewer* y por eso obtenemos el mismo resultado.

En conclusión, el *script dump.py* también tuvo que descartarse puesto que tenía el mismo problema con las *WebViews* que *uiautomatorviewer*.

3. Mediante *EasyDevice*.

Se encontró otra extensión interesante de *MonkeyRunner*, en este caso una *librería* llamada *EasyDevice* que sirve para interactuar con los objetos de la aplicación mediante su *id*. A continuación se muestra un ejemplo del uso de esta *librería*:

device = MonkeyRunner.waitForConnection()
easy_device = EasyMonkeyDevice(device)
easy_device.touch(By.id('id/nameId'),MonkeyDevice.DOWN_AND_UP)

Con el uso de esta *librería*, una vez conocidos (o asignados) los nombres de los objetos de la aplicación móvil, ya se puede interactuar con ella. No obstante, los nombres no se conocen porque no se dispone del código de la aplicación y, la asignación de éstos

se ha visto anteriormente que no está resuelta, con lo que sólo por esto ya podría descartarse esta opción. A ello se suma el hecho de que *EasyDevice* sólo puede ejecutarse sobre terminales de desarrollo, con lo que para un móvil comercial no sirve.

En conclusión, no se encontró ninguna forma de conseguir que la creación de *scripts* se pudiera hacer de forma rápida, sencilla y/o automatizada. Se encontraron *atajos* interesantes para detectar los objetos, pero ninguno de ellos aplicaba totalmente. Así, la única vía con la que poder interactuar con todos ellos era la inicial, *prueba y error* con el tipo de objeto.

MonkeyRunner con *Java*

MonkeyRunner también puede ejecutarse desde *Java*, no obstante, el estudio es equivalente al de *Python*, así que simplemente se comenta como curiosidad.

El SDK de Android proporciona archivos *.jar* que pueden incluirse en proyectos *Java*. Son los siguientes:

- ddmlib.jar
- ddms.jar
- jython.jar
- monkeyrunner.jar
- sdklib.jar

En la imagen 4.7 se muestra un ejemplo del código que se utilizaría en *Java*. La primera línea de código inicializa el ADB y las siguientes líneas se entienden por sí solas.

```
AdbBackend adb = new AdbBackend();
AndroidDebugBridge b = AndroidDebugBridge.getBridge();

DdmPreferences.setTimeOut(300000);

IDevice[] devices = b.getDevices();
AdbMonkeyDevice md = new AdbMonkeyDevice(devices[0]);
MonkeyManager mgr = md.getManager();

boolean res = false;
try
{
    md.getManager().press("KEYCODE_VOLUME_UP");
    md.getManager().press("KEYCODE_VOLUME_DOWN");
    res = md.getManager().touch(10, 20);
}
catch(Exception e)
{
    e.printStackTrace();
}
```

Figura 4.7: MonkeyRunner: Ejemplo de test escrito en *Java*.

Pros y Contras de la herramienta

- Pros:
  - No es necesario conocer el código de la aplicación.
  - Muchas de las funciones se pueden utilizar en terminales comerciales.
  - Capaz de realizar *tests* en varios dispositivos a la vez.
  - Aplicable a simulador y terminal real.
  - El lenguaje es sencillo de aprender.
  - Existe mucha documentación en Internet.
- Contras:
  - Sólo aplicable a terminales Android.
  - Los *scripts* son muy personalizados y conlleva mucho tiempo realizarlos.

Compatibilidad con el PFC

Esta herramienta puede parecer interesante puesto que aporta una comunicación *por objeto* con las aplicaciones aún sin tener acceso al código de las mismas, pero es importante pensar en el mantenimiento del sistema completo que se va a realizar en este proyecto. No es admisible que la realización de los *tests* sea tediosa. Este proceso debe ser lo más sencillo y rápido posible, ya que sino se perdería mucho tiempo para cada *test* que solicitara un cliente, lo cual no sería nada conveniente para NexTReT ni para el empleado que tenga esa tarea. Es por ello que *MonkeyRunner* se descarta en ese investigación.

4.2.2.4. MonkeyTalk / FoneMonkey

MonkeyTalk, anteriormente conocido como *FoneMonkey*, es una herramienta *open source* diseñada para ayudar a los desarrolladores y a los controles de calidad de aplicaciones móviles iOS y Android.

La plataforma consiste en dos componentes principales: el IDE de *MonkeyTalk* y los *agentes* de *MonkeyTalk*. El IDE es una herramienta basada en Eclipse con la que grabar, editar, ejecutar y administrar *tests*. En la referencia bibliográfica [4] se explica como configurar *Eclipse* para poder utilizarlo como IDE de *MonkeyTalk*. Por otro lado, los denominados *agentes* son *librerías* para iOS y Android que deben conectarse con la aplicación móvil.

Como puede verse en la documentación facilitada en [5], esta herramienta lo que hace es utilizar como capa de abstracción *UI Automation* en el caso de iOS y *MonkeyRunner* en el caso de Android, para unificar ambos Sistemas Operativos en un único lenguaje de *testing*.

Compatibilidad con el PFC

Puesto que *MonkeyTalk* se basa en *UI Automation* y en *MonkeyRunner*, si se descartaron estas opciones anteriormente, *MonkeyTalk* también queda descartada.

4.2.2.5. Frank

Frank es una herramienta *open source* para realizar *testing UI* de aplicaciones iOS. *Frank* utiliza *Cucumber* [6] y *JSON*, a través de los cuales se pueden escribir *tests* y ejecutarlos contra una aplicación iOS. *Frank* también incluye un *inspector de aplicaciones* llamado *Symbiote* con el que se puede obtener información detallada de la aplicación que se está ejecutando.

Los *scripts* de *Frank* son muy sencillos de realizar. En la imagen 4.8 se muestra un ejemplo en el que se navega entra las distintas ventanas de una aplicación.

```
Scenario: Moving from the 'Home' screen to the 'Events' screen
Given I launch the app.
Then I should be on the Home screen
When I navigate to "Events"
Then I should be on the Events screen
```

Figura 4.8: Frank: Ejemplo de *script*.

Pros y Contras de la herramienta

- Pros:
  - Viene con *Symbiote*, con lo que no sólo puede realizar *testing UI* sino que puede obtener información detallada de la aplicación en ejecución.
  - Puede grabar un vídeo mientras realiza el *test*.
  - Aplicable tanto a simulador como a terminal físico.
  - Tiene un lenguaje muy sencillo.
- Contras:
  - Escasa posibilidad de gestos.
  - Para utilizarse en un terminal iOS físico es necesario tener el código de la aplicación móvil.

Compatibilidad con el PFC

Dado que requiere del código de la aplicación iOS, se descarta esta herramienta.

4.2.2.6. Appium

Appium es un *framework open source* que realiza *testing* de aplicaciones iOS. Básicamente lo que hace es utilizar *UI Automation* como capa de abstracción para después escribir los *tests* con comandos propios. No aporta nada distinto a *UI Automation*, con lo que quedó descartado.

4.2.2.7. Native Driver

Native Driver ha derivado del proyecto *Appium*. Es un *framework* que engloba tanto iOS como Android, con lo que se puede interactuar con ambos Sistemas Operativos con un mismo lenguaje, pero dado que proviene de *Appium*, la parte iOS tiene el problema de necesitar el código de la aplicación. En conclusión, también se descarta esta solución.

4.2.2.8. Robotium

Robotium es un *framework open source* para automatizar *tests* para aplicaciones Android. Se puede usar tanto en aplicaciones de las que se disponga el código como en aplicaciones de las que sólo se tenga el *.apk* y los detalles de la implantación se desconozcan.

Se estudiaron las posibilidades de esta herramienta y se descartó ya que sólo servía para Android y no aportaba nada distinto que no se hubiera visto con *MonkeyRunner*.

4.2.2.9. UISpec

UISpec es un *framework* Behavior Driven Development (BDD) para iOS que proporciona una solución para automatizar *testing UI* del cual se puede encontrar más información en la referencia bibliográfica [7]. En lo que se refiere a la investigación, se descartó por requerir el código de la aplicación móvil.

4.2.2.10. Keep it Functional (KIF)

KIF es un *framework open source* desarrollado por la compañía *square*. Permite desarrollar *tests* en *Objective C* sobre aplicaciones iOS en un simulador iPhone o iPad. Para ello se integran los *tests* en el IDE de desarrollo *Xcode* y se ejecutan desde ahí.

Esta herramienta se descartó puesto que sólo es útil en simuladores iOS, además, requiere conocer la estructura del código de la aplicación.

4.2.2.11. Sikuli

Después de descartar las herramientas anteriores, se presenta la solución por la que finalmente se optó. Durante la fase de investigación se encontró una herramienta llamada *Sikuli* capaz de realizar *testing UI* mediante reconocimiento de imágenes.

Sikuli permite realizar acciones automatizadas sobre cualquier cosa que sea visible en la pantalla del ordenador. Además, es compatible tanto con *Windows* como con *Mac OS X*.

Análisis de la herramienta

En la imagen 4.9, extraída de la referencia bibliográfica [9], pueden verse los módulos que hacen posible la herramienta.

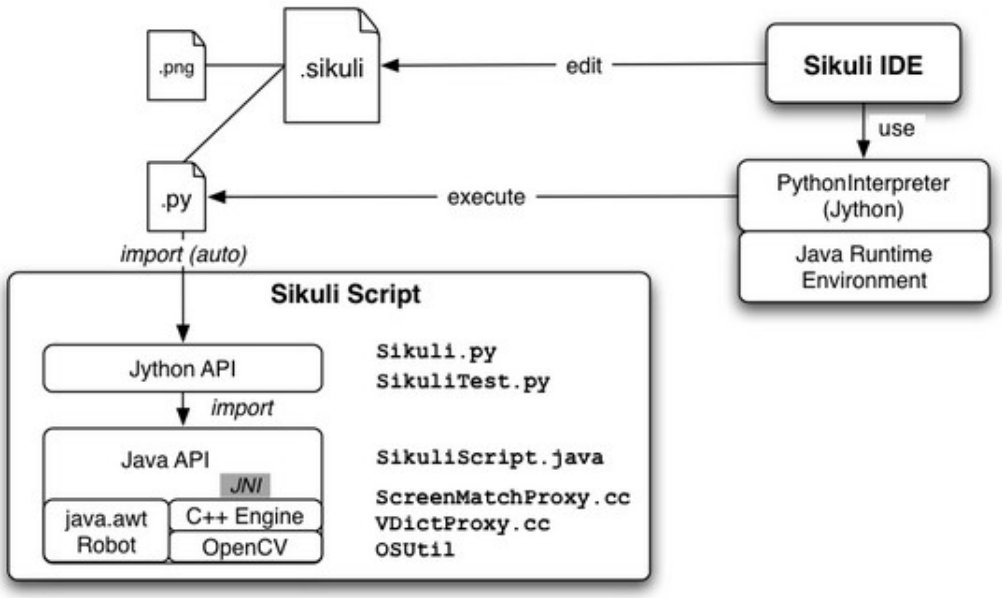


Figura 4.9: Sikuli: Diagrama de la herramienta.

Sikuli Script

Sikuli Script es una *librería Jython* y *Java* que automatiza interacción GUI usando patrones de imagen para dirigir eventos de teclado y ratón del ordenador. El *core* de *Sikuli Script* es una *librería Java* que consiste en dos partes:

- *java.awt.Robot*: efectúa eventos de teclado y ratón en la posición adecuada.
- Motor *C++* basado en *OpenCV*: busca los patrones de imagen en la pantalla del ordenador.

El motor *C++* está conectado a *Java* mediante Java Native Interface (JNI). Por encima de la *librería Java*, existe una capa *Jython* para el usuario final que realiza los *scripts*, con una serie de comandos fácilmente comprensibles.

Estructura de un *script* de *Sikuli*

Un *script* de *Sikuli*(un archivo *.sikuli*) es un directorio que contiene un archivo *.py* y todas las imágenes *.png* que se mencionan en el archivo *.py*. El código del *script .py* puede modificarse en cualquier editor de texto, pero si se hace desde el IDE de *Sikuli*, éste crea un archivo extra *.html* que también se crea en el directorio *.sikuli*. Este último archivo *.html* no aporta nada en cuanto a la ejecución de los *tests*, simplemente aporta una forma diferente de visualizar el *script*.

El IDE de *Sikuli*

El IDE de *Sikuli* permite editar y compilar los *scripts*. Integra una funcionalidad para capturar imágenes y un editor de texto personalizado para facilitar la tarea de escritura de los *scripts*. En la imagen 4.10 se muestra el IDE de *Sikuli* en un *Windows XP*.

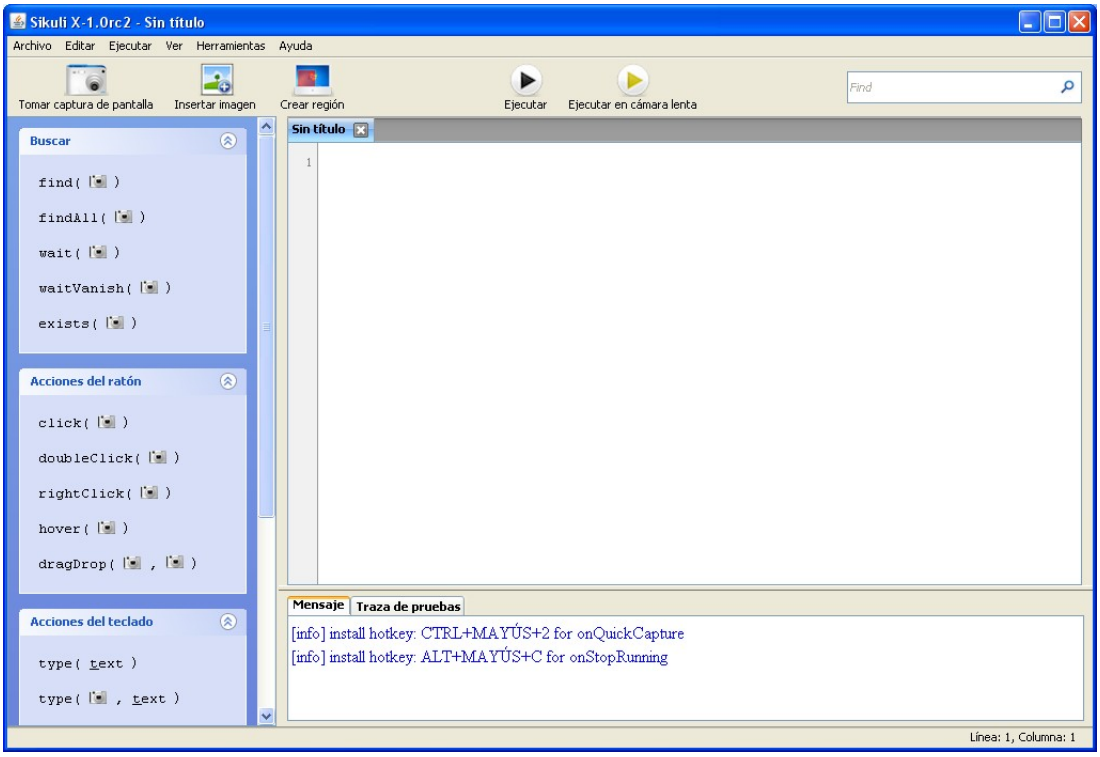


Figura 4.10: Sikuli: IDE de Sikuli.

Para ejecutar un *script* de *Sikuli*, el IDE de *Sikuli* crea un *org.python.util.PythonInterpreter* y automáticamente envía una serie de cabeceras (como los módulos *Jython* de *Sikuli* y la ruta

del directorio *.sikuli*) al intérprete. Una vez establecidas estas cabeceras, se ejecuta el archivo *.py* mediante *PythonInterpreter.execfile()*.

Realización de *scripts*

El archivo *.py* del directorio *.sikuli* es el que contiene el código de las acciones que debe realizar *Sikuli*. Como se ha comentado anteriormente, dicho archivo puede modificarse directamente en un editor de texto o mediante el IDE de *Sikuli*. El resultado del *script* será exactamente el mismo, tan sólo cambia la interfaz gráfica que el usuario tiene al escribirlo. Por ejemplo, en las imágenes 4.11 y 4.12 puede verse el mismo código, en la primera sería desde el IDE de *Sikuli* y en la segunda desde *Notepad ++*.

Figura 4.11: Sikuli: Ejemplo *script* desde el IDE de Sikuli.

```
click("LogoWindows.png")
type("hello world")
```

Figura 4.12: Sikuli: Ejemplo *script* desde un editor de texto.

Cómo encaja *Sikuli* en el proyecto

Sikuli automatiza acciones sobre la pantalla de un ordenador con lo que, si se obtuviera la imagen de un móvil en el ordenador, también podría automatizar acciones sobre la pantalla de un móvil. Para ello tan sólo quedaría enviar esas acciones al móvil. Todo esto se resuelve con una conexión VNC entre el ordenador y el móvil, donde el primero actúa de cliente y el segundo de servidor.

En este sentido se estudió si esto era posible y se encontró lo siguiente:

- Terminales iOS:

  La compañía Apple es muy restrictiva con según qué acciones se quieren realizar sobre terminales iOS y, en este caso, no permite subir aplicaciones de tipo servidor VNC a *App Store*. No obstante, si se realiza *Jailbreak* al dispositivo es posible encontrar este tipo de aplicaciones en *Cydia*, el equivalente a *App Store* una vez se ha realizado *Jailbreak*. Concretamente existe una aplicación llamada *Veency* que realiza esta función.

- Terminales Android:

  La mayoría de aplicaciones Android que sirven como servidor VNC requieren acceso *root* en el móvil, pero existe una aplicación de pago disponible en *Play Store* llamada *VNCLite* que no lo necesita.

Pros y Contras de la herramienta

- Pros:
  - No requiere ningún tipo de conocimiento del código de la aplicación.
  - El lenguaje de los *scripts* es muy sencillo.

  - Aplicable a todo tipo de Sistemas Operativos móviles.
  - Se puede ajustar el grado de exactitud de la imagen de referencia con la imagen obtenida en la pantalla.

- Contras:
  - Una imagen no se detecta si, a pesar de ser la misma que la imagen de referencia, el tamaño o la resolución es diferente.
  - En el caso de dispositivos iOS, es necesario realizar *Jailbreak*.

4.2.3. Conclusiones de la búsqueda

Durante la fase de investigación se consideró que la última herramienta mencionada, *Sikuli*, era la que mejor se podía adaptar a este PFC. Si bien es cierto que con ella no se cumplen todos los objetivos que se habían propuesto, sí gran parte de ellos.

Como objetivo indispensable se había fijado encontrar una tecnología compatible con iOS. Pues bien, con *Sikuli* no sólo se obtiene compatibilidad con iOS sino también con Android y con cualquier Sistema Operativo móvil.

Otro objetivo indispensable era que la herramienta fuera útil sin conocer el código de la aplicación. Obviamente *Sikuli* lo es ya que no interactúa en ningún caso con el código, sino simplemente con la imagen que obtiene en la pantalla del ordenador.

De entre los requisitos deseables que se habían mencionado, a parte de la compatibilidad tanto con iOS y Android que ya se ha comentado, no ha sido posible evitar el acceso *root* a los terminales. En el caso de Android sí que se puede evitar si se utiliza la aplicación de pago *VNCLite*, pero a los terminales iOS se les tendría que realizar un *Jailbreak* para utilizar *Sikuli*.

Por otro lado, en un principio se buscaba una herramienta que interactuara con las aplicaciones *por objeto* porque se consideró que sería más sencillo de esa forma, pero finalmente se vio que ese objetivo complicaba las cosas. En el caso de iOS, simplemente no es posible realizarlo sin tener el código de la aplicación, mientras que en Android sí puede hacerse, pero se ha visto que no de una forma sencilla. En este sentido, finalmente no se interactuará con la aplicación *por objeto* sino *por imagen*.

Por otro lado, otro objetivo deseable era que la creación y modificación de los *scripts* fuera sencilla. Con *Sikuli* no sólo es sencilla sino que visualmente se entiende perfectamente las acciones que realiza el *test*.

En conclusión, después de un largo periodo de investigación acerca de las herramientas de comunicación con terminales móviles existentes, se encontró una manera de solventar el hándicap mediante una solución que nada tiene que ver, a priori, con aplicaciones móviles, pero que aplicada de forma inteligente cumple perfectamente el cometido.

Capítulo 5

Agente

5.1. Fase de diseño

Tras la investigación que se ha detallado en el capítulo anterior, se inició el desarrollo más importante de este PFC, el *software* denominado agente. Éste se desarrolló en Java J2SE dada la oportunidad de reutilización y la versatilidad de *librerías* de objetos de este lenguaje. Precisamente con el objetivo de aprovechar esas características propias del Java así como para realizar una buena planificación del código, se diseñaron diagramas Unified Modeling Language (UML) previos al propio desarrollo de la aplicación.

5.1.1. Diagramas UML

UML es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de *software*. Así, la realización de diagramas UML sirve para representar visualmente las reglas de creación, estructura y comportamiento de un grupo relacionado de objetos y procesos, así como para mantener ágilmente las especificaciones ante cambios y nuevas actualizaciones de la arquitectura.

A continuación se comentarán los diagramas UML que se realizaron en este PFC, en los que se detalló qué debía realizar la aplicación Java y cómo debía hacerlo a nivel de código.

5.1.1.1. Diagrama de flujo

En primer lugar se ideó un diagrama de flujo en el que plasmar los pasos que debía realizar la aplicación Java a desarrollar. Cabe mencionar la diferencia entre el diagrama de flujo que se expondrá a continuación y el previamente mostrado en la imagen 3.2. Éste último resume de forma general las funcionalidades del sistema completo de monitorización englobando todos los elementos de la arquitectura del sistema, mientras que el diagrama de flujo al que se refiere esta sección describe exclusivamente el flujo a seguir del código Java, es decir, incluyendo tan sólo el agente y el *Web Service*.

En el anexo C puede verse el diagrama de flujo realizado. El estado inicial corresponde al momento en que se abre el programa, a partir del cual empieza su inicialización. Durante ese proceso se muestra un mensaje de bienvenida indicando que la aplicación se está cargando. Mientras tanto se realizan una serie de acciones necesarias para la monitorización. Esto es básicamente identificar el robot en el que se ejecuta el agente y descargar los correspondientes tests. Para ello el usuario habrá escrito previamente el nombre del robot en un archivo de tipo *properties* y a partir de éste la aplicación identifica los tests de dicho robot y los descarga de un servidor. Esta descarga se realiza una única vez, en la inicialización, para posteriormente reproducirlos de forma cíclica. En caso de que el usuario quisiera cambiar la identificación del robot o actualizar de alguna forma la lista de tests (por ejemplo, añadiendo uno nuevo), tendría que reiniciarse la aplicación para que reconociera los cambios.

Una vez terminada la inicialización, el mensaje de bienvenida desaparece y se abre el programa en estado de espera. En el momento en que el usuario pulsa el botón *play*, la ejecución de tests comienza siguiendo el orden de una lista. Para ello en primer lugar es necesario comprobar si al test le toca ejecutarse, es decir, como parte de la configuración de un test es posible seleccionar, por ejemplo, horas en concreto en las que el test se ejecuta, con lo que es necesario comprobar si la hora en la que se realiza esa comprobación está dentro del calendario específico del test. En caso afirmativo, lo siguiente es saber en qué terminal móvil debe ejecutarse para abrir una conexión VNC con el mismo. Dicha conexión se realiza en dos fases, primero se redirecciona el puerto TCP local de la máquina al del móvil y después se abre un cliente VNC al puerto TCP local, es decir, conectando con *localhost*. Se decidió hacerlo de esta forma para que la conexión entre la máquina y el móvil fuera vía USB y no por WiFi para ser así más rápida y estable.

Una vez abierta la conexión VNC, se ejecuta el test correspondiente y se realizan las acciones pertinentes para poder ver su resultado con posterioridad. Por un lado se actualiza el archivo de logs del test. En este archivo pueden verse todos los pasos que ha realizado el *script*, incluso si ha tenido un error de código. Este archivo es relevante en caso de detectarse una anomalía en la ejecución del test ya que se detalla todo lo que ha hecho y en qué punto ha fallado.

Por otro lado, a partir de la salida del *script* (la que se ha escrito a su vez en el archivo de logs anterior), se extraen datos relevantes como pueden ser los pasos que se han realizado, en qué momento (con precisión de milisegundos), el nombre de las capturas realizadas y si el test ha finalizado correctamente o no. Todos estos datos se procesan y se envían donde corresponden. Las capturas al servidor y FTP y todo lo demás a la Base de Datos.

Además del flujo que se acaba de comentar, en el diagrama del anexo C también se contemplan casos anómalos en los que se producen errores de diferente tipo, ya que de no tenerlos en cuenta la aplicación fallaría en tiempo de ejecución o se perderían resultados, dependiendo del tipo de error.

5.1.1.2. Diagrama de clases

A partir del diagrama de flujo anterior, se realizó el correspondiente diagrama de clases. En él se pensó cómo realizar en código Java todas las funcionalidades y los estados descritos en el diagrama de flujo. Para ello se estudiaron las posibilidades de simplificación y reutilización del código, siendo el resultado el que se muestra en el anexo D.

La estructura del código se divide en distintas clases Java, cada una de las cuales tiene un sentido como tal y una serie de funcionalidades. A continuación se citan todas las clases que contiene el agente razonando la existencia de cada una de ellas.

Principal.java

Esta clase contiene el método *main()*, con lo que es la primera que se utiliza al iniciar la aplicación. Así, se encarga de realizar toda la fase de inicialización que se detalla en el diagrama de flujo del anexo C, de forma que instancia muchas de las otras clases para después *pasarlas* y que éstas no sean nuevamente instanciadas.

ArchivosProperties.java

Esta clase es la encargada de gestionar los archivos *.properties* que se utilizan en la aplicación, que en este caso son dos, uno de configuración y otro de lenguaje. El primero corresponde a un archivo que se encuentra en una ruta local de la máquina, de forma que el usuario puede modificarlo en un editor de texto. Su sentido reside en proporcionar la posibilidad de configurar elementos externos al código y que el usuario puede querer cambiar, tales como el nombre del robot asignado a la máquina y las IP's tanto del *Web Service* como del servidor FTP. En el anexo E puede verse un ejemplo del contenido de este archivo.

El otro archivo *.properties* forma parte del proyecto Java y contiene una serie de literales a partir de los cuales definir las cadenas de caracteres que tiene la aplicación. La creación de este archivo se hizo con la idea de, en caso de querer cambiar el idioma del programa, crear otro con los mismos literales cambiando tan sólo el idioma de la definición. Por ejemplo, una línea de dicho archivo podría ser la siguiente:

whatToWriteWhenTestsAreStopped = Pulse el botón PLAY para iniciar las ejecuciones

En la versión inglesa esa misma línea sería de la siguiente forma:

whatToWriteWhenTestsAreStopped = Press PLAY button to start executions

Con la utilización de este tipo de archivos de lenguaje se evita utilizar cadenas de caracteres constantes, de forma que para cambiar el idioma de la aplicación tan sólo habría que apuntar al archivo correspondiente, siendo todo el código el mismo.

ParteGraficaAgente.java

Esta clase contiene todo el código relacionado con el *display* del agente, tanto la ventana de inicio en la que se muestra que se está cargando el programa, como la interfaz del agente y todos los mensajes de información o de error que puedan ir apareciendo.

ServidorFTP.java

Esta clase es la encargada de comunicarse con el servidor FTP. La comunicación puede ser tanto de subida como de bajada. Por ejemplo, siguiendo el diagrama de flujo del anexo C, durante la inicialización del programa es necesario descargar los tests del robot, es decir, archivos *.sikuli*. Por otro lado, según ese mismo diagrama, después de cada ejecución de un test deben enviarse las capturas, con lo que en ese caso la comunicación será de subida.

Robot.java

Esta clase se utiliza para contener las características del robot que forma la máquina en la que se ejecuta el agente. Cada robot contiene una lista de tests a ejecutar, por ello esta clase tiene una relación de 1 a 1...* con la clase *Test.java*.

La existencia de la clase *Robot.java* es un claro ejemplo de programación orientada a objetos, ya que mediante el objeto *Robot* el resto de clases acceden a todos los tests y a sus calendarios.

Test.java

Esta clase define a un test de los que el agente reproduce. Cada test tiene ciertas características tales como el *id* que tiene en la Base de Datos, su nombre, el terminal móvil en el que se reproduce, el número de reintentos, etc. Una característica importante de un test es su calendario, es decir, la ventana de tiempo en la que se ejecuta. Este concepto tiene suficiente entidad como para requerir de una clase propia, en este caso, *CalendarioTest.java*. La relación con esta clase es de 1 a 1 ya que un test tiene un único calendario.

CalendarioTest.java

Esta clase contiene toda la información relativa al intervalo de tiempo en el que debe reproducirse un test. El nivel de personalización es muy alto, puede definirse lo siguiente:

- Día del año en el que comenzar las ejecuciones.
- Día del año en el que finalizar las ejecuciones.
- Hora del día de inicio.
- Hora del día de fin.
- Días de la semana concretos.

- Días del mes concretos.
- Frecuencia de ejecución (por ejemplo, cada 5 minutos).

Así, con esta clase se controla si a un test le toca ejecutarse en un momento determinado.

WebService.java

Esta clase se utiliza para la comunicación con el *Web Service*, necesaria tanto para obtener información de la Base de Datos como para enviarla.

GestionEjecucionesTests.java

Esta clase ejerce la función de *scheduler*, es decir, controla el orden de ejecución de los tests siguiendo una lista definida. Si a un test le toca ejecutarse, también se encarga de realizar esa ejecución. Para ello hace uso de la clase *ConexionConMovil.java* para establecer la conexión VNC con el terminal móvil. Por último, una vez finalizado el test utiliza la clase *GestionResultadosTest.java* para que ésta se encargue de hacer lo que corresponda con los resultados obtenidos de esa ejecución.

ConexionConMovil.java

Esta clase se encarga de toda la interacción con el móvil. Es la encargada, por ejemplo, de comprobar si el móvil está conectado a la máquina y de establecer la conexión VNC con éste. También se haría uso de esta clase en caso de querer obtener información del terminal como pudiera ser, por ejemplo, si éste tiene salida a Internet.

GestionResultadosTests.java

Esta clase realiza las funciones relativas a analizar, almacenar y enviar los resultados de las ejecuciones de los tests. De forma resumida cuando un test acaba, lo que hace esta clase es lo siguiente:

- Leer la salida del test para obtener la información de su ejecución. Los datos obtenidos se almacenan en el objeto *ResultadosTest.java*.
- Actualizar el archivo de *logs* correspondiente escribiendo la salida del test.
- Calcular el tiempo entre pasos.
- Enviar las capturas al servidor FTP.
- Enviar los resultados al *Web Service* para que éste los envíe a la Base de Datos.

ResultadosTest.java

Esta clase contiene aquellos atributos que definen el resultado de un test para poder ser enviado a la Base de Datos. Algunos son propios del test, tales como del nombre de éste o el terminal móvil en el que se reproduce, pero la mayoría son relativos a la ejecución, por ejemplo, si el test ha ido bien o mal, el número de pasos que se han ejecutado, el momento en el que se ha ejecutado cada uno, el nombre de las capturas que se han tomado, etc.

La existencia de esta clase facilita la gestión de los resultados. La clase *GestionResultadosTests.java* obtiene todos los datos y los almacena en el objeto *ResultadosTest.java* para

utilizarlo de forma inteligente. Por ejemplo, para enviar las capturas por FTP, en vez de *pasárselas* una a una a la clase *ServidorFTP.java*, le *pasa* el objeto *ResultadosTest.java* ya que contiene esa información. Con el mismo objetivo también le envía el objeto *ResultadosTest.java* a la clase *WebService.java*.

Log.java

Esta clase se encarga de gestionar los logs de la aplicación, los cuales son necesarios para controlar su funcionamiento. Durante la ejecución del agente se van actualizando dos archivos, uno acabado en "log" y otro en "error". La idea es que el usuario consulte el archivo acabado en "error" para ver qué errores ha tenido la aplicación y, en caso de querer ver cómo se ha llegado a ese error, consultar entonces el otro archivo.

En la imagen 5.1 se puede ver un ejemplo del contenido del archivo de errores. Los errores que se muestran se produjeron realmente y en esta imagen se han marcado con colores para facilitar su explicación. El rojo fue un error que se produjo en el momento de establecer conexión VNC con el terminal móvil. El verde se produjo al enviar los resultados de un test a la Base de datos. Se estableció conexión con el Web Service, pero éste tuvo un problema al enviar la sentencia SQL a la Base de Datos, con lo que realmente no se enviaron los resultados. Por último, el azul se produjo en un momento en el se cayó el Web Service, con lo que la aplicación no pudo establecer conexión con él.

```
[14-01-2014 09:37:17,233] ERROR: No se ha encontrado ningun Android conectado        [com.nextret.ConexionConMovil:142]
[15-01-2014 19:31:24,736] ERROR al enviar la sentencia SQL a la Base de datos.
                          No se han enviado los resultados.                          [com.nextret.WebService:228]
[16-01-2014 00:31:39,658] java.net.ConnectException: Connection timed out: connect
        at java.net.DualStackPlainSocketImpl.connect0(Native Method)
        at java.net.DualStackPlainSocketImpl.socketConnect(Unknown Source)
        at java.net.AbstractPlainSocketImpl.doConnect(Unknown Source)
        at java.net.AbstractPlainSocketImpl.connectToAddress(Unknown Source)
        at java.net.AbstractPlainSocketImpl.connect(Unknown Source)
        at java.net.PlainSocketImpl.connect(Unknown Source)
        at java.net.SocksSocketImpl.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at org.apache.commons.net.SocketClient.connect(SocketClient
                                .java:182)
        at org.apache.commons.net.SocketClient.connect(SocketClient.java:203)
        at com.nextret.ServidorFTP.sendImageToFTPServer(ServidorFTP.java:75)
        at com.nextret.ServidorFTP.sendTestImagesToF
                            TPServer(ServidorFTP.java:59)
        at com.nextret.GestionResultadosTest.createResultsAndSendThem(
                            GestionResultadosTest.java:88)
        at com.nextret.GestionEjecucionesTests$1.run(GestionEjecucionesTests
                            .java:68)
                                                                                   [com.nextret.Log:259]
```

Figura 5.1: Agente: Ejemplo contenido archivo de logs de errores de la aplicación.

En los logs se indica tanto el día como la hora de cada traza, el mensaje y, tras éste, la clase y la línea de la aplicación. Además, la clase *Log.java* se ha hecho de tal forma que los archivos no ocupen más de 100 KB, es decir, en caso de alcanzar ese tamaño, se crea un nuevo archivo. De no hacerse de esa forma habría únicamente dos archivos, pero ocuparían tanto que no podrían abrirse. Por otro lado, para identificar los archivos lo que se hace es indicar la fecha de la primera traza en el nombre del archivo. En la imagen 5.2 puede verse un ejemplo del nombre de estos archivos.

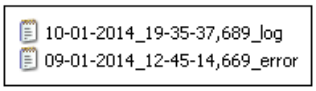


Figura 5.2: Agente: Ejemplo contenido archivo de logs de errores de la aplicación.

5.2. Resultado del desarrollo

Una vez realizados los diagramas de flujo y de clases, la tarea de escribir código consistió en seguir la lógica de ambos diagramas. En esta sección se explica el resultado de esa programación.

5.2.1. Estructura del Proyecto

El proyecto Java desarrollado tiene la estructura que se muestra en la imagen 5.3. Como puede verse se ha creado un *package* llamado *com.nextret* que contiene tanto las clases Java como el archivo *.properties* de idioma, mientras que las imágenes están en *com.nextret.images*. El proyecto se ha compilado en Java 7 y se ha requerido el uso de las siguientes librerías:

- *Commons-net.jar*: Necesaria para la comunicación FTP con el servidor.
- *Joda-time.jar*: Se utiliza para todo lo relacionado con el tiempo: para definir el calendario de un test, para comprobar si éste se encuentra en el intervalo de ejecución en un momento determinado, para calcular el tiempo entre pasos, etc.
- *Jsch.jar*: Necesaria para la comunicación SSH con el terminal móvil.
- *Swt.jar*: Se escogió esta librería para realizar la interfaz gráfica del agente dado que adopta el estilo del Sistema Operativo en el que se ejecuta la aplicación.

5.2.2. Análisis del código

En este apartado se comenta el código desarrollado siguiendo el flujo de la aplicación, analizando los métodos que se consideran más relevantes.

5.2.2.1. Inicialización de la aplicación

En el momento en que se abre el programa se ejecutan las líneas de código que se muestran en la imagen 5.4 y que corresponden a la función *main* de la clase *Principal.java*. Dicha función ejecuta el método *initializeAgente*, que es el que se encarga de realizar toda la inicialización del programa. El código de *initializeAgente* se muestra de forma resumida en la imagen 5.5. Se ha considerado oportuno eliminar partes del código poco relevantes y poner puntos suspensivos en su lugar para resaltar las líneas más interesantes y que el lector siga la explicación de forma más fluida.

Así, como se puede ver en la imagen 5.5, antes de realizar ninguna acción se ejecuta la línea *parteGraficaAgente.showInitWindow()* con la que, tal y como requería el diagrama de flujo del anexo C, se muestra una ventana de inicio mientras se realizan las acciones pertinentes. La ventana de inicio es como la de la imagen 5.6 en la que se informa del progreso que se lleva.

Siguiendo con el diagrama de flujo, tras abrir la ventana inicial en primer lugar se cargan los archivos *properties*. A continuación, a partir del nombre del robot que el usuario haya puesto en el archivo *.properties* de la máquina, se accede a la Base de Datos y se obtiene el identificador del robot. A partir de ese identificador, se accede nuevamente a la Base de Datos y se obtiene la lista de tests asignada al robot. A continuación, para cada test se obtiene la siguiente información de la Base de Datos:

1. A partir del identificador del test, se obtienen los identificadores de los calendarios que tiene el test (en caso de tener alguno).
2. A partir de esos identificadores, se obtiene cada uno de los campos que identifican al calendario para crear así las respectivas clases *CalendarioTest.java*.

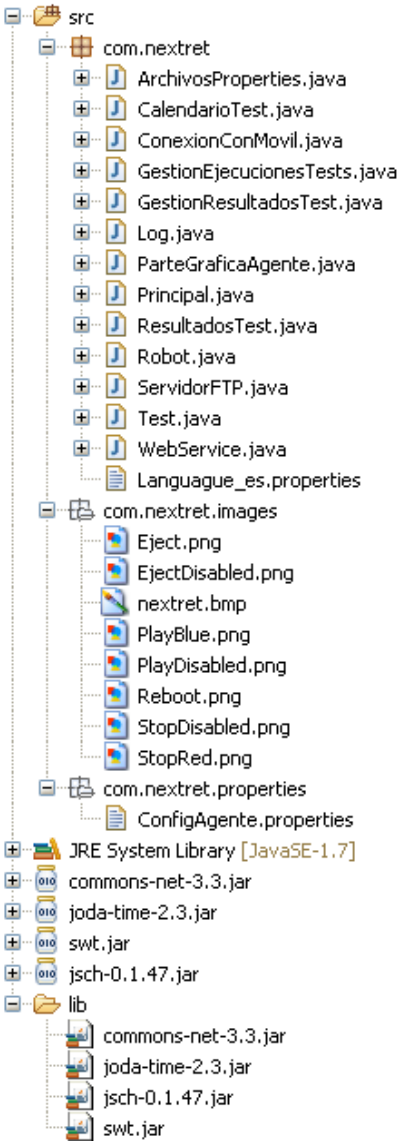


Figura 5.3: Agente: Estructura del proyecto Java.

3. Una vez se han creado los calendarios del test, a partir del identificador del test que ya se tenía, se obtienen el resto de características que identifican al test y se crea la clase *Test.java* correspondiente.

Puesto que este proceso se repite para cada test, los pasos se hacen mediante un "for" y los objetos *Test.java* se van almacenando de forma ordenada en un *TreeMap*. Finalmente se crea el objeto *Robot.java* que contiene todos los tests y se procede a descargar los correspondientes archivos *.sikuli* del servidor. Para ello se hace uso de la clase *ServidorFTP.java*, en concreto de la función *downloadTestsFromFTPServer*. Si la descarga de estos archivos se realiza correctamente, se puede dar por finalizada la inicialización del agente.

5.2.2.2. Interfaz gráfica de la aplicación

Tras la inicialización, desaparece la ventana de bienvenida y tal y como puede verse en la imagen 5.4, se ejecuta la función *showAgente* de la clase *ParteGraficaAgente.java*, es decir, se muestra la interfaz del agente que se ve en la imagen 5.7.

```
public static void main(String[] args) {
    Log.info("////////// Iniciando agente //////////");

    boolean initializationOK = initializeAgente();
    if (initializationOK) {
        parteGraficaAgente.showAgente(archivosProperties, robot, gestionEjecucionesTests);
    }
}
```

Figura 5.4: Agente: Código función *main* de la clase *Principal.java*

En la interfaz gráfica del agente se incluyen los botones PLAY y STOP para iniciar o parar las ejecuciones y, según el caso, éstos cambian. A la izquierda de la imagen 5.8 pueden verse ambos botones en estado de pausa, con el botón STOP deshabilitado, mientras que los botones de la derecha corresponden al estado de ejecución, con el botón PLAY deshabilitado.

En la ventana del agente también se muestra la lista de tests que, en caso de estar activa, sirve para indicar en cada momento el test que se está ejecutando. Esto se hace con un icono verde a la izquierda del test, tal y como se puede ver en la imagen 5.9. Por otro lado, en la parte inferior de la ventana, debajo de la lista, se incluye un *display* con el que informar continuamente de las acciones que está haciendo la aplicación. De esta forma el usuario sabe que el agente está activo aunque parezca que no esté haciendo nada, como puede ser el intervalo entre test y test en el que el usuario no ve nada, pero el agente realiza acciones en *background* como enviar los resultados a la Base de Datos.

Para finalizar con la explicación de la interfaz gráfica, comentar que en la última fase del proyecto se han empezado a desarrollar nuevas funcionalidades del agente y para ello se ha añadido una nueva pestaña al programa, la que tiene por nombre "Móviles Conectados". Al clicar en ella aparecen los terminales móviles conectados a la máquina y una serie de opciones configurables por el usuario. En la imagen 5.10 se muestra un ejemplo en el que se detecta un único terminal móvil, en este caso un iPad 3. Las opciones que se están desarrollando son las siguientes:

- Conexión: Se refiere al acceso a Internet del móvil. Las opciones son WiFi o 3G. La idea es que en caso de que el móvil tenga una tarjeta con datos, el usuario pueda optar por la opción 3G de este *combo box*, de forma que el agente deshabilite el WiFi del móvil provocando así la activación automática de los datos móviles.
- Reiniciar: En caso de clicar este botón, el usuario reiniciaría el terminal móvil desde el agente, sin necesidad de tocar el dispositivo físico.

5.2.2.3. Gestión de las ejecuciones de los tests

En el momento en que el usuario clica el botón PLAY, se ejecutan las líneas de la imagen 5.11. Esto es, una serie de cambios en la interfaz gráfica para mostrar mediante los botones PLAY y STOP que las ejecuciones se han iniciado (parte derecha de la imagen 5.8) y a continuación se le da el control a la función *createThreadToExecuteTests* de la clase *GestionEjecucionesTests.java*.

Para facilitar la explicación sobre la función *createThreadToExecuteTests*, se muestra su código de forma resumida en la imagen 5.12. Como puede verse, la función se ejecuta en un *thread* paralelo. Esto se hace para que el *thread* principal siga activo para quedar a la escucha de las acciones del usuario (por si pulsa el botón STOP, por ejemplo).

La función *createThreadToExecuteTests* básicamente realiza las siguientes acciones:

1. Obtiene la lista de tests a ejecutar a partir del objeto *robot*, parámetro de la función de tipo *Robot.java*.

2. Asigna el valor 1 al parámetro *idGrupTest*, que es el que controla el número de ejecución (el primer test es 1, el segundo 2, el tercero 3, etc. independientemente de qué test es).

3. Empieza un bucle en el que se ejecutan los tests cíclicamente y que no acaba a menos que el usuario pulse el botón STOP o que se detecte que el cable USB no está conectado. Este segundo caso se hace porque, si el cable no está conectado, no tiene sentido seguir ejecutando ya que es un problema de *hardware* que sólo resuelve un usuario. Dicho bucle repite lo siguiente:

   - Se ejecuta la función *getNextTestToExecute* para obtener el siguiente test a reproducir.
   - Una vez se tiene el test, se ejecuta la función *connectToDevice* de la clase *ConexionConMovil.java*, la encargada establecer conexión VNC con el terminal móvil.
   - Si la conexión ha sido satisfactoria, se ejecuta el test, que a nivel de código corresponde a la función *executeTest*.
   - Tras ejecutar el test se pasa el testigo a la clase *GestionResultadosTests.java*, concretamente a la función *createResultsAndSendThem*.
   - Finalmente cuando la función anterior finaliza las tareas pertinentes con respecto a los resultados del test, se aumenta en 1 el valor de la variable *idGrupTest* para el siguiente test. En ese momento el ciclo comienza de nuevo.

La función *getNextTestToExecute* anteriormente citada pertenece a la clase *GestionEjecucionesTests.java* y se encarga de comprobar mediante el calendario de los tests si el siguiente test de la lista está en la ventana de ejecución. En caso de no estarlo, pasa a comprobar el siguiente test y así sucesivamente hasta encontrar un test al que le toque ejecutarse. Como se puede en la imagen 5.13 en la que se muestra el código de la función *getNextTestToExecute*, quien realmente comprueba si al test le toca ejecutarse es la función *checkTimeTest* de la clase *Test.java*. Esta función, a su vez, hace uso del método *checkTimeCalendarioTest* de la clase *CalendarioTest.java*. Esto es así porque como se ha comentado con anterioridad, toda la lógica de los calendarios la gestiona esa clase.

La función *checkTimeTest* funciona de la siguiente manera: en caso de que el test no tenga ningún calendario asignado, directamente se devuelve "true" ya que no se debe comprobar ninguna ventana temporal en concreto, en cambio, si el test tiene algún calendario, se comprueba cada uno de ellos y en caso de que alguno cumpla, el test se ejecuta. Para comprobar el calendario, como se ha dicho anteriormente, se hace uso de la función *checkTimeCalendarioTest* cuyo código se muestra en la imagen 5.14. Básicamente mediante una serie de "if's"se comprueba cada uno de los campos que determinan el calendario de un test: fecha de inicio, fecha de fin, día del mes, día de la semana, hora de inicio, hora de fin y tiempo entre ejecuciones. Para cada campo se comprueba si éste ha sido definido, si lo ha sido, se mira si el momento actual está dentro de los márgenes que marca el campo y, en caso de haberse dejado el campo sin especificar, se supera el "if" directamente. Si se superan todos los "if's" significa que el test está dentro de la ventana de ejecución con lo que, en caso de tener un intervalo de tiempo entre muestras, se actualiza el atributo *nextExec* en la función *setNextExec* que puede verse en la imagen 5.15.

Como se ha comentado anteriormente, tras encontrar el test a reproducir el siguiente paso, el cual se detallará en la siguiente sección, es establecer conexión VNC con el terminal móvil correspondiente. Si la conexión se realiza correctamente, se procede a ejecutar el test. Para ello se utiliza la función *executeTest* mostrada en la imagen 5.16. Esta es la función encargada de la interacción con *Sikuli*, la tecnología que realmente realiza el test. La solución que se ha encontrado es *lanzar* la ejecución mediante línea de comandos. Se aprovecha además esa circunstancia para enviar al *script* a ejecutar una serie de argumentos que éste utiliza tanto en el archivo de *logs* sobre el que escribe como en las capturas que realiza.

5.2.2.4. Conexión VNC con el móvil

La función *connectToDevice* de la clase *ConexionConMovil.java* es la encargada de conectarse al móvil mediante VNC. Para ello habría dos opciones:

- Conectarse por VNC directamente a la IP del móvil: esta opción es la más sencilla de desarrollar ya que tan sólo requiere conocer la IP del móvil, pero la conexión VNC no es muy fluida al ser vía WiFi.
- Redireccionar el puerto TCP de la máquina al puerto TCP del móvil para conectarse entonces a la IP *127.0.0.1*. Para ello el móvil debe estar conectado por USB a la máquina. Esta opción requiere de unas acciones previas, pero obtiene un resultado óptimo ya que la conexión VNC es muy fluida.

Se hicieron pruebas y se vio que con una conexión VNC por WiFi los resultados no serían del todo fiables ya que la imagen no llegaba a la máquina local en tiempo real, sino que tenía un pequeño *delay* que podía falsear los resultados. Teniendo en cuenta que se estaba desarrollando una herramienta de monitorización, la opción más sensata era la que la consiguiera una conexión VNC rápida ya que no es permisible que quien monitorice tenga sus propios errores. Así, se investigó cómo resolver la segunda opción y se encontraron dos soluciones, una para iOS y otra para Android.

En la imagen 5.17 puede verse el código de la función *connectToDevice*. En primer lugar se cierran las posibles conexiones VNC que hayan podido quedar abiertas (en principio no debería haber ninguna, pero se hace por seguridad). A continuación se hace uso de la función *forwardPort* que sirve para redirigir un puerto local de la máquina a uno del móvil. Para ello necesita dos parámetros:

- El Sistema Operativo del móvil: sólo utiliza este dato del móvil ya que no se contempla la posibilidad de tener más de un móvil del mismo Sistema Operativo conectado por no ser relevante para una monitorización. En *testing* sí que es importante probar una aplicación móvil en varios terminales, pero en este caso la monitorización de la aplicación es la misma en un terminal o en otro, lo importante es diferenciar entre Sistemas Operativos ya que la aplicación cambia. En ese sentido, la función *forwardPort* interactúa con el móvil de una forma distinta según sea Android o iOS, pero no comprueba a qué terminal en concreto se está conectando (no mira número de serie). En conclusión, tal y como se ha desarrollado el agente, no pueden haber dos Androids o dos iOS's conectados, pero sí un iOS y un Android.
- El puerto a redirigir: la función *forwardPort* se utiliza para redirigir tanto el puerto TCP como el SSH, con lo que se debe especificar cuál redirigir.

En la imagen 5.18 se muestra el código resumido de la función *forwardPort*. En primer lugar se abre una *pipe* que se utiliza para controlar la conexión con el móvil. Esto es debido a que la función que realmente realiza la redirección del puerto es *launchForwardPort* y, como lo hace en un *thread* distinto al que utiliza la función *forwardPort*, ambas funciones se sincronizan mediante una *pipe*. De esta forma, la función *launchForwardPort* indica si la redirección ha sido correcta o no (escribiendo un *string* concreto en la *pipe*) y la función *forwardPort* actúa en consecuencia. Para ello se han establecido ciertos *strings* a enviar y, en consecuencia, a recibir. Con ellos se inicializa la variable *possibleReceivedStrings*, que posteriormente se utiliza en un "for" para comprobar cuál de esos *strings* ha llegado.

La idea es la siguiente: si, por ejemplo, se recibe la palabra "Forwarding", significa que la redirección se ha hecho correctamente, con lo que procederá a realizar la conexión VNC, seguirá esperando una nueva lectura y si recibe "Connection established" saldrá del bucle y de la función, ya que se habrá conseguido el objetivo, establecer la conexión.

La lectura de la *pipe* se hace dentro de un bucle, *while(continueReading)*, del que se sale según lo que se recibe. Como se ha comentado anteriormente, para la redirección del puerto local de la máquina al del móvil se obtuvieron dos soluciones, una para iOS y otra para Android. Es por ello

que el código contiene un *switch* que diferencia ambos casos.

En la imagen 5.19 se muestra el código resumido de la función *launchForwardPort*. Como se puede ver, para conseguir la redirección del puerto se *lanza* una línea de comandos distinta según si se quiere interactuar con un iOS o un Android. Para cada caso se busca en la salida del comando unas líneas concretas y, según lo que se obtiene, se escribe un *string* u otro en la *pipe*. A continuación se explicará la solución empleada para cada caso.

- iOS

  Como se puede ver en el código de la función *launchForwardPort* en la imagen 5.19, lo que se hace es *lanzar* un *script python* indicando el puerto a redirigir. Este *script* es capaz de redirigir el puerto local de la máquina al puerto de un iOS conectado. La salida de este *script* indica en todo momento el estado de esa redirección, con lo que se puede saber si se ha redirigido correctamente, si no ha sido posible porque ya hay una redirección ejecutándose, si no lo ha sido porque no se encuentra ningún iOS conectado, etc.

  Durante la conexión VNC, este *script* debe continuar ejecutándose ya que en el momento que deja de hacerlo, la conexión se cierra. Es por ello que la función *launchForwardPort* se ejecuta en un *thread* paralelo.

- Android

  En este caso no se *lanza* un *script* sino que se hace uso del comando *adb* del SDK de Android.

5.2.2.5. Gestión de los resultados de los tests

Volviendo al código de la función *createThreadToExecuteTests* de la imagen 5.12, se puede ver que tras finalizar la ejecución de un test se procede a tratar los resultados mediante la función *createResultsAndSendThem* de la clase *GestionResultadosTests.java*. Esta función se encarga de escribir los logs del test en el correspondiente archivo, enviar los resultados a la Base de Datos y enviar las capturas al Servidor FTP. En la imagen 5.20 se puede ver el código resumido de la función.

La función *readProcessOutputToWriteInLogFile* es la encargada de leer la salida del *script* y a partir de ésta escribir los logs del test y actualizar algunos de los campos del atributo *resultadosTest*, como pueden ser los nombres de las capturas o el momento en el que se ha ejecutado cada paso.

Cuando la función *readProcessOutputToWriteInLogFile* acaba es porque ha acabado el test, ya que lee su salida a tiempo real y finaliza cuando ya no hay más a leer. Por tanto, si el test ha finalizado se puede cerrar la conexión VNC, que es lo que hace la función *closeVNCconnections* de la clase *ConexionConMovil.java*.

A continuación se puede ver un "if" que se encarga de comprobar si durante la ejecución del test se ha pulsado el botón STOP, o si el test ha fallado por un problema con el código del *script* o del propio *Sikuli*. En ninguno de esos casos se enviarían los resultados ya que, por un lado, si se pulsa el botón STOP durante la ejecución del test éste no ha podido finalizar, con lo que no tiene sentido enviar los resultados y, por otro lado, si el test ha tenido un error de código, sus resultados tampoco son relevantes. Este segundo caso se controla en la función *readProcessOutputToWriteInLogFile* ya que ésta detecta si el *script* ha escrito algo en la salida de error.

En caso de que el test no haya experimentado ninguno de los dos casos anteriores, que será lo más habitual, se entra en el "if" para procesar y enviar los resultados del test. En primer lugar se comprueba si, en caso de que el test haya fallado, lo ha hecho debido a que el terminal móvil

no tiene conexión a Internet. Para ello se establece conexión SSH y se realiza un *ping* o algún comando equivalente. Esto se hace mediante la función *checkDeviceInternetConnection* que básicamente hace uso de la función *forwardPort* que el lector ya conoce. Esta comprobación se hace para descartar que el problema que ha tenido el test sea del propio móvil ya que, en caso de ser así, no será un error que el cliente deba tener en cuenta.

Tras realizar esa comprobación, se calcula el tiempo entre pasos haciendo uso de la función *calculateAllTimesBetweenSteps*. A continuación, se envían las capturas del test al servidor FTP. En este caso en el código de la imagen 5.20 se puede ver que sólo se envían en caso de que el test falle, pero puede hacerse siempre. Actualmente se envían sólo si el test falla ya que se ha comprobado tras varias semanas el agente genera una cantidad de capturas que ocupan bastante espacio en disco, con lo que por el momento se ha optado por esta opción ya que lo más habitual es que el cliente visualice las capturas de las ejecuciones erróneas para ver qué ha pasado.

En la imagen 5.21 se puede ver el código de la función *sendTestImagesToFTPServer*, la encargada de enviar las capturas al servidor. Téngase en cuenta que las capturas las realiza el *script* mientras ejecuta el test y las almacena en un directorio concreto que depende del Sistema Operativo del móvil y del nombre del test, con lo que el agente ya sabe dónde ir a buscarlas y con esa ruta inicializa la variable *pathLocalImagesDirectory*. Por otro lado, en el servidor existe una lógica parecida para almacenar las capturas, así que con la correspondiente ruta se inicializa la variable *pathRemoteImagesDirectory*. Finalmente se hace uso de la función *sendImageToFTPServer* para enviar una a una todas las capturas. El código resumido de dicha función se puede ver en la imagen 5.22. Básicamente lo que se hace es establecer conexión con el servidor FTP, hacer login con el usuario y el password, crear el directorio pertinente en caso de que no exista, enviar la captura a dicho directorio y, en caso de haberla enviado correctamente, eliminar la captura local para que no ocupe espacio innecesario.

Cabe destacar el sentido de crear distintos directorios a la hora de enviar las capturas al servidor. La lógica de almacenaje que se sigue es, en primer lugar, separar los tests por Sistema Operativo, con lo que existiría una carpeta llamada "iPhone" y otra "Android". Dentro de cada una de esas carpetas, habría una carpeta para cada test de ese Sistema Operativo (con el nombre del test sin espacios) y, finalmente, cada test tendría una serie de carpetas con los distintos días en los que se suben capturas (de ahí que se creen directorios en la función *sendImageToFTPServer*.) El motivo de diferenciar las ejecuciones por días es evitar que en un mismo directorio existan muchos archivos, ya que esto siempre causa problemas de rendimiento.

Siguiendo con la función *createResultsAndSendThem*, tras enviar las capturas al servidor el último paso consiste en enviar los resultados al *Web Service* para que éste los envíe a la Base de Datos. Para ello se hace uso de la función *sendResultsToBBDD* de la clase *WebService.java*. Puesto que se dedica el capítulo 6 a la parte del *Web Service*, se considera oportuno comentar el código del agente referente a esta cuestión en ese capítulo.

```
private static boolean initializeAgente() {

    boolean initializationOK = false;

    parteGraficaAgente.showInitWindow();

    if (archivosProperties.loadPropertiesLanguage()) {
        if (archivosProperties.loadPropertiesConfiguration()) {

            ...

            String nameRobot = archivosProperties.getPropertyFromConfigurationProperties("nameRobot");

            ...

            if ( (idRobot = webService.getIdRobotFromNameRobot(nameRobot)) != null ) {

                ...

                if ( (arrayIdsTests = webService.getIdsTestsFromIdRobot(idRobot)) != null ) {

                    ...

                    for ( int i=0; i<arrayIdsTests.size(); i++ ) {
                        isArrayTestsOK = false;
                        idTest = arrayIdsTests.get(i);

                        if ( (arrayIdsCalendaris = webService.getIdsCalendarisFromIdTest(idTest)) != null ) {

                            if ( arrayIdsCalendaris.size() > 0 ) { // si tiene calendarios asignados, que se los baje
                                for ( int j=0; j<arrayIdsCalendaris.size(); j++ ) {
                                    isArrayCalendariosTestOK = false;
                                    idCalendari = arrayIdsCalendaris.get(j);
                                    if ( (calendarioTest = webService.getCalendarioTestClassFromIdCalendari(idCalendari)) != null ) {
                                        isArrayCalendariosTestOK = true;
                                        arrayCalendariosTest.add(calendarioTest);
                                    }
                                    ...
                                }
                            }
                            if (isArrayCalendariosTestOK) {
                                if ( (test = webService.getTestClassFromIdTest(idTest, arrayCalendariosTest)) != null ) {
                                    isArrayTestsOK = true;
                                    arrayTests.put(i,test);
                                }
                            }
                            ...
                        }
                        ...
                    }

                    if ( isArrayTestsOK ) {

                        robot = new Robot(idRobot, arrayTests);

                        ...

                        if ( servidorFTP.downloadTestsFromFTPServer(robot.getArrayTests()) ) { // descarga los ".sikuli"

                            ...

                            Log.info("////////// Agente iniciado ///////////");
                            initializationOK = true;
                            parteGraficaAgente.closeInitWindow();
                        }
                        ...
```

Figura 5.5: Agente: Código inicial de la función *initializeAgente* de la clase *Principal.java*

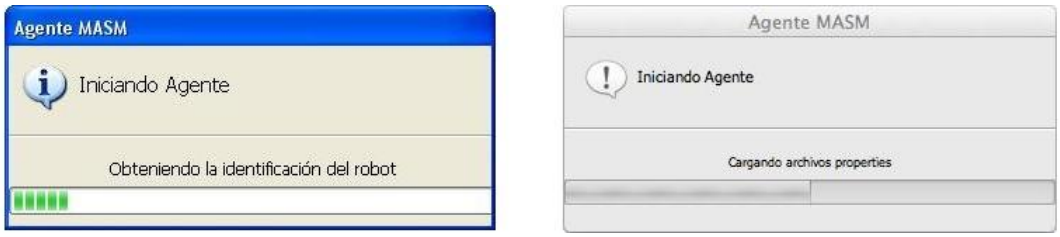


Figura 5.6: Agente: Ventana de inicio (Izquierda: Windows XP, Derecha: Mac OS)

Figura 5.7: Agente: Interfaz gráfica (Izquierda: Windows XP, Derecha: Mac OS)

Figura 5.8: Agente: Botones PLAY y STOP (Izquierda: pausa, Derecha: ejecución)

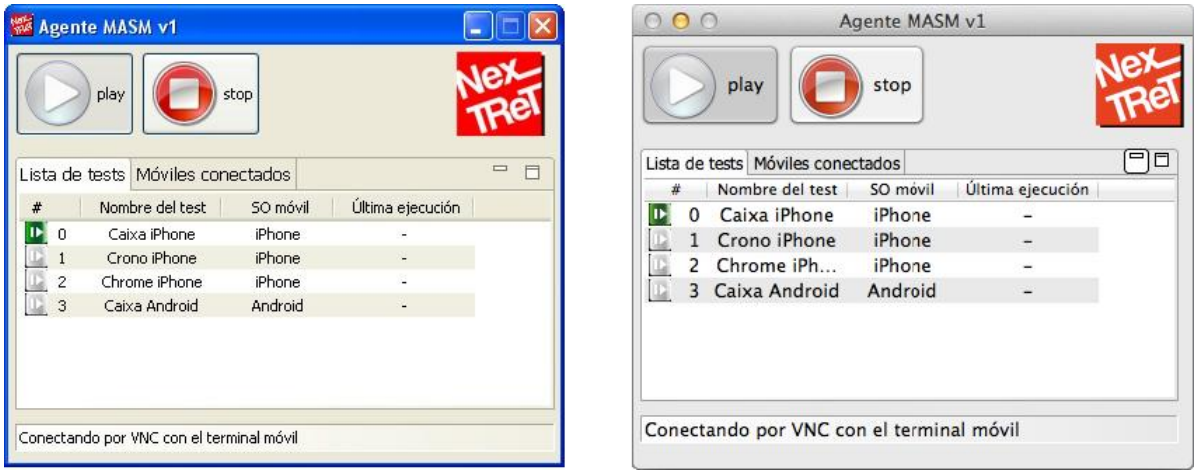


Figura 5.9: Agente: Interfaz gráfica en la que se indica el test que se está ejecutando (Izquierda: Windows XP, Derecha: Mac OS)

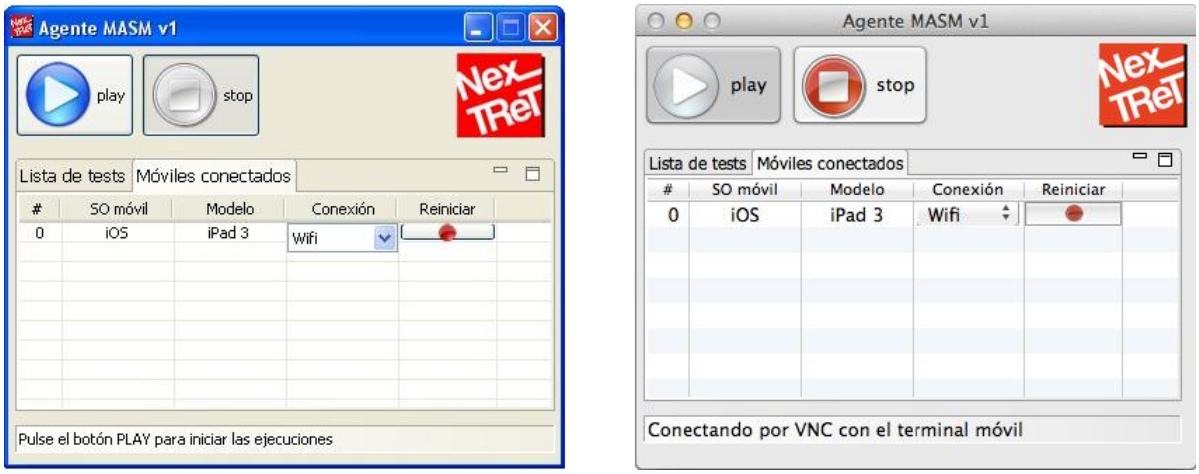


Figura 5.10: Agente: Interfaz gráfica pestaña "Móviles Conectados" (Izquierda: Windows XP, Derecha: Mac OS)

```
// Funcionamiento botón PLAY
buttonPlay.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event arg0) {
        buttonPlay.setImage(imagePlayDisabled);
        buttonPlay.setSelection(true);
        buttonStop.setImage(imageStopRed);
        buttonStop.setSelection(false);
        clickStop = false;
        gestionEjecucionesTests.createThreadToExecuteTests(archivosProperties,robot);
    }
});
// --- FIN funcionamiento botón PLAY
```

Figura 5.11: Agente: Código *listener* botón PLAY de la clase *ParteGraficaAgente.java*

```
public void createThreadToExecuteTests (final ArchivosProperties archivosProperties, final Robot robot ) {

    new Thread() {
        public synchronized void run() {

            ...

            TreeMap<Integer,Test> testList = robot.getTestList();
            Iterator<Entry<Integer,Test>> iterator = testList.entrySet().iterator(); //Para recorrer el TreeMap

            ...

            int idGrupTest = 1;
            while ( ! parteGraficaAgente.getClickStop() && isUsbCableConnected ) {

                ...

                isUsbCableConnected = conexionConMovil.connectToDevice(SOdevice);
                if ( isUsbCableConnected ) {

                    ...

                    process = executeTest(robot, test,idGrupTest);
                    if ( process != null ) {
                        gestionResultadosTest.createResultsAndSendThem(parteGraficaAgente, robot, test, process,
                                idGrupTest, conexionConMovil);
                        idGrupTest++;
                    }
                    ...
                }
                ...
            }
        }
    }.start();
}
```

Figura 5.12: Agente: Código resumido de la función *createThreadToExecuteTests* de la clase *GestionEjecucionesTests.java*

```
public Test getNextTestToExecute(TreeMap<Integer, Test> testList, Iterator<Entry<Integer, Test>> iterator) {

    Test testToCheck = null;
    CalendarioTest calendarioTestToCheck;
    boolean execTimeOk = false;

    while ( !execTimeOk ) { // Sigue hasta que a un test le toque ejecutarse
        numNextTest = iterator.next().getKey();
        testToCheck = arrayTests.get(numNextTest);
        execTimeOk = testToCheck.checkTimeTest();
        if (!iterator.hasNext()) {
            iterator = arrayTests.entrySet().iterator();
        }
    }
    return testToCheck;
}
```

Figura 5.13: Agente: Código de la función *getNextTestToExecute* de la clase *GestionEjecuciones-Tests.java*

```
public boolean checkTimeTest() {

    boolean check = false;
    DateTime today = DateTime.now();

    if ((startDateIsSet() &&
            (getStartDateInDateTimeFormat().isBefore(today) || getStartDateInDateTimeFormat().isEqual(today)))
        || !startDateIsSet()) {

        if ((endDateIsSet() &&
                (getEndDateInDateTimeFormat().isAfter(today) || getEndDateInDateTimeFormat().isEqual(today)))
            || !endDateIsSet()) {

            if ((daysOfMonthIsSet() &&
                    this.daysOfMonth.contains(today.getDayOfMonth()))
                || !daysOfMonthIsSet()) {

                if ((daysOfWeekIsSet() &&
                        this.daysOfWeek.contains(today.getDayOfWeek()))
                    || !daysOfWeekIsSet()) {

                    if (getStartHourOfDay() <= today.getHourOfDay() &&
                        getStartMinuteOfHour() <= today.getMinuteOfHour()) {

                        if (getEndHourOfDay() >= today.getHourOfDay() &&
                            getEndMinuteOfHour() >= today.getMinuteOfHour()) {

                            if ((everyXminutesIsSet() &&
                                this.nextExec.isBefore(today)) || !everyXminutesIsSet()) {

                                check = true;

                                if (everyXminutesIsSet()) {
                                    setNextExec(today);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return check;
}
```

Figura 5.14: Agente: Código de la función *checkTimeCalendarioTest* de la clase *CalendarioTest.java*

```
public void setNextExec (DateTime lastExec) {
    this.nextExec = lastExec.plusMinutes(this.everyXminutes);
}
```

Figura 5.15: Agente: Código de la función *setNextExec* de la clase *CalendarioTest.java*

```java
public Process executeTest (Robot robot, Test test, int idGrupTest) {

    String SOdevice = test.getSOdevice();
    String nameTestWithoutSpaces = test.getNameTest().replaceAll(" ", "");
    String nameImage = robot.getIdRobot() + "-" + nameTestWithoutSpaces;
    String command = commandExecTestFirstPart + SOdevice + "/"
            + nameTestWithoutSpaces + ".sikuli" + "\"" + " "
            + commandExecTestSecondPart + SOdevice + "/"
            + nameTestWithoutSpaces + "/" + "\"" + " " + "\"" + nameImage
            + "\"" + " " + idGrupTest;
    Process process = null;

    try {
        Log.info("Ejecutando comando cmd: " + command);
        process = Runtime.getRuntime().exec(new String[]{"cmd.exe", "/c", command});
    } catch (IOException e) {
        Log.error(e);
    }
    return process;
}
```

Figura 5.16: Agente: Código de la función *executeTest* de la clase *GestionEjecucionesTests.java*

```java
public boolean connectToDevice ( String SOdevice ) {

    closeVNCconnections(); // por si se ha quedado alguna sesión abierta

    boolean connectionEstablished = forwardPort(SOdevice, portVNC);

    return connectionEstablished;
}
```

Figura 5.17: Agente: Código de la función *connectToDevice* de la clase *ConexionConMovil.java*

```
private boolean forwardPort( String SOdevice, String portToRedirect ) {

    ...

        PipedReader pipedReader = new PipedReader();
        PipedWriter pipedWriter = new PipedWriter();
        pipedWriter.connect(pipedReader);

        launchForwardPort( pipedWriter, SOdevice, portToRedirect );

        ArrayList<String> possibleReceivedStrings = new ArrayList<String>();
        possibleReceivedStrings.add("Forwarding");
        possibleReceivedStrings.add("Connection established");
        possibleReceivedStrings.add("No device found");
        possibleReceivedStrings.add("Traceback");
        possibleReceivedStrings.add("NULL");

        ...

        while (continueReading) {

            ...

            for (int i = 0; i < possibleReceivedStrings.size() ; i++) {

                stringReceived = possibleReceivedStrings.get(i);

                if (builder.toString().equals(stringReceived)) {

                    ...

                    switch (SOdevice) {

                    case "iPhone":

                        switch (stringReceived) {

                        case "Forwarding":
                            ...
                        case "Connection established":
                            ...
                        case "No device found":
                            ...
                        case "Traceback":
                            ...
                        case "NULL":
                            ...
                        }
                        break;

                    case "Android":

                        switch (stringReceived) {

                        case "Forwarding":
                            ...
                        case "No device found":
                            ...
                        case "Traceback":
                            ...
                        }
                        break;
                    }
                }
            }
        }

    ...

    return functionFinishedOK;
}
```

Figura 5.18: Agente: Código resumido de la función *forwardPort* de la clase *ConexionConMovil.java*

```
private void launchForwardPort (final PipedWriter pipedWriter, final String SOdevice,
        final String portToRedirect) {
    new Thread() {
        public synchronized void run() {

                ...

                switch (SOdevice) {
                case "iPhone":
                    command = "python " + "\"" + pathTcpRelayIphone + "\"" + " -t "
                            + portToRedirect;
                    break;

                case "Android":
                    command = "adb forward tcp:" + portToRedirect + " tcp:"
                            + portToRedirect;
                    break;
                }

                ...

                Process process = Runtime.getRuntime().exec(
                        new String[] { "cmd.exe", "/c", command });

                ...

                while (continueReading) {

                    switch (SOdevice) {

                    case "iPhone":

                        ...

                        break;

                    case "Android":

                        ...

                        break;
                    }
                }
                ...
        }
    }.start();
}
```

Figura 5.19: Agente: Código resumido de la función *launchForwardPort* de la clase *ConexionConMovil.java*

```
public void createResultsAndSendThem(final ParteGraficaAgente parteGraficaAgente, Robot robot,
        final Test test, Process process, int idGrupTest, ConexionConMovil conexionConMovil) {

    resultadosTest = new ResultadosTest();

    try {
        readProcessOutputToWriteInLogFile (test, process);
        conexionConMovil.closeVNCconnections();

        if ( ! parteGraficaAgente.getClickStop() && resultadosTest.isSendResults() ) {

            ...

            if ( resultadosTest.isTestOk() ) {
                Log.info("El test se ha ejecutado correctamente");
            } else {
                Log.info("El test ha fallado");
                if ( conexionConMovil.checkDeviceInternetConnection(test.getSOdevice()) ) {
                    resultadosTest.setIdError("1");
                } else {
                    resultadosTest.setIdError("3");
                }
            }

            ...

            calculateAllTimesBetweenSteps();

            ...

            if ( resultadosTest.isTestOk() ) {
                deleteImages(test);
            } else {
                servidorFTP.sendTestImagesToFTPServer(test,resultadosTest);
            }

            ...

            webService.sendResultsToBBDD(resultadosTest);
        }
    } catch (FileNotFoundException e) {
        Log.error(e);
    }
}
```

Figura 5.20: Agente: Código resumido de la función *createResultsAndSendThem* de la clase *GestionResultadosTest.java*

```
public void sendTestImagesToFTPServer ( Test test, ResultadosTest resultadosTest ) {

    Log.info("Enviando capturas al servidor");
    String nameTestWithoutSpaces = test.getNameTest().replaceAll(" ", "");
    String pathLocalImagesDirectory = pathImages + test.getSOdevice() + "/" + nameTestWithoutSpaces + "/";
    String pathRemoteImagesDirectory = test.getSOdevice() + "/" + nameTestWithoutSpaces + "/";

    ArrayList<String> namesCapturasPrevias = resultadosTest.getNamesCapturasPrevias();
    String nameLocalImageToSend;

    for ( int i=0; i<namesCapturasPrevias.size() ; i++ ) { // envía las capturas previas
        nameLocalImageToSend = namesCapturasPrevias.get(i);
        sendImageToFTPServer(pathLocalImagesDirectory, nameLocalImageToSend, pathRemoteImagesDirectory);

    }
    if ( ! resultadosTest.isTestOk() ) { // si el test ha ido mal, envía también la captura de error
        nameLocalImageToSend = resultadosTest.getNameCapturaError();
        sendImageToFTPServer(pathLocalImagesDirectory, nameLocalImageToSend, pathRemoteImagesDirectory);
    }
}
```

Figura 5.21: Agente: Código resumido de la función *sendTestImagesToFTPServer* de la clase *ServidorFTP.java*

```
private void sendImageToFTPServer ( String pathLocalImagesDirectory, String nameLocalImageToSend,
        String pathRemoteImagesDirectory ) {

        ...

        String nameRemoteDirectory = DateTime.now().toString("yyyyMMdd");

        Log.info("Conectando con el servidor FTP");
        ftpClient.connect(ip, port);
        if ( ftpClient.login(user, password) ) {

            ...

            ftpClient.changeWorkingDirectory(pathRemoteImagesDirectory);
            boolean directoryExists = false;

            if ( ! (directoryExists = ftpClient.changeWorkingDirectory(nameRemoteDirectory)) ) {
                if ( ftpClient.makeDirectory(nameRemoteDirectory) ) {
                    directoryExists = ftpClient.changeWorkingDirectory(nameRemoteDirectory);
                }
            }
            if (directoryExists) {
                InputStream inputStream = new FileInputStream(localFile);
                Log.info("Enviando la captura " + nameLocalImageToSend + " al servidor FTP");
                sentOK = ftpClient.storeFile(nameLocalImageToSend, inputStream);
                inputStream.close();
                if (sentOK) {
                    localFile.delete();
                }
                ...
            }
            ...
        }
        ...
}
```

Figura 5.22: Agente: Código resumido de la función *sendImageToFTPServer* de la clase *ServidorFTP.java*

Capítulo 6

Web Service

6.1. Introducción

En este proyecto se ha desarrollado un *Web Service* para intercambiar información entre el agente y la Base de Datos. En este caso en particular se da la situación de que ya se disponía de un *Web Service* y se decidió aprovecharlo, con lo que se añadieron métodos al *Web Service* existente. Esta aclaración se hace puesto que de haberlo hecho de cero, probablemente se habría desarrollado de otra manera, ya que como se verá a continuación, este *Web Service* no es convencional.

El agente para comunicarse con el *Web Service* envía URL's con parámetros *doGet* que el *Web Service* interpreta. Éste contesta con una estructura XML y el agente realiza las acciones oportunas según la respuesta que obtiene.

Por ejemplo, el agente podría enviar la siguiente URL:

```
http://192.168.7.129:8080/WS_20_Mobile/PeticionsXML?strXML=<Peticions IdRobot=
'0'><GetIdRobotFromNameRobot NameRobot='robot_piloto'></GetIdRobotFromNameRobot>
</Peticions>
```

La cual se divide de la siguiente forma:

- `http://192.168.7.129:8080/WS_20_Mobile/PeticionsXML?strXML=`

  La parte inicial de la URL hace referencia al servidor donde está desplegado el *servlet*. "`WS_20_Mobile`" es el nombre del proyecto, mientras que "`PeticionsXML`" es la clase en la que están los métodos desarrollados. Esta sentencia es constante, con lo que forma parte del contenido del fichero *ConfigAgente.properties*. Así, en caso de cambiar el servidor de aplicaciones, tan sólo habría que modificar dicho archivo.

- `<Peticions IdRobot='0'>`

  Este atributo, así como su cierre "`</Peticions>`", formaba parte del *Web Service* existente y debía mantenerse.

- `<GetIdRobotFromNameRobot NameRobot='robot_piloto'></GetIdRobotFromNameRobot>`

  Esta es la aportación personal al *Web Service*. En este caso el agente estaría solicitando información sobre la tabla "Robots_Mobile" de la Base de Datos, concretamente el identificador auto-incremental de la fila cuyo nombre de robot sea "robot_piloto". El *Web Service* iría a buscar esa información a la Base de Datos y respondería con un XML como el que se muestra en la imagen 6.1. El agente es capaz de interpretar ese XML y recoger el valor "1" como la información que estaba solicitando.

```
<Resposta  DataServidor='20140131144833'  value='OK' >
<GetIdRobotFromNameRobot  IdRobot='1'  value='OK' >
</GetIdRobotFromNameRobot>
</Resposta>
```

Figura 6.1: *Web Service*: Ejemplo respuesta del *Web Service* a la función *GetIdRobotFromNameRobot*.

En este capítulo se detallarán las funciones relacionadas con el *Web Service* que se han desarrollado, tanto en el código del agente (clase *WebService.java*) como en la aplicación que se despliega en el servidor y que constituye el *Web Service* en sí.

6.2. Resultado del desarrollo

6.2.1. Agente: función *sendURLAndGetNodeResponse* de la clase *WebService.java*

Se considera interesante dedicar un apartado a esta función ya que es la encargada de comunicarse con el *Web Service* del servidor. Así, el resto de funciones la utilizan pasándole por parámetro los atributos que debe enviar y recibiendo de ella un objeto de tipo *Node* con la respuesta del *Web Service*.

Como se puede ver en la imagen 6.2, en primer lugar la función completa la URL a enviar con los parámetros que se le envían. Tras ello, codifica la URL y establece conexión mediante un objeto *URLConnection*. A continuación las siguientes líneas de código se dedican a interpretar la respuesta del XML recibido con tal de encontrar el nodo que interesa.

6.2.2. *Web Service*: cómo interpretar una petición

En el código del *Web Service*, una vez se diferencian los parámetros de la petición que se recibe, se identifica la función que se está solicitando mediante una serie de *if*'s. Los correspondientes a las funciones desarrolladas en este proyecto son los que se muestran en la imagen 6.3. Los 7 primeros *if*'s corresponden a los métodos necesarios para la inicialización del agente, mientras que el último *if* es el que se encarga de insertar los resultados de un test a la Base de Datos.

6.2.3. Agente y *Web Service*: funciones relacionadas con la inicialización del agente

Para seguir la explicación de esta sección se recomienda tener a la vista la imagen 5.5.

Las funciones que se comentan a continuación son bastante equivalentes entre ellas ya que todas tienen un factor común: recibir un parámetro y a partir de éste enviar lo que se ha encontrado en la Base de Datos.

6.2.3.1. Función *getIdRobotFromNameRobot*

Agente

Como se vio en la sección 5.2.2.1 del capítulo 5, la inicialización del agente se realiza a partir del nombre del robot que el usuario haya asignado a la máquina en el fichero *ConfigAgente.properties*. Para ello se hace uso de la función *getIdRobotFromNameRobot* de la clase *WebService.java*. El código de dicha función se muestra en la parte superior de la imagen 6.4. Como se puede ver, este método recibe como parámetro el nombre del robot para enviar como salida el *Id* de dicho robot. Para ello, prepara los atributos a enviar en la variable *urlAttributes*, hace uso de la función *sendURLAndGetNodeResponse* y a partir del objeto *Node* que obtiene de ésta, interpreta si el identificador del robot ha llegado correctamente o no.

Web Service

Por su parte el *Web Service* tiene su función *getIdRobotFromNameRobot* correspondiente encargada de, precisamente, responder a la función *getIdRobotFromNameRobot* del agente. El código se puede ver en la parte inferior de la imagen 6.4. Básicamente se lee el valor del atributo *NameRobot* y con éste se crea la sentencia SQL que obtiene el identificador que se busca. La sentencia puede ser, por ejemplo, si se hubiera recibido "robot_piloto" como nombre, la siguiente:

SELECT Distinct Id FROM Robots_Mobile where Nom='robot_piloto'

De esta forma se obtienen todos los valores del campo *Id* que cumplen que el valor del campo *NameRobot* es "robot_piloto". Así, se controlan posibles errores en la Base de Datos (como haber introducido más de una vez ese nombre) o posibles errores del usuario (como haber puesto un nombre al robot que en la Base de Datos no está definido).

6.2.3.2. Función *getIdsTestsFromIdRobot*

Agente

Una vez se ha obtenido el valor del identificador del robot, se pueden obtener los tests relacionados a éste. De esto se encarga la función *getIdsTestsFromIdRobot* cuyo código se muestra en la parte superior de la imagen 6.5. Como se puede ver, el atributo a enviar en este caso es el *IdRobot* obtenido en la función anterior.

Web Service

La función equivalente en la parte del *Web Service* es la que se muestra en la parte inferior de la imagen 6.5. En este caso la información que se solicita a la Base de Datos es la lista de parámetros *IdTest* que se corresponden con el *IdRobot* recibido. En caso de no obtener ningún test, se informa al agente para que éste muestre un mensaje de error.

6.2.3.3. Función *getIdsCalendarisFromIdTest*

Agente

En caso de obtener uno o más tests, el agente tiene sentido puesto que puede ejecutar algún test, así que debe procederse a inicializar cada uno de los tests con los parámetros que correspondan.

El primer atributo que se obtiene es el calendario del test, puesto que es el más complejo. Para ello, en primer lugar debe tenerse en cuenta que un test puede tener entre 0 y N calendarios. Si no tiene ningún calendario, se podrá ejecutar en cualquier momento. Por otro lado, puede tener tantos calendarios como el usuario quiera, es decir, puede cumplir distintos requisitos. Por ejemplo, ejecutarse de 9h a 18h de lunes a viernes i de 9h a 14h los fines de semana. En ese caso serían dos ventanas temporales distintas, lo que correspondería a dos calendarios. En definitiva, en el código deben contemplarse esas opciones y se ha optado por lo que se explica a continuación.

Para definir los calendarios de un test, en primer lugar se busca si dicho test tiene algún calendario definido. Para ello se mira si tiene algún *IdTest* asignado. Esto se hace con la función *getIdsCalendarisFromIdTest* cuyo código se muestra en la parte superior de la imagen 6.6. Como se puede ver, en caso de no tener ningún calendario asignado, la variable *arrayIdsCalendaris* tendrá longitud 0. Así, la longitud de dicha variable será más adelante la forma de controlar si un test tiene calendarios.

Web Service

La función equivalente en la parte del *Web Service* se muestra en la parte inferior de la imagen 6.6. Tas ejecutar la sentencia *SQL*, se contemplan las distintas posibilidades (entre 0 y N calendarios) así como los posibles errores.

6.2.3.4. Función *getCalendarioTestClassFromIdCalendari*

Agente

Si un test tiene uno o más calendarios, a partir de sus identificadores se obtendrán los campos de cada uno de ellos. Esto se realiza mediante la función *getCalendarioTestClassFromIdCalendari* cuyo código resumido se muestra en la parte superior de la imagen 6.7. Como se puede ver, se envía un único parámetro, *IdCalendari* y se obtienen todos los campos que definen el calendario. Se debe tener en cuanta que ninguno de los campos son son obligatorios, es decir, es posible que un calendario se defina simplemente por la frecuencia de ejecución, con lo que tan sólo se definiría el parámetro *everyXminutes*.

Web Service

En la zona inferior de la imagen 6.7 se puede ver el código resumido de la función *getCalendarioTestClassFromIdCalendari* en la parte del *Web Service*. Como se ve en la imagen, mediante el atributo *tableFields* se encuentra cada uno de los campos de la tabla *Calendaris_Mobile*.

6.2.3.5. Función *getIdsPasosFromIdTest*

Agente

Una vez se han obtenido los calendarios de cada test, se procede a obtener los identificadores de cada paso que tienen. Para ello se utiliza la función *getIdsPasosFromIdTest*, cuyo código se muestra en la parte superior de la imagen 6.8.

Web Service

El código equivalente del *Web Service* se muestra en la parte inferior de la imagen 6.8. Como se puede ver, a partir del identificador del test, *IdTest*, se va a buscar los *Id* a la tabla *Passos_Mobile*. En caso de no encontrar ningún campo *Id* que se corresponda con el *IdTest*, significaría que el test no tiene ningún paso, lo cual no tiene sentido ya que el test no haría nada, así que se reportaría el error correspondiente.

6.2.3.6. Función *getTestBDFieldsFromIdTest*

Agente

A continuación se procede a obtener le resto de atributos que definen al test y que se encuentran en la tabla *Test_Mobile* de la Base de Datos. Estos pasos los realiza la función *getTestBDFieldsFromIdTest* cuyo código se muestra en la imagen 6.9.

Cabe mencionar que de entre los campos de la tabla *Test_Mobile*, *DataAlta* y *DataModificacio* no se extraen puesto que no son de utilidad para el agente, estos dos campos están pensados con otro objetivo.

Web Service

El código encargado de encontrar los campos de la Base de Datos es el correspondiente al *Web Service* y se muestra de forma resumida en la parte inferior de la imagen 6.9.

6.2.3.7. Función *getSOdeviceFromIdTerminal*

Agente

Tras todas las funciones anteriores, tan sólo queda inicializar un atributo de la clase *Test.java* del agente, el llamado *SOdevice*. Para ello se hace se parte del atributo *IdTerminal* que se ha obtenido en la función anterior y se va a buscar a la Base de Datos qué Sistema Operativo corresponde con dicho terminal. Esto lo realiza el método *getSOdeviceFromIdTerminal* de la parte superior de la imagen 6.10.

Web Service

En el *Web Service* la función encargada a atender la petición anterior es la que se muestra en la parte inferior de la imagen 6.10.

6.2.4. Agente y *Web Service*: funciones relacionadas con el envío de resultados a la Base de Datos

Para enviar los resultados de la ejecución de un test a la Base de Datos, éstos se envían al *Web Service* y él se encarga de insertarlos en la tabla *Resultats_Mobile* de la Base de Datos.

Agente

Como se pudo ver en la sección 5.2.2.5 del capítulo 5 cuando se explicaba el código de la imagen 5.20, el agente envía los resultados mediante la función *sendResultsToBBDD* de la clase *WebService.java*. Como se muestra en la imagen 6.11, de igual forma que el resto de métodos que se comunican con el *Web Service*, en primer lugar se inicializa la variable *urlAttributes* con los datos a enviar. En este caso, puesto que la creación de la variable es algo compleja dado que depende de cómo haya ido el test, este paso se ha delegado a la función *createSetMobileTestResultsURL*.

Como se puede ver en la imagen 6.12, la función *createSetMobileTestResultsURL* obtiene el objeto *ResultadosTest.java* como parámetro y, a partir de éste, crea un *String* que constituye los campos XML que se envían al *Web Service*. Para ello se debe tener en cuenta que los atributos de la clase *ResultadosTest.java* tienen una relación directa con los campos de la tabla *Resultats_Mobile* de la Base de Datos. Esa relación se muestra a continuación (a la izquierda atributos de la clase *ResultadosTest.java* y a la derecha campos de la tabla *Resultats_Mobile*):

- String idRobot $\Longleftrightarrow$ IdRobot
- String idTerminal $\Longleftrightarrow$ IdTerminal
- String idTest $\Longleftrightarrow$ IdTest
- String idGrupTest $\Longleftrightarrow$ IdGrupTest
- ArrayList<String>idPasos $\Longleftrightarrow$ IdPas
- String idError $\Longleftrightarrow$ IdError
- ArrayList<String>dataPasos $\Longleftrightarrow$ Data
- ArrayList<String>tempsPasos $\Longleftrightarrow$ TempsPas
- String nameImatge $\Longleftrightarrow$ PathImatge
- ArrayList<String>namesCapturasPrevias $\Longleftrightarrow$ PathCapturaPrevia
- String nameCapturaError $\Longleftrightarrow$ PathCapturaError

Como se puede ver, en las relaciones anteriores faltan 3 atributos de la clase *ResultadosTest.java*. A continuación se explica el motivo:

- boolean testOk: no es necesario como campo en la Base de Datos ya que, para saber si un test ha fallado o no, se recurre al campo *IdError*. Este campo sirve para saber qué tipo de error ha tenido el test en caso de haber fallado, pero a su vez, si su valor es 0 implica que el test ha ido bien.
- int numExecutedSteps: este atributo se usa en la función *createSetMobileTestResultsURL* para controlar cuántos pasos se han ejecutado y en consecuencia poder crear el *String* con los datos de cada paso.
- boolean sendResults: con este atributo se controla si los resultados del test se deben enviar o no. Por ejemplo, si se detecta que un test ha fallado debido a que el móvil no tenía acceso a Intenet, el error no ha sido debido a un fallo de la aplicación que se está monitorizando, así que el resultado no es de interés para el cliente.

Volviendo a la función *createSetMobileTestResultsURL*, para comprender su código se debe tener en cuenta que la inserción de resultados en la tabla *Resultats_Mobile* de la Base de Datos no se hace test a test, sino paso a paso. Es decir, si un test ha ejecutado 4 pasos, el resultado de dicho test implicará 4 filas nuevas en la tabla . Es por ello que en la imagen 6.12 se han resaltado dos partes diferenciadas del código de la función *createSetMobileTestResultsURL*: el rectángulo rosa marca los atributos que se envían una sola vez al *Web Service* puesto que son constantes para todos los pasos, mientras que el rectángulo verde marca los atributos que cambian en cada paso y que por ello deben obtenerse uno a uno a partir de los objetos *ArrayList* correspondientes.

Un ejemplo de los atributos XML que podría crear esta función sería el siguiente:

```
IdRobot=‘1’ IdTerminal=‘2’ IdTest=‘7’ IdGrupTest=‘1’ IdError=‘0’ PathImatge=‘’
PathCapturaError=‘’ TestOk=‘true’ NumExecutedSteps=‘3’ IdPas1=‘20’ IdPas2=‘21’
IdPas3=‘22’ DataPas1=‘20140212135049’ DataPas2=‘20140212135053’
DataPas3=‘20140212135059’ TempsPas1=‘4.388’ TempsPas2=‘5.750’ TempsPas3=‘4.496’
PathCapturaPrevia1=‘1-CaixaAndroid-20140212_135048.jpg’
PathCapturaPrevia2=‘1-CaixaAndroid-20140212_135052.jpg’
PathCapturaPrevia3=‘1-CaixaAndroid-20140212_135058.jpg’
```

En este caso se habría ejecutado un test de 3 pasos y éste habría finalizado correctamente. Por eso los atributos *PathImatge* y *PathCapturaError* no tienen valor alguno, ya que sólo se utilizan en caso de que el test falle para indicar cuál era la imagen que se buscaba y qué captura de error se ha obtenido.

Siguiendo con la función *sendResultsToBBDD*, como se puede ver en la imagen 6.11, cuando ya ha obtenido los atributos XML a enviar mediante *createSetMobileTestResultsURL*, hace uso del método *sendURLAndGetNodeResponse* para comunicarse con el *Web Service* y obtener la respuesta de éste. Según la lectura de esta respuesta determina si los resultados se han insertando correctamente o no en la Base de Datos.

Web Service

En la parte del *Web Service* la función encargada de recibir los resultados e insertarlos en la Base de Datos es *setMobileTestResults*, cuyo código resumido se muestra en la imagen 6.13. Como se puede ver, se han marcado distintas zonas del código en colores: el rectángulo rosa y verde marcan lo mismo que marcaban en la imagen 6.12: el rosa los atributos constantes que siempre llegan y en verde los que dependen de cada paso.

De esta forma, tras interpretar todos los atributos recibidos, se procede a prepararlos en una sentencia SQL para poder enviarlos a la Base de Datos. Esto se hace en el rectángulo azul. Las flechas azules de la imagen señalan los métodos que preparan dichas sentencias, uno en caso de que el test se haya ejecutado correctamente y otro en caso de que haya fallado. Para ello se utilizan dos funciones distintas:

- *getSentenciaKoTablaResultats*: Este método se se utiliza en caso de que el test haya fallado ya que el último paso habrá que marcarlo como erróneo. El código se muestra en la imagen 6.14.
- *getSentenciaOkTablaResultats*: Este método se se utiliza para los pasos que se han ejecutado correctamente. El código se muestra en la imagen 6.15.

Siguiendo el código de la imagen 6.13, en la zona señalada en rojo, una vez se ha obtenido la sentencia correspondiente mediante las funciones *getSentenciaKoTablaResultats* o *getSentenciaOk-TablaResultats*, finalmente se utiliza la función *sendMobileResultsToBD* para insertar los resultados del paso en la Base de Datos. Tras enviar el resultado de ese paso se procede a hacer lo mismo con el resto de pasos hasta que finalice el bucle *for*.

El código de la función *sendMobileResultsToBD*, que es quien realmente envía los resultados a la Base de Datos, se muestra en la imagen 6.16.

```
private Node sendURLAndGetNodeResponse(String nodeName, String urlAttributes) {

    Node node = null;
    String variableURL = "<Peticions IdRobot='0'><" + nodeName + " "
            + urlAttributes + "></" + nodeName + "></Peticions>";
    String errorToWriteInCatch = "ERROR: No se ha podido leer la respuesta de "
            + "la Base de Datos";

    try {
        String encodedVariableURL = URLEncoder.encode(variableURL, "UTF-8");
        URL fullURL = new URL(fixedURL + encodedVariableURL);
        Log.info("Enviando URL al Web Service: " + fixedURL + variableURL);
        URLConnection conexionURL = fullURL.openConnection();
        BufferedReader reader = new BufferedReader(new InputStreamReader(
                conexionURL.getInputStream()));
        String line;
        StringBuffer buffer = new StringBuffer();

        while ((line = reader.readLine()) != null) {
            if (!line.equals("")) { // así me evito las líneas inútiles
                Log.info(line + "\n");
                buffer.append(line + "\n");
            }
        }

        String xmlString = buffer.toString();
        InputStream xmlInputStream = new ByteArrayInputStream(
                xmlString.getBytes("UTF-8"));
        Document document = DocumentBuilderFactory.newInstance()
                .newDocumentBuilder().parse(xmlInputStream);
        NodeList nodeListResposta = document
                .getElementsByTagName("Resposta");
        Node nodeResposta = nodeListResposta.item(0); // nodo "Resposta"

        if (nodeResposta.getAttributes().getNamedItem("value")
                .getNodeValue().equals("OK")) {

            NodeList childrenNodeList = nodeResposta.getChildNodes();

            for (int i = 0; i < childrenNodeList.getLength(); i++) {

                Node childNode = childrenNodeList.item(i);

                if (childNode.getNodeName().equals(nodeName)) {
                    node = childNode;
                    break;
                }
            }
        }

        ...
```

Figura 6.2: Agente: Código resumido de la función *sendURLAndGetNodeResponse* de la clase *WebService.java*.

```
if(nd.getNodeName().equals("GetIdRobotFromNameRobot")){
    getIdRobotFromNameRobot(nd, resposta, lPoolBD);
}
if(nd.getNodeName().equals("GetIdsTestsFromIdRobot")){
    getIdsTestsFromIdRobot(nd, resposta, lPoolBD);
}
if(nd.getNodeName().equals("GetIdsCalendarisFromIdTest")){
    getIdsCalendarisFromIdTest(nd, resposta, lPoolBD);
}
if(nd.getNodeName().equals("GetCalendarioTestClassFromIdCalendari")){
    getCalendarioTestClassFromIdCalendari(nd, resposta, lPoolBD);
}
if(nd.getNodeName().equals("GetIdsPasosFromIdTest")){
    getIdsPasosFromIdTest(nd, resposta, lPoolBD);
}
if(nd.getNodeName().equals("GetTestBDFieldsFromIdTest")){
    getTestBDFieldsFromIdTest(nd, resposta, lPoolBD);
}
if(nd.getNodeName().equals("GetSOdeviceFromIdTerminal")){
    getSOdeviceFromIdTerminal(nd, resposta, lPoolBD);
}
if(nd.getNodeName().equals("SetMobileTestResults")){
    setMobileTestResults(nd, resposta, lPoolBD);
}
```

Figura 6.3: *Web Service*: Lista de *if*'s para identificar el método a ejecutar.

```java
public String getIdRobotFromNameRobot(String nameRobot) {

    String idRobot = null;
    String urlAttributes = "NameRobot='" + nameRobot + "'";
    Node node = sendURLAndGetNodeResponse("GetIdRobotFromNameRobot",
            urlAttributes);
    NamedNodeMap attributes = node.getAttributes();

    if (attributes.getNamedItem("value").getNodeValue().equals("OK")) {
        idRobot = attributes.getNamedItem("IdRobot").getNodeValue();
    } else {
        if (attributes.getNamedItem("RobotsFound") != null) {
            String stringRobotsFound = attributes.getNamedItem(
                    "RobotsFound").getNodeValue();
            int intStringRobotsFound = Integer.parseInt(stringRobotsFound);
            if (intStringRobotsFound == 0) {
                Log.error("ERROR: El nombre de robot "
                        + nameRobot
                        + " no existe en la Base de Datos. No se ha obtenido"
                        + " el IdRobot.");
            } else {
                Log.error("ERROR: El nombre de robot "
                        + nameRobot
                        + " aparece más de una vez en la Base de Datos. No se"
                        + " ha obtenido el IdRobot.");
            }
        } else {
            Log.error("ERROR: El Web Service no ha recibido correctamente el"
                    + " atributo \"NameRobot\"");
        }
    }
    return idRobot;
}
```

```java
public void getIdRobotFromNameRobot(Node node, Document documentResposta,
        Pool poolBD) {

    NamedNodeMap attributes = node.getAttributes();
    String nameRobot = "NULL";
    Node namedItem;
    Element element = (Element) documentResposta
            .createElement("GetIdRobotFromNameRobot");

    if ((namedItem = attributes.getNamedItem("NameRobot")) != null) {
        nameRobot = namedItem.getNodeValue();
        String lstrConsulta = "SELECT Distinct Id FROM Robots_Mobile where Nom='"
                + nameRobot + "'";
        oLog.debug("PeticionsXML::getIdRobotFromNameRobot - sentenciaBD: \""
                + lstrConsulta + " \"");
        BBDD oBBDD = new BBDD();
        try {
            oBBDD = poolBD.getConnectionBBDD();
            oBBDD.abrirResultSet(lstrConsulta);
            int robotsFound = 0;
            while (oBBDD.siguiente()) {
                robotsFound++;
                element.setAttribute("IdRobot", oBBDD.obtenerCadena(1));
            }
            if (robotsFound == 1) {
                element.setAttribute("value", "OK");
            } else {
                element.setAttribute("value", "KO");
                element.setAttribute("RobotsFound",
                        Integer.toString(robotsFound));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            oBBDD.cerrarResultSet();
            oBBDD.finalizarConexion();
        }
    } else {
        oLog.debug("ERROR: No se ha recibido correctamente el nombre del robot \n");
        element.setAttribute("value", "KO");
    }
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.4: Código de las funciones *getIdRobotFromNameRobot* (arriba clase *WebService.java* del agente y abajo *Web Service*).

```java
public ArrayList<String> getIdsTestsFromIdRobot(String idRobot) {

    ArrayList<String> arrayIdTests = new ArrayList<String>();
    String urlAttributes = "IdRobot='" + idRobot + "'";
    Node node = sendURLAndGetNodeResponse("GetIdsTestsFromIdRobot",
            urlAttributes);
    NamedNodeMap attributes = node.getAttributes();

    if (attributes.getNamedItem("value").getNodeValue().equals("OK")) {
        for (int i = 0; i < attributes.getLength(); i++) {
            if (attributes.item(i).getNodeName().contains("IdTest")) {
                arrayIdTests.add(attributes.item(i).getNodeValue());
            }
        }
        return arrayIdTests;
    } else {
        if (attributes.getNamedItem("TestsFound") != null) {
            Log.error("ERROR: No se ha definido ningún test para el robot con Id="
                    + idRobot);
        } else {
            Log.error("ERROR: El Web Service no ha recibido correctamente"
                    + " el atributo \"IdRobot\"");
        }
        return null;
    }
}
```

```java
public void getIdsTestsFromIdRobot(Node node, Document documentResposta,
        Pool poolBD) {

    NamedNodeMap attributes = node.getAttributes();
    String idRobot = "NULL";
    Node namedItem;
    Element element = (Element) documentResposta
            .createElement("GetIdsTestsFromIdRobot");

    if ((namedItem = attributes.getNamedItem("IdRobot")) != null) {
        idRobot = namedItem.getNodeValue();
        String lstrConsulta = "SELECT Distinct IdTest FROM Robot_Test_Mobile "
                + "where IdRobot='" + idRobot + "'";
        oLog.debug("PeticionsXML::getIdsTestsFromIdRobot - sentenciaBD: \""
                + lstrConsulta + " \"");
        BBDD oBBDD = new BBDD();
        try {
            oBBDD = poolBD.getConnectionBBDD();
            oBBDD.abrirResultSet(lstrConsulta);
            int testsFound = 0;
            while (oBBDD.siguiente()) {
                testsFound++;
                element.setAttribute("IdTest" + testsFound,
                        oBBDD.obtenerCadena(1));
            }
            if (testsFound > 0) {
                element.setAttribute("value", "OK");
            } else {
                element.setAttribute("value", "KO");
                element.setAttribute("TestsFound",
                        Integer.toString(testsFound));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            oBBDD.cerrarResultSet();
            oBBDD.finalizarConexion();
        }
    } else {
        oLog.debug("ERROR: No se ha recibido correctamente el id del robot \n");
        element.setAttribute("value", "KO");
    }
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.5: Código de las funciones *getIdsTestsFromIdRobot* (arriba clase *WebService.java* del agente y abajo *Web Service*).

```java
public ArrayList<String> getIdsCalendarisFromIdTest(String idTest) {

    ArrayList<String> arrayIdsCalendaris = new ArrayList<String>();
    String urlAttributes = "IdTest='" + idTest + "'";
    Node node = sendURLAndGetNodeResponse("GetIdsCalendarisFromIdTest",
            urlAttributes);
    NamedNodeMap attributes = node.getAttributes();

    if (attributes.getNamedItem("value").getNodeValue().equals("OK")) {
        String stringCalendarisFound = attributes.getNamedItem(
                "CalendarisFound").getNodeValue();
        int intStringCalendarisFound = Integer
                .parseInt(stringCalendarisFound);
        if (intStringCalendarisFound > 0) {
            for (int i = 0; i < attributes.getLength(); i++) {
                if (attributes.item(i).getNodeName()
                        .contains("IdCalendari")) {
                    arrayIdsCalendaris.add(attributes.item(i)
                            .getNodeValue());
                }
            }
        }
        return arrayIdsCalendaris;
    } else {
        Log.error("ERROR: El Web Service no ha recibido correctamente"
                + " el atributo \"IdTest\"");
        return null;
    }
}
```

```java
public void getIdsCalendarisFromIdTest(Node node,
        Document documentResposta, Pool poolBD) {

    NamedNodeMap attributes = node.getAttributes();
    String idTest = "NULL";
    Node namedItem;
    Element element = (Element) documentResposta
            .createElement("GetIdsCalendarisFromIdTest");

    if ((namedItem = attributes.getNamedItem("IdTest")) != null) {
        idTest = namedItem.getNodeValue();
        String lstrConsulta = "SELECT Distinct IdCalendari FROM "
                + "Test_Calendari_Mobile where IdTest='"
                + idTest + "'";
        oLog.debug("PeticionsXML::getIdsCalendarisFromIdTest - sentenciaBD: \""
                + lstrConsulta + " \"");
        BBDD oBBDD = new BBDD();
        try {
            oBBDD = poolBD.getConnectionBBDD();
            oBBDD.abrirResultSet(lstrConsulta);
            int calendarisFound = 0;
            while (oBBDD.siguiente()) {
                calendarisFound++;
                element.setAttribute("IdCalendari" + calendarisFound,
                        oBBDD.obtenerCadena(1));
            }
            element.setAttribute("value", "OK");
            element.setAttribute("CalendarisFound",
                    Integer.toString(calendarisFound));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            oBBDD.cerrarResultSet();
            oBBDD.finalizarConexion();
        }
    } else {
        oLog.debug("ERROR: No se ha recibido correctamente el id del test \n");
        element.setAttribute("value", "KO");
    }
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.6: Código de las funciones *getIdsCalendarisFromIdTest* (arriba clase *WebService.java* del agente y abajo *Web Service*).

```java
public CalendarioTest getCalendarioTestClassFromIdCalendari(
        String idCalendari) {

    CalendarioTest calendarioTest = new CalendarioTest();

    String urlAttributes = "IdCalendari='" + idCalendari + "'";
    Node node = sendURLAndGetNodeResponse(
            "GetCalendarioTestClassFromIdCalendari", urlAttributes);
    NamedNodeMap attributes = node.getAttributes();
    Node namedItem;

    if (attributes.getNamedItem("value").getNodeValue().equals("OK")) {
        if ((namedItem = attributes.getNamedItem("StartDate")) != null) {
            calendarioTest.setStartDate(namedItem.getNodeValue());
        }
        if ((namedItem = attributes.getNamedItem("EndDate")) != null) {
            calendarioTest.setEndDate(namedItem.getNodeValue());
        }
        if ((namedItem = attributes.getNamedItem("StartHour")) != null) {
            calendarioTest.setStartHour(namedItem.getNodeValue());
        }
        if ((namedItem = attributes.getNamedItem("EndHour")) != null) {
            calendarioTest.setEndHour(namedItem.getNodeValue());
        }
        if ((namedItem = attributes.getNamedItem("DaysOfWeek")) != null) {
            calendarioTest.setDaysOfWeek(namedItem.getNodeValue());
        }
        if ((namedItem = attributes.getNamedItem("DaysOfMonth")) != null) {
            calendarioTest.setDaysOfMonth(namedItem.getNodeValue());
        }
        if ((namedItem = attributes.getNamedItem("EveryXminutes")) != null) {
            calendarioTest.setEveryXMinutes(namedItem.getNodeValue());
        }
        return calendarioTest;
    } else {
        ...
    }
}
```

```java
public void getCalendarioTestClassFromIdCalendari(Node node,
        Document documentResposta, Pool poolBD) {

    NamedNodeMap attributes = node.getAttributes();
    Element element = (Element) documentResposta
            .createElement("GetCalendarioTestClassFromIdCalendari");
    String idCalendari = "NULL";
    Node namedItem;
    ArrayList<String> tableFields = new ArrayList<String>();
    tableFields.add("StartDate");
    tableFields.add("EndDate");
    tableFields.add("StartHour");
    tableFields.add("EndHour");
    tableFields.add("DaysOfWeek");
    tableFields.add("DaysOfMonth");
    tableFields.add("EveryXminutes");

    if ((namedItem = attributes.getNamedItem("IdCalendari")) != null) {

        element.setAttribute("value", "OK");

        idCalendari = namedItem.getNodeValue();
        String tableField;
        String lstrConsulta;

        for (int i = 0; i < tableFields.size(); i++) {
            tableField = tableFields.get(i);
            lstrConsulta = "SELECT " + tableField
                    + " FROM Calendaris_Mobile WHERE Id=" + idCalendari;
            oLog.debug("PeticionsXML::getCalendarioTestClassFromIdCalendari"
                    + " - sentenciaBD: \"" + lstrConsulta + " \"");
            BBDD oBBDD = new BBDD();
            try {
                oBBDD = poolBD.getConnectionBBDD();
                oBBDD.abrirResultSet(lstrConsulta);
                if (oBBDD.siguiente()) {
                    element.setAttribute(tableField, oBBDD.obtenerCadena(1));
                } else {
                    oLog.debug("ERROR: En la Base de Datos no existe ningun"
                            + " calendario definido con el id "
                            + idCalendari + "\n");
                    element.setAttribute("value", "KO");
                    element.setAttribute("error", "IdCalendari");
                    break;// que salga del "for"
                }
            }
            ...
        }
    }
    ...
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.7: Código resumido de la funciones *getCalendarioTestClassFromIdCalendari* (arriba clase *WebService.java* del agente y abajo *Web Service*).

```java
private ArrayList<String> getIdsPasosFromIdTest(String idTest) {

    ArrayList<String> arrayIdsPasos = new ArrayList<String>();

    String urlAttributes = "IdTest='" + idTest + "'";
    Node node = sendURLAndGetNodeResponse("GetIdsPasosFromIdTest",
            urlAttributes);
    NamedNodeMap attributes = node.getAttributes();

    if (attributes.getNamedItem("value").getNodeValue().equals("OK")) {
        String stringPasosFound = attributes.getNamedItem("PasosFound")
                .getNodeValue();
        int intStringPasosFound = Integer.parseInt(stringPasosFound);
        if (intStringPasosFound > 0) {
            for (int i = 0; i < attributes.getLength(); i++) {
                if (attributes.item(i).getNodeName().contains("IdPas")) {
                    arrayIdsPasos.add(attributes.item(i).getNodeValue());
                }
            }
            return arrayIdsPasos;
        } else {
            Log.error("ERROR: El test con Id=\"" + idTest
                    + "\" no tiene definido ningun paso");
            return null;
        }
    } else {
        Log.error("ERROR: El Web Service no ha recibido correctamente "
                + "el atributo \"IdTest\"");
        return null;
    }
}
```

```java
public void getIdsPasosFromIdTest(Node node, Document documentResposta,
        Pool poolBD) {

    NamedNodeMap attributes = node.getAttributes();
    String idTest = "NULL";
    Node namedItem;
    Element element = (Element) documentResposta
            .createElement("GetIdsPasosFromIdTest");

    if ((namedItem = attributes.getNamedItem("IdTest")) != null) {
        idTest = namedItem.getNodeValue();
        String lstrConsulta = "SELECT Distinct Id FROM Passos_Mobile "
                + "where IdTest='"
                + idTest + "'";
        oLog.debug("PeticionsXML::getIdsPasosFromIdTest - sentenciaBD: \""
                + lstrConsulta + " \"");
        BBDD oBBDD = new BBDD();
        try {
            oBBDD = poolBD.getConnectionBBDD();
            oBBDD.abrirResultSet(lstrConsulta);
            int pasosFound = 0;
            while (oBBDD.siguiente()) {
                pasosFound++;
                element.setAttribute("IdPas" + pasosFound,
                        oBBDD.obtenerCadena(1));
            }
            element.setAttribute("value", "OK");
            element.setAttribute("PasosFound", Integer.toString(pasosFound));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            oBBDD.cerrarResultSet();
            oBBDD.finalizarConexion();
        }
    } else {
        oLog.debug("ERROR: No se ha recibido correctamente el id del test \n");
        element.setAttribute("value", "KO");
    }
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.8: Código de las funciones *getIdsPasosFromIdTest* (arriba clase *WebService.java* del agente y abajo *Web Service*).

```java
private Test getTestBDFieldsFromIdTest(String idTest) {

    Test test = new Test();

    String urlAttributes = "IdTest='" + idTest + "'";
    Node node = sendURLAndGetNodeResponse("GetTestBDFieldsFromIdTest",
            urlAttributes);
    NamedNodeMap attributes = node.getAttributes();
    Node namedItem;

    if (attributes.getNamedItem("value").getNodeValue().equals("OK")) {
        if ((namedItem = attributes.getNamedItem("Nom")) != null) {
            test.setNameTest(namedItem.getNodeValue());
        }
        if ((namedItem = attributes.getNamedItem("Reintents")) != null) {
            int retries = Integer.parseInt(namedItem.getNodeValue());
            test.setRetries(retries);
        }
        if ((namedItem = attributes.getNamedItem("IdTerminal")) != null) {
            test.setIdTerminal(namedItem.getNodeValue());
        }
        return test;
    } else {
        if (attributes.getNamedItem("error").getNodeValue().equals("null")) {
            Log.error("ERROR: El Web Service no ha recibido correctamente"
                    + " la variable \"IdTest\"");
        } else if (attributes.getNamedItem("error").getNodeValue()
                .equals("IdTest")) {
            Log.error("ERROR: En la Base de Datos no existe ningun test"
                    + " definido con el id "
                    + idTest);
        }
        return null;
    }
}
```

```java
public void getTestBDFieldsFromIdTest(Node node, Document documentResposta,
        Pool poolBD) {

    NamedNodeMap attributes = node.getAttributes();
    Element element = (Element) documentResposta
            .createElement("GetTestBDFieldsFromIdTest");
    String idTest = "NULL";
    Node namedItem;
    ArrayList<String> tableFields = new ArrayList<String>();
    tableFields.add("Nom");
    tableFields.add("Reintents");
    tableFields.add("IdTerminal");

    if ((namedItem = attributes.getNamedItem("IdTest")) != null) {

        element.setAttribute("value", "OK");

        idTest = namedItem.getNodeValue();
        String tableField;
        String lstrConsulta;

        for (int i = 0; i < tableFields.size(); i++) {
            tableField = tableFields.get(i);
            lstrConsulta = "SELECT " + tableField
                    + " FROM Tests_Mobile WHERE Id=" + idTest;
            oLog.debug("PeticionsXML::getTestBDFieldsFromIdTest - "
                    + "sentenciaBD: \"" + lstrConsulta + " \"");
            BBDD oBBDD = new BBDD();
            try {
                oBBDD = poolBD.getConnectionBBDD();
                oBBDD.abrirResultSet(lstrConsulta);
                if (oBBDD.siguiente()) {
                    element.setAttribute(tableField, oBBDD.obtenerCadena(1));
                }
                ...
            }
            ...
        }
    }
    ...
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.9: Código de las funciones *getTestBDFieldsFromIdTest* (arriba clase *WebService.java* del agente y abajo *Web Service*).

```java
private String getSOdeviceFromIdTerminal(String idTerminal) {

    String SOdevice = null;
    String urlAttributes = "IdTerminal='" + idTerminal + "'";
    Node node = sendURLAndGetNodeResponse("GetSOdeviceFromIdTerminal",
            urlAttributes);
    NamedNodeMap attributes = node.getAttributes();

    if (attributes.getNamedItem("value").getNodeValue().equals("OK")) {
        SOdevice = attributes.getNamedItem("SOdevice").getNodeValue();
    } else {
        if (attributes.getNamedItem("TerminalsFound") != null) {
            Log.error("ERROR: El id de terminal "
                    + idTerminal + " no existe en la Base de Datos."
                    + " No se ha obtenido el SOdevice.");
        } else {
            Log.error("ERROR: El Web Service no ha recibido "
                    + "correctamente el atributo \"IdTerminal\"");
        }
    }
    return SOdevice;
}
```

```java
public void getSOdeviceFromIdTerminal(Node node, Document documentResposta,
        Pool poolBD) {

    NamedNodeMap attributes = node.getAttributes();
    String idTerminal = "NULL";
    Node namedItem;
    Element element = (Element) documentResposta
            .createElement("GetSOdeviceFromIdTerminal");

    if ((namedItem = attributes.getNamedItem("IdTerminal")) != null) {
        idTerminal = namedItem.getNodeValue();
        String lstrConsulta = "SELECT Distinct SOdevice FROM Terminals_Mobile"
                + " where Id='" + idTerminal + "'";
        oLog.debug("PeticionsXML::getSOdeviceFromIdTerminal - sentenciaBD: \""
                + lstrConsulta + " \"");
        BBDD oBBDD = new BBDD();
        try {
            oBBDD = poolBD.getConnectionBBDD();
            oBBDD.abrirResultSet(lstrConsulta);
            if (oBBDD.siguiente()) {
                element.setAttribute("value", "OK");
                element.setAttribute("SOdevice", oBBDD.obtenerCadena(1));
            } else {
                element.setAttribute("value", "KO");
                element.setAttribute("TerminalsFound", Integer.toString(0));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            oBBDD.cerrarResultSet();
            oBBDD.finalizarConexion();
        }
    } else {
        oLog.debug("ERROR: No se ha recibido correctamente el id del terminal \n")
        element.setAttribute("value", "KO");
    }
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.10: Código de las funciones *getSOdeviceFromIdTerminal* (arriba clase *WebService.java* del agente y abajo *Web Service*).

```java
public boolean sendResultsToBBDD(ResultadosTest resultadosTest) {

    Log.info("Enviando resultados a la Base de Datos");
    boolean everythingOK = false;
    String urlAttributes = createSetMobileTestResultsURL(resultadosTest);
    Node node = sendURLAndGetNodeResponse("SetMobileTestResults",
            urlAttributes);

    if (node != null) {

        Node nodeValue = node.getAttributes().item(0);

        if (nodeValue.getNodeName().equals("value")
                && nodeValue.getNodeValue().equals("OK")) {
            everythingOK = true;
        } else if (nodeValue.getNodeName().equals("value")
                && nodeValue.getNodeValue().equals("KO")) {
            Log.error("ERROR al enviar la sentencia SQL a la Base de "
                    + "datos. No se han enviado los resultados.");
        }
    }
    return everythingOK;
}
```

Figura 6.11: Código de la función *sendResultsToBBDD* de la clase *WebService.java* del agente.

```java
private String createSetMobileTestResultsURL ( ResultadosTest resultadosTest ) {

    int numExecutedSteps = resultadosTest.getNumExecutedSteps();

    String idRobot = resultadosTest.getIdRobot();
    String idTerminal = resultadosTest.getIdTerminal();
    String idTest = resultadosTest.getIdTest();
    String idGrupTest = resultadosTest.getIdGrupTest();
    String idError = resultadosTest.getIdError();
    String nameImatge = resultadosTest.getNameImatge();
    String nameCapturaError = resultadosTest.getNameCapturaError();
    String testOk = new Boolean(resultadosTest.isTestOk()).toString();

    StringBuffer buffer = new StringBuffer();
    buffer.append("IdRobot='" + idRobot + "' IdTerminal='" + idTerminal
            + "' IdTest='" + idTest + "' IdGrupTest='" + idGrupTest
            + "' IdError='" + idError + "' PathImatge='" + nameImatge
            + "' PathCapturaError='" + nameCapturaError + "' TestOk='"
            + testOk + "' NumExecutedSteps='"
            + Integer.toString(numExecutedSteps) + "'");

    String attribute;

    for (int i = 0; i < numExecutedSteps; i++) {
        attribute = resultadosTest.getIdPasos().get(i);
        buffer.append(" " + "IdPas" + Integer.toString(i + 1) + "='" + attribute + "'");

        attribute = resultadosTest.getDataPasos().get(i);
        buffer.append(" " + "DataPas" + Integer.toString(i + 1) + "='" + attribute + "'");

        attribute = resultadosTest.getNamesCapturasPrevias().get(i);
        buffer.append(" " + "PathCapturaPrevia" + Integer.toString(i + 1) + "='" + attribute + "'");

        if (i == numExecutedSteps - 1 && !resultadosTest.isTestOk()) {
            buffer.append(" " + "TempsPas" + Integer.toString(i + 1) + "='0'");
        } else {
            attribute = resultadosTest.getTempsPasos().get(i);
            buffer.append(" " + "TempsPas" + Integer.toString(i + 1) + "='" + attribute + "'");
        }
    }

    String urlAttributes = buffer.toString();

    return urlAttributes;
}
```

Figura 6.12: Código de la función *createSetMobileTestResultsURL* de la clase *WebService.java* del agente.

```
public void setMobileTestResults (Node node, Document documentResposta, Pool poolBD)
{
    NamedNodeMap attributes = node.getAttributes();

    ...

    if ( (namedItem = attributes.getNamedItem ( "IdRobot" )) != null ) {
        idRobot = namedItem.getNodeValue();
    }
    if ( (namedItem = attributes.getNamedItem ( "IdTerminal" )) != null ) {
        idTerminal = namedItem.getNodeValue();
    }
    if ( (namedItem = attributes.getNamedItem ( "IdTest" )) != null ) {
        idTest = namedItem.getNodeValue();
    }
    if ( (namedItem = attributes.getNamedItem ( "IdGrupTest" )) != null ) {
        idGrupTest = namedItem.getNodeValue();
    }
    if ( (namedItem = attributes.getNamedItem ( "IdError" )) != null ) {
        idError = namedItem.getNodeValue();
    }
    if ( (namedItem = attributes.getNamedItem ( "TestOk" )) != null ) {
        testOK = namedItem.getNodeValue();
    }
    if ( (namedItem = attributes.getNamedItem ( "NumExecutedSteps" )) != null ) {
        numExecutedSteps = Integer.parseInt(namedItem.getNodeValue());
    }

    ...

    if ( numExecutedSteps !=0 ) {
        for (int i=1 ; i<=numExecutedSteps ; i++) {
            if ( ( namedItem = attributes.getNamedItem ( "IdPas" + i )) != null ) {
                idPas = namedItem.getNodeValue();
            }
            if ( ( namedItem = attributes.getNamedItem ( "DataPas" + i )) != null ) {
                dataPas = namedItem.getNodeValue();
            }
            if ( ( namedItem = attributes.getNamedItem ( "TempsPas" + i )) != null ) {
                tempsPas = namedItem.getNodeValue();
            }
            if ( ( namedItem = attributes.getNamedItem ( "PathCapturaPrevia" + i )) != null ) {
                pathCapturaPrevia = namedItem.getNodeValue();
            }
            if ( testOK.equals("false") && i==numExecutedSteps ) {

                ...

                if ( (namedItem = attributes.getNamedItem ( "PathImatge" )) != null ) {
                    pathImatge = namedItem.getNodeValue();
                }
                if ( (namedItem = attributes.getNamedItem ( "PathCapturaError" )) != null ) {
                    pathCapturaError = namedItem.getNodeValue();
                }
                sentenciaBD = getSentenciaKoTablaResultats(idRobot, idTerminal, idTest,
                                                           idGrupTest, idPas, idError, dataPas,
                                                           pathImatge, pathCapturaPrevia,
                                                           pathCapturaError);
            }
            else {
                sentenciaBD = getSentenciaOkTablaResultats(idRobot, idTerminal, idTest,
                                                           idGrupTest, idPas, dataPas, tempsPas,
                                                           pathCapturaPrevia);
            }
            sendMobileResultsToBD(documentResposta, poolBD, sentenciaBD);
        }
    }

    ...
}
```

Figura 6.13: *Web Service*: Código resumido de la función *setMobileTestResults*.

```
public String getSentenciaKoTablaResultats(String idRobot,
        String idTerminal, String idTest, String idGrupTest, String idPas,
        String idError, String data, String pathImatge,
        String pathCapturaPrevia, String pathCapturaError) {
    StringBuffer sentenciaBD = new StringBuffer()
            .append("insert into Resultats_Mobile (IdRobot, IdTerminal, IdTest, IdGrupTest,"
                    + " IdPas, IdError, Data, PathImatge, PathCapturaPrevia, PathCapturaError) ")
    sentenciaBD.append("values (");
    sentenciaBD.append(idRobot + ",");
    sentenciaBD.append(idTerminal + ",");
    sentenciaBD.append(idTest + ",");
    sentenciaBD.append(idGrupTest + ",");
    sentenciaBD.append(idPas + ",");
    sentenciaBD.append(idError + ",");
    sentenciaBD.append(data + ",");
    sentenciaBD.append("'" + pathImatge + "',");
    sentenciaBD.append("'" + pathCapturaPrevia + "',");
    sentenciaBD.append("'" + pathCapturaError + "')");
    return sentenciaBD.toString();
}
```

Figura 6.14: *Web Service*: Código de la función *getSentenciaKoTablaResultats*.

```
public String getSentenciaOkTablaResultats(String idRobot,
        String idTerminal, String idTest, String idGrupTest, String idPas,
        String data, String tempsPas, String pathCapturaPrevia) {

    StringBuffer sentenciaBD = new StringBuffer()
            .append("insert into Resultats_Mobile (IdRobot, IdTerminal, IdTest, IdGrupTest,"
                    + " IdPas, IdError, Data, TempsPas, PathCapturaPrevia) ");

    sentenciaBD.append("values (");
    sentenciaBD.append(idRobot + ",");
    sentenciaBD.append(idTerminal + ",");
    sentenciaBD.append(idTest + ",");
    sentenciaBD.append(idGrupTest + ",");
    sentenciaBD.append(idPas + ",");
    sentenciaBD.append("0" + ","); // Para los pasos que han ido bien, idError = 0
    sentenciaBD.append(data + ",");
    sentenciaBD.append(tempsPas + ",");
    sentenciaBD.append("'" + pathCapturaPrevia + "')");
    return sentenciaBD.toString();
}
```

Figura 6.15: *Web Service*: Código de la función *getSentenciaOkTablaResultats*.

```
public void sendMobileResultsToBD(Document documentResposta, Pool poolBD, String sentenciaBD ) {
    oLog.debug("PeticionsXML::sendMobileResultatsToBBDD - sentenciaBD: \"" + sentenciaBD + " \"");

    BBDD objetoBBDD = null;
    boolean sentenciaOK = false;

    try {
        objetoBBDD = poolBD.getConnectionBBDD();
        sentenciaOK = objetoBBDD.ejecutarSentencia(sentenciaBD.toString());
    } finally {
        objetoBBDD.finalizarConexion();
    }

    Element element = (Element)documentResposta.createElement("SetMobileTestResults");

    if (sentenciaOK) {
        element.setAttribute("value","OK");
    }
    else {
        oLog.debug("ERROR al ejecutar la sentencia \"" + sentenciaBD + " \"");
        element.setAttribute("value","KO");
    }
    documentResposta.getFirstChild().appendChild(element);
}
```

Figura 6.16: *Web Service*: Código de la función *sendMobileResultsToBD*.

Capítulo 7

Base de datos

7.1. Introducción

Un elemento importante dentro del diagrama de la arquitectura de este proyecto es la Base de Datos. Gracias a ella se puede almacenar toda la información relativa a la configuración de cada robot, test, paso, usuario, etc., además de, obviamente, registrar los resultados de las ejecuciones de los tests.

En primer lugar, antes de crear la Base de Datos se pensaron las tablas necesarias. Para ello se siguió el diagrama de flujo del agente que previamente se había realizado y se pensó en las tablas que harían falta en casa paso. Además, debían tenerse en cuenta también las relativas a la página web, tales como el control de usuarios que pueden acceder o las tablas con las que hacer las historificaciones periódicas.

En este capítulo no se entrará en detalle de cómo se montó la Base de Datos por considerarse irrelevante para el objetivo del PFC, sino que tan sólo se hablará de las tablas que se crearon y el sentido de cada una.

7.2. Diagrama de base de datos

En el anexo F se adjunta el diagrama de la Base de Datos necesaria en este proyecto. Como se puede ver, existe una tabla llamada *Resultats_Mobile* que, como su nombre indica, almacena los resultados de todas las ejecuciones de los tests. A partir de los datos de esta tabla existe un *Stored Procedure* que se ejecuta periódicamente con el objetivo de agrupar los resultados con temporalidad hora, día y mes para su posterior explotación en la web. Puesto que tanto el *procedure* como las respectivas tablas forman parte de la explotación de resultados y no de la parte innovadora del proyecto, no se considera interesante detallarlo. Así, se hablará del razonamiento de la existencia de todas las otras tablas. Para ello se seguirán los pasos que debe hacer el agente desde que se inicia:

1. En primer lugar, el agente obtiene el nombre asignado a la máquina como nombre del robot a partir del archivo *ConfigAgente.properties*. Con ese nombre se accede a la tabla *Robots_Mobile* con una sentencia SQL como la siguiente:

   SELECT DISTINCT Id FROM Robots_Mobile WHERE Nom='robot_piloto'

   En el ejemplo, el nombre del robot sería "robot_piloto".

2. A partir del *Id* obtenido en la sentencia anterior, se acude a la tabla *Robot_Test_Mobile* para conocer los tests asignados a dicho robot. Para ello se ejecuta una sentencia SQL como la siguiente:

   SELECT DISTINCT IdTest FROM Robot_Test_Mobile WHERE IdRobot='1'

   En el ejemplo, el *Id* del robot sería "1".

3. Tras la sentencia anterior, se habrá obtenido una lista de números que corresponden al campo *Id* de la tabla *Tests_Mobile*. Con ellos se podrá obtener todos los datos relativos a cada test para así inicializar los objetos correspondientes. De esta forma se seguirán, mediante un bucle, los pasos que se comentan a continuación.

   En primer lugar se procede a inicializar el calendario del test, para lo que se accede a la tabla *Test_Calendari_Mobile* con una sentencia como la siguiente:

SELECT DISTINCT IdCalendari FROM Test_ Calendari_ Mobile WHERE IdTest='1'

En el ejemplo, el *Id* del test sería "1". Con esta sentencia se obtienen los posibles calendarios que tiene el test, ya que recordemos que puede no tener ninguno, en cuyo caso se le asigna el calendario por defecto.

4. En caso de obtener uno o más calendarios asignados al test, se procede a determinar los campos que define a cada uno de ellos. Para ello se accede a la tabla *Calendaris_ Mobile* con los *Id*'s obtenidos anteriormente y se obtienen todos los campos de dicha tabla con sentencias SQL como la siguiente:

   SELECT StartDate FROM Calendaris_ Mobile WHERE Id='1'

   En el ejemplo, el *Id* del calendario sería "1" y el campo de la tabla *Calendaris_ Mobile* que se estaría buscando sería *StartDate*. El agente debe, para cada calendario que tenga el test, encontrar cada uno de los campos del calendario. Con lo que la sentencia anterior se repetiría con el resto de campos de la tabla *Calendaris_ Mobile* y eso se haría para todos los calendarios del test.

5. Tras haber definido los calendarios de los tests, se procede a obtener los pasos de cada uno. Para ello se accede a la tabla *Passos_ Mobile* de la Base de Datos de la siguiente forma:

   SELECT DISTINCT Id FROM Passos_ Mobile WHERE IdTest='1'

   En el ejemplo, el *Id* del test sería "1".

6. Una vez creados tanto los calendarios como los pasos, se accede a la tabla *Tests_ Mobile* para obtener los campos que definen al test que todavía no se tienen. Para ello se ejecutaría:

   SELECT Nom FROM Tests_ Mobile WHERE IdTest='1'

   En el ejemplo, el *Id* del test sería "1" y el campo de la tabla *Tests_ Mobile* que se estaría buscando sería *Nom*, con lo que se ejecutaría una sentencia para cada campo de la tabla *Tests_ Mobile*.

7. Finalmente, en el proceso de inicialización del agente tan sólo queda un paso que esté relacionado con la Base de Datos, que es el de obtener el Sistema Operativo del móvil en el que se ejecuta cada test. Esto se hace accediendo a la tabla *Terminals_ Mobile* a partir del *IdTerminal* que se ha obtenido en el paso anterior:

   SELECT SOdevice FROM Terminals_ Mobile WHERE Id='1'

   En el ejemplo, el *Id* del terminal sería "1".

Tras todos esos accesos a Base de Datos, el agente habría inicializado los objetos Java y procedería a descargar los tests mediante FTP. En ese momento empezarían las ejecuciones y después de cada una de ellas se enviarían los resultados a la Base de Datos. Para ello se recuerda que los resultados se envían paso a paso y no test a test, es decir, que cada paso que se ejecuta en un test genera una nueva entrada en la tabla *Resultats_ Mobile*. De esta forma, los resultados se insertan

con sentencias como la siguiente:

INSERT INTO Resultats_Mobile (IdRobot , IdTerminal , IdTest , IdGrupTest , IdPas , IdError , Data , TempsPas , PathCapturaPrevia) VALUES (1 , 2 , 7 , 372 , 17 , 0 , '20140220002939' , 4.072 , '1-CaixaAndroid-20140220_002938.jpg')

En la imagen 7.1 se puede ver la entrada que generaría en la tabla *Resultats_Mobile* la sentencia de ejemplo anterior .

Id	IdRobot	IdTerminal	IdTest	IdGrupTest	IdPas	IdError	Data	TempsPas	PathImatge	PathCapturaPrevia	PathCapturaError
394266	1	2	7	372	17	0	20140220002939	4,072	NULL	1-CaixaAndroid-20140220_002938.jpg	NULL

Figura 7.1: Base de Datos: Ejemplo del contenido de la tabla Resultats_Mobile para un paso finalizado correctamente.

Como se puede ver tanto en la sentencia como en la imagen, la fecha que se indica en el campo *Data* sigue un formato de tipo *yyyyMMddhhmmss* especificando año, mes, día, hora, minuto y segundo, por ese orden, en el que se ha iniciado el paso. También puede verse que los tiempos de los pasos se dan con una precisión de milisegundos, así como el formato del nombre que se le asigna a las imágenes que se envían al servidor: *idRobot-nombreTest-yyyymmdd_hhmmss.jpg*. Ese formato contiene, en primer lugar, el *Id* del robot donde se ejecuta el test. Tras él, el nombre del test y, finalmente, la fecha del momento en el que se toma la imagen. Con este formato se consigue evitar la posibilidad de que en el servidor existan dos capturas con el mismo nombre.

La sentencia del ejemplo anterior correspondería a la inserción del resultado de un paso que hubiera finalizado correctamente, puesto que no se envían los campos *PathImatge* y *PathCapturaError* que también tiene la tabla *Resultats_Mobile*. Eso es debido a que dichos campos se han creado con la posibilidad de recibir valores *null* para, precisamente, los casos en que el test falla. Recordemos que el campo *PathCapturaError* contendría el nombre de la imagen del móvil en el momento en el que falla, es decir, la captura de error, mientras que el campo *PathImatge* correspondería a la imagen que se estaba buscando y no se ha encontrado.

La sentencia que se muestra a continuación sería un ejemplo de cómo se insertaría el resultado de un paso que hubiera fallado:

INSERT INTO Resultats_Mobile (IdRobot , IdTerminal , IdTest , IdGrupTest , IdPas , IdError , Data , PathImatge , PathCapturaPrevia , PathCapturaError) VALUES (1 , 2 , 7 , 352 , 17 , 1 , '201402200008495' , 'LineaAbierta.png' , '1-CaixaAndroid-20140220_002938.jpg' , '1-CaixaAndroid-20140220_000910.jpg')

Esta sentencia crearía una entrada en la tabla *Resultats_Mobile* como la de la imagen 7.2. Como se puede ver, esta vez el campo que no se inserta es el de *TempsPas*, puesto que al ser un paso que no ha podido finalizar, no tiene sentido indicar ningún tiempo.

Id	IdRobot	IdTerminal	IdTest	IdGrupTest	IdPas	IdError	Data	TempsPas	PathImatge	PathCapturaPrevia	PathCapturaError
394266	1	2	7	352	17	1	20140220000849	NULL	LineaAbierta.png	1-CaixaAndroid-20140220_002938.jpg	1-CaixaAndroid-20140220_000910.jpg

Figura 7.2: Base de Datos: Ejemplo del contenido de la tabla Resultats_Mobile para un paso finalizado erróneamente.

La explicación anterior corresponde a las tablas que utiliza el agente, pero aún queda alguna tabla cuya existencia no se ha justificado, dejando a un lado claro las relativas al proceso de *procedure* que ya se ha comentado con anterioridad. Esas tablas son:

- *Usuaris_ Mobile*: Se utiliza en el *Reporting Web* para comprobar las credenciales que el usuario introduce al hacer *login*.

- *Test_ Usuari_ Mobile*: Se utiliza en el *Reporting Web* para mostrar la información de los tests correspondientes al usuario que ha hecho *login*.

- *Passos_ Mobile*: Se utiliza en el *Reporting Web* para dar la posibilidad al usuario de visualizar los datos de uno o varios pasos en concreto. En ese caso aparece un *pop-up* como el de la imagen 7.3 con los pasos que contiene el test. Para visualizar esa información el código de la web acude a la tabla *Passos_ Mobile* buscando el campo *Descripcio* a partir del campo *Id*.

- *Errors_ Mobile*: Se utiliza en el *Reporting Web* para mostrar la descripción del tipo de error que ha tenido un test. Recordemos que el agente al enviar los resultados de un paso erróneo, envía en el campo *IdError* el tipo de fallo que ha tenido. De esta forma, en el código de la web a partir del *IdError* se accede a la tabla *Errors_ Mobile* y se obtienen los campos *Tipus* y *Descripcio*. Esto es útil en el apartado de la web en el que se muestra el listado de errores y capturas que se ha tenido durante un periodo determinado por el usuario. En la imagen 7.4 se muestra un ejemplo de lo que podría ser el primer error de un test el día 20 de febrero de 2014. Realmente en la web aparecería una tabla más larga, pero en este caso se muestra tan sólo la primera fila. Como es fácil imaginar, el contenido de las columnas "Error" y "Descripción" corresponden a los valores de los campos *Tipus* y *Descripcio* de la tabla *Errors_ Mobile* de la Base de Datos.

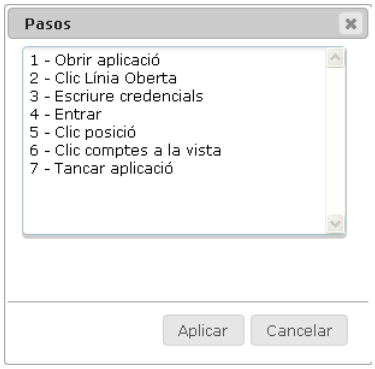


Figura 7.3: *Reporting Web*: Escoger uno o varios pasos sobre los que visualizar los resultados.

Día	Hora	Robot	Test	Script	Error	Descripción	Capturas
20/02/2014	00:08:49	Robot Android	Caixa Android	1 - Obrir aplicació	Error de timeout	No se ha encontrado la imagen	JPG → JPG | JPG

Figura 7.4: *Reporting Web*: Ejemplo de la visualización de un error.

Capítulo 8

Evaluación de la solución

8.1. Control de errores

El código del agente se desarrolló realizando continuas pruebas con la finalidad de minimizar cualquier tipo de problema interno de la aplicación, ya que se debe tener en cuenta que el resultado del desarrollo iba a ser un sistema de monitorización con lo que ello comporta: es inadmisible que el encargado de evaluar la disponibilidad de una aplicación tenga sus propios problemas, ya que sus resultados deben ser consistentes y fiables. Es por ello que en el código del agente se controlan distintos tipos de errores que pueden ocurrir en líneas críticas, con el objetivo de evitar cualquier tipo de *cuelgue* de la aplicación.

Por otro lado, es importante recordar que el agente genera archivos de *logs*, tal y como se dijo en el capítulo 5, en los que quedan registrados todos los pasos que realiza. De esta forma, los posibles errores que pudiera tener la aplicación tanto a nivel de código como a nivel de comunicación con el resto de elementos de la arquitectura, se podrían consultar en el archivo de *logs* de errores. No obstante, este es un método útil a nivel de diagnóstico, pero no de detección y resolución de incidencias en tiempo real. Para ese tipo de soluciones existe lo siguiente:

- Almacenamiento de resultados en local.

  En el código del agente, tal y como se puede ver en el diagrama de flujo del anexo C, se contempla el caso de que los resultados no se puedan enviar a la Base de Datos. Esto puede ocurrir por cualquier problema de comunicación entre el agente y el *Web Service* o éste último y la Base de Datos. Así, en caso de que el ciclo completo no finalice, los resultados se guardan en una Base de Datos local durante el tiempo en el que persiste el problema. En el momento en que se detecta que la inserción de resultados ya finaliza correctamente, se procede a enviar a la Base de Datos remota todos los resultados que se han almacenado localmente, de forma que se evita perder resultados pese a tener algún problema en la arquitectura. Cabe destacar que obviamente esta metodología se realiza con persistencia de datos, es decir, los resultados almacenados en local no se pierden ni aunque la máquina se reinicie.

- Monitorización del servicio.

  En la empresa NexTReT existe un departamento 24x7 cuyo objetivo es monitorizar servicios tanto internos como externos, lo cual realizan mediante un sistema de monitorización llamado *Nagios*. Para que la solución que se presenta en ese proyecto tenga alta disponibilidad, este departamento monitoriza los elementos de su arquitectura con el objetivo de garantizar el servicio en todo momento. Por un lado, se controla que todos los servidores implicados estén operativos: los servidores en los que están las máquinas virtuales de los robots, el servidor en el que se ha desplegado el *Web Service*, el de la Base de Datos, el de la *web*, etc. Por otro lado, también se controla que los robots estén insertando datos a la Base de Datos, de forma que en caso de que algún robot lleve un tiempo sin insertar datos a pesar de estar operativo, por ejemplo porque la máquina o el agente se hayan quedado *colgados*, se detecta al momento.

En conclusión, en caso de que el servicio deje de estar operativo por cualquier motivo, los operadores del departamento 24x7 de NexTReT lo detectarían al momento. Por otro lado, mientras se solucionara la incidencia los resultados se guardarían en local, con lo que no se perderían datos. Finalmente, si la incidencia no fuera ni de sistemas ni de comunicaciones sino debida a un problema del agente, mediante los *logs* de éste se podría ver qué ha pasado para así corregir el código en caso de ser necesario.

8.2. Explotación de resultados

El sistema de monitorización de aplicaciones móviles que se presenta en este PFC se basa en un elemento clave: el agente. Éste es quien interactúa con la aplicación móvil y en consecuencia quien detecta si está disponible o no. La finalidad del resto de elementos de la arquitectura es la de dar visibilidad a los resultados que obtiene el agente y esto básicamente se hace de tres formas: alertas, métricas enviadas a la página web e informes a partir de las métricas anteriores.

- Alertas.

  Si la ejecución de un test no finaliza correctamente, es decir, tiene cualquier tipo de error en uno de sus pasos, automáticamente se envía una alerta al cliente para que sea consciente de dicho error. El módulo de alertas es altamente configurable ya que, por un lado, se puede escoger tanto la franja horaria en la que enviar las alertas como la persona o personas a quien enviárselas, pudiendo discriminar, por ejemplo, por el tipo de error producido. Por ejemplo, al cliente le puede parecer adecuado que un error de servidor se envíe a un departamento en concreto, mientras que un error de código se notifique a otro departamento. Por otro lado, si el cliente tiene un sistema de monitorización propio, las alertas se pueden integrar en su sistema. En caso contrario, éstas se pueden notificar mediante opciones como mensajes a móviles o correos electrónicos.

- Página web.

  Si el cliente no recibe ninguna alerta, puede dar por hecho que su aplicación no está teniendo ningún problema, pero de todas maneras siempre puede acudir al *Reporting Web* para comprobar de forma activa el comportamiento de su aplicación, puesto que todos los resultados de las ejecuciones de tests que genera el agente se envían a la web. A continuación se explica algunas de las vistas y opciones que tiene la web:

  - Panel de estado.

    Es la primera vista al acceder a la web. Consiste en un resumen sobre la disponibilidad de todos los tests que el cliente ha contratado. Como se muestra en la imagen 8.1, con un vistazo rápido se puede ver cómo ha ido la última ejecución de todos los tests, así como las estadísticas de disponibilidad de cada uno de ellos en la última hora, el día actual y el último mes. Además, la información se muestra con un código de colores que facilita su comprensión.

  - Información detallada de cada test.

    Además del resumen general que ofrece el panel de estado, clicando a cualquiera de los tests se puede acceder a información detallada del test. Una vez se clica, se puede ver entre otras cosas un marco como el de la imagen 8.2 sobre el que se puede escoger los resultados que se quieren visualizar: tipo de información ("Tiempo visual", "Disponibilidad" o "Todas las muestras"), información por pasos (en caso de querer ver sólo los resultados de ciertos pasos del test), información de varios tests a la vez y, por último, calendario (se puede escoger uno o más días). A continuación se muestra un ejemplo de los resultados de un test que actualmente se está monitorizando, concretamente los resultados del día 27/02/2014.

    * Tiempo visual.

      En la imagen 8.3 se muestra la información relativa al tiempo visual del test el día 27/02/2014 y, como se puede ver, se ha marcado la ventana temporal en la que el test tuvo problemas. Tal y como se indica en la tabla, entre las 15h y las 16h

hubo más errores de lo normal, lo que hizo que en la gráfica los tiempos subieran (ya que en este caso eran errores de *timeout* puesto que la aplicación tardaba en cargar).
Como se ha dicho anteriormente, la información también se puede visualizar por pasos así que en la imagen 8.4 se muestra el tiempo visual de cada uno de los pasos. Como se puede ver, en la franja en la que hubo problemas todos los pasos tuvieron la misma tendencia, lo que quiere decir que todos tardaban más de lo normal. Además, al visualizar la información por pasos es sencillo ver qué pasos son más críticos puesto que tienen tiempos más altos o más variables.

* Disponibilidad.

  En la imagen 8.5 se puede ver que la disponibilidad del test disminuyó entre las 15h y las 16h debido a los errores anteriormente comentados.

* Todas las muestras.

  En las vistas anteriores la información se detallaba de forma resumida hora a hora, pero también se pueden ver todas y cada una de las ejecuciones que se realizan, para lo que se acudiría a la opción "Todas las muestras". En la imagen 8.6 se puede ver el formato con el que se muestra la información, pero se debe tener en cuenta que la imagen se ha retocado (se han puesto puntos suspensivos) ya que todas las muestras del día no podrían mostrarse en una sola página. Se han dejado las primeras ejecuciones del día así como algunas de las muestras entre las 15h y las 16h, cuando sabemos que hubo errores. Como se puede ver, en la franja citada las muestras que finalizaron correctamente tuvieron tiempos superiores a los de la media debido a, como se ha dicho anteriormente, la lentitud de la aplicación. Por otro lado, las muestras erróneas superaron el *timeout* máximo establecido, con lo que la aplicación tardó aún más en cargar y de ahí que el test no finalizara sus pasos.

* Captura de errores.

  De todas las ejecuciones que se realizan, en la vista "Captura de errores" tan sólo se muestras las que han fallado. De esta forma, se puede ver fácilmente dónde y cómo ha fallado el test. En la imagen 8.7 pueden verse los primeros errores que tuvo el test el día 27/02/2014, el que se está comentado durante todo el escrito. Los *JPGs* que se ven a la derecha de la imagen corresponden, por este orden, a: captura del paso previo al que falla, captura de error (una vez falla el paso) y captura del contenido que se estaba buscando en la aplicación y no se ha encontrado. Al pasar el ratón por encima de los iconos *JPGs* se ve una imagen en miniatura de la captura, lo que permita hacer un vistazo rápido, pero además las capturas se pueden descargar haciendo clic.
  En la imagen 8.8 se puede ver un ejemplo de lo que correspondería a las capturas anteriores. Esta muestra corresponde a una de las ejecuciones que se produjeron entre las 15h y las 16h. Como se puede ver, la captura previa al paso que falla es la que introduce las credenciales en la aplicación. A continuación se pulsó el botón "Entrar" y, como no se encontró la imagen de la derecha (con el contenido "Todos"), se capturó la imagen en la que se ve que la aplicación se quedó cargando. En la imagen 8.9 se muestra otro error que se tuvo, también de *timeout*. En este caso en el paso previo a hacer login, ya que como se puede ver se buscaba el texto "Identificación", pero la aplicación no acabó de cargar esa vista y por eso falló.
  Además de errores de timeout, se detecta cualquier tipo de error que tenga una aplicación ya que la premisa es sencilla: los tests tienen una serie de pasos definidos y si uno de ellos no puede finalizar, la aplicación ha fallado. En las imágenes 8.10, 8.11 y 8.12 se muestran otro tipo de errores que se han encontrado en la aplicación de "La Caixa".

- Informes.

  Si el cliente lo desea se le pueden enviar informes periódicos informando del comportamiento de su aplicación. Por ejemplo, cada día a las 0h puede recibir un resumen del funcionamiento de todos los tests contratados con información relativa a la disponibilidad de cada uno de ellos durante ese día. El cliente puede escoger tanto la frecuencia con la que quiere recibir dichos informes como el contenido de los mismos. De esta forma, puede ser consciente del funcionamiento de su aplicación sin necesidad de acceder al *Reporting Web*.

MASM - Estado de los tests

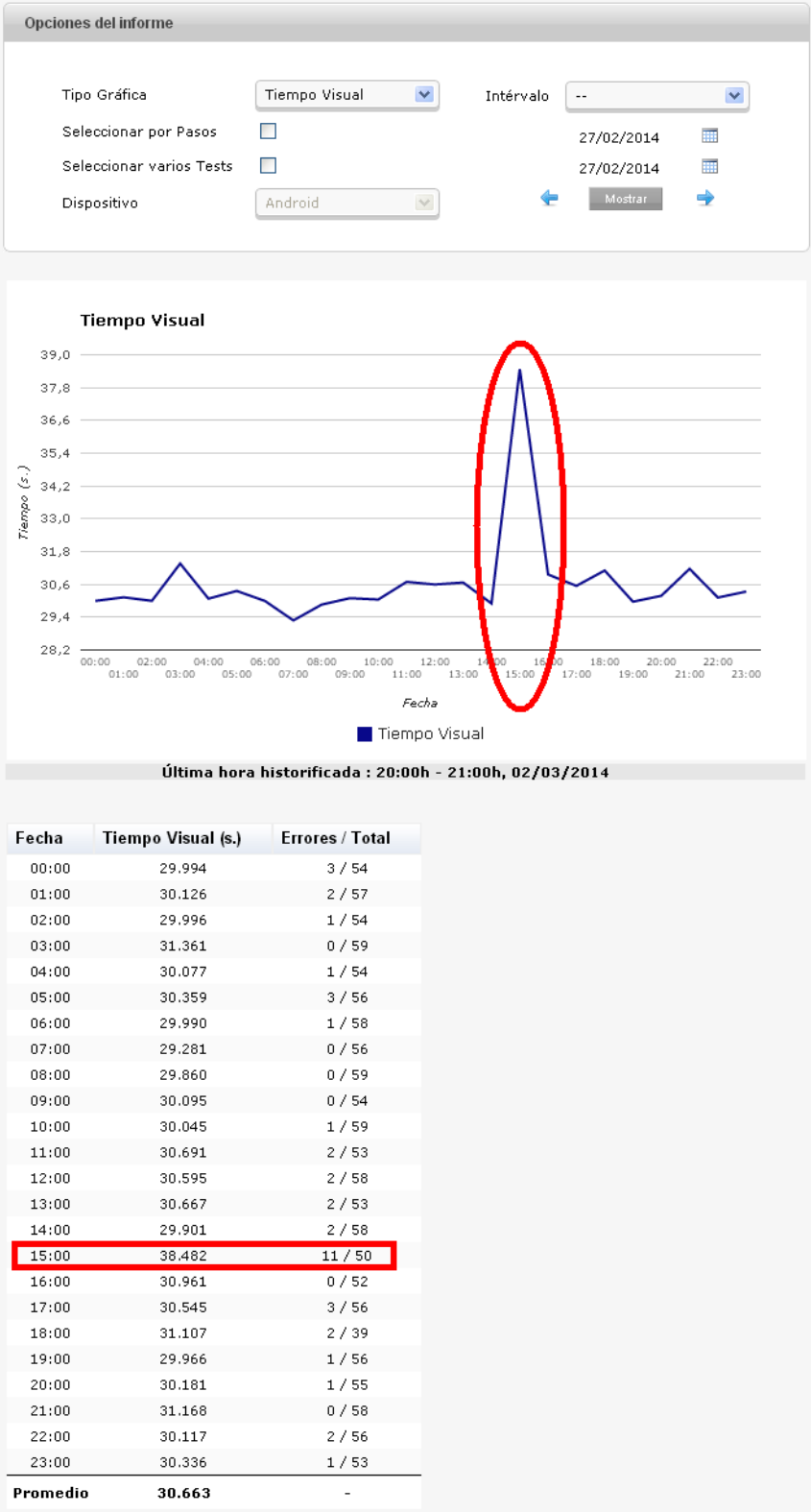
Test	Estado	Última Ejecución	Tiempo Visualización	Disponibilidad Última hora	Disponibilidad Hoy	Disponibilidad Mes
NombreCliente - NombreTest1	✓	02-03-2014 20:44:54	3.20	100%	100%	100%
NombreCliente - NombreTest2	✓	02-03-2014 20:46:26	9.63	100%	79.76%	88.76%
NombreCliente - NombreTest3	✓	02-03-2014 20:53:43	4.00	100%	100%	99.41%
NombreCliente - NombreTest4	✓	02-03-2014 20:40:37	9.62	100%	98.46%	95.21%
NombreCliente - NombreTest5	✓	02-03-2014 20:56:38	4.64	100%	100%	100%
NombreCliente - NombreTest6	✓	02-03-2014 20:43:37	3.68	100%	100%	100%
NombreCliente - NombreTest7	✓	02-03-2014 20:52:12	2.62	100%	97.44%	97.66%
NombreCliente - NombreTest8	✓	02-03-2014 20:55:54	5.25	100%	89.74%	92.26%
NombreCliente - NombreTest9	STOP	-	-	-	-	-
NombreCliente - NombreTest10	✗	02-03-2014 20:44:50	0.00	0%	0%	16.76%
NombreCliente - NombreTest11	✗	02-03-2014 20:46:37	0.00	0%	50%	74.57%
NombreCliente - NombreTest12	✓	02-03-2014 20:43:40	2.71	100%	100%	100%
NombreCliente - NombreTest13	✓	02-03-2014 20:55:17	22.70	50%	75.5%	83.85%
NombreCliente - NombreTest14	✓	02-03-2014 20:55:50	110.70	100%	100%	98.52%

Figura 8.1: *Reporting Web*: Panel de estado.

Opciones del informe

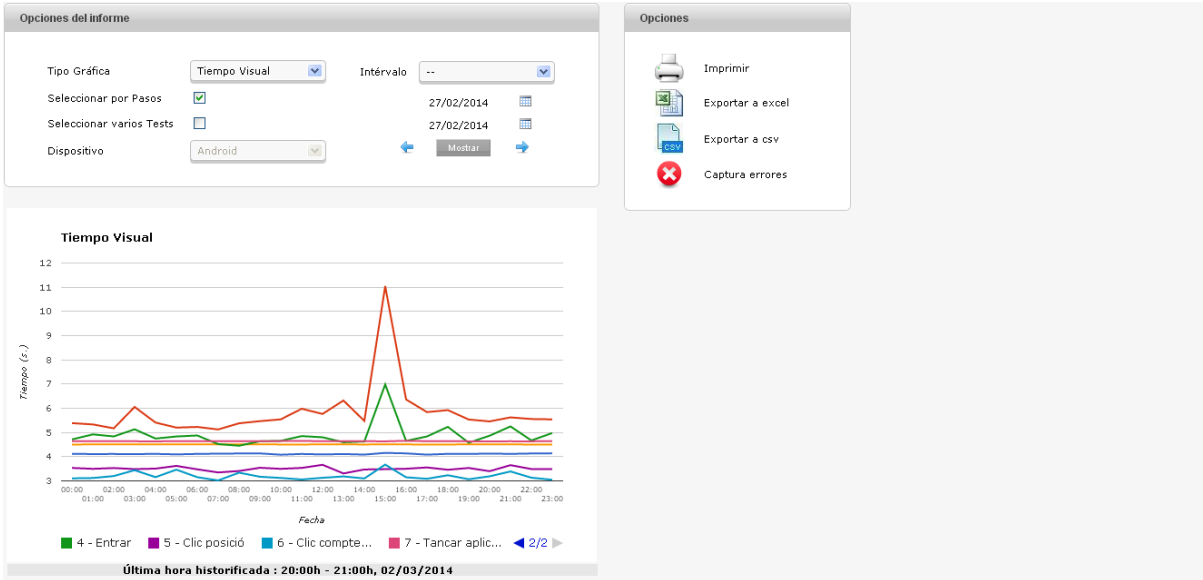
Tipo Gráfica: Disponibilidad
Intérvalo: --
Seleccionar por Pasos
27/02/2014
Seleccionar varios Tests
27/02/2014
Dispositivo: Android
Mostrar

Figura 8.2: *Reporting Web*: Opciones de visualización de los resultados de un test.



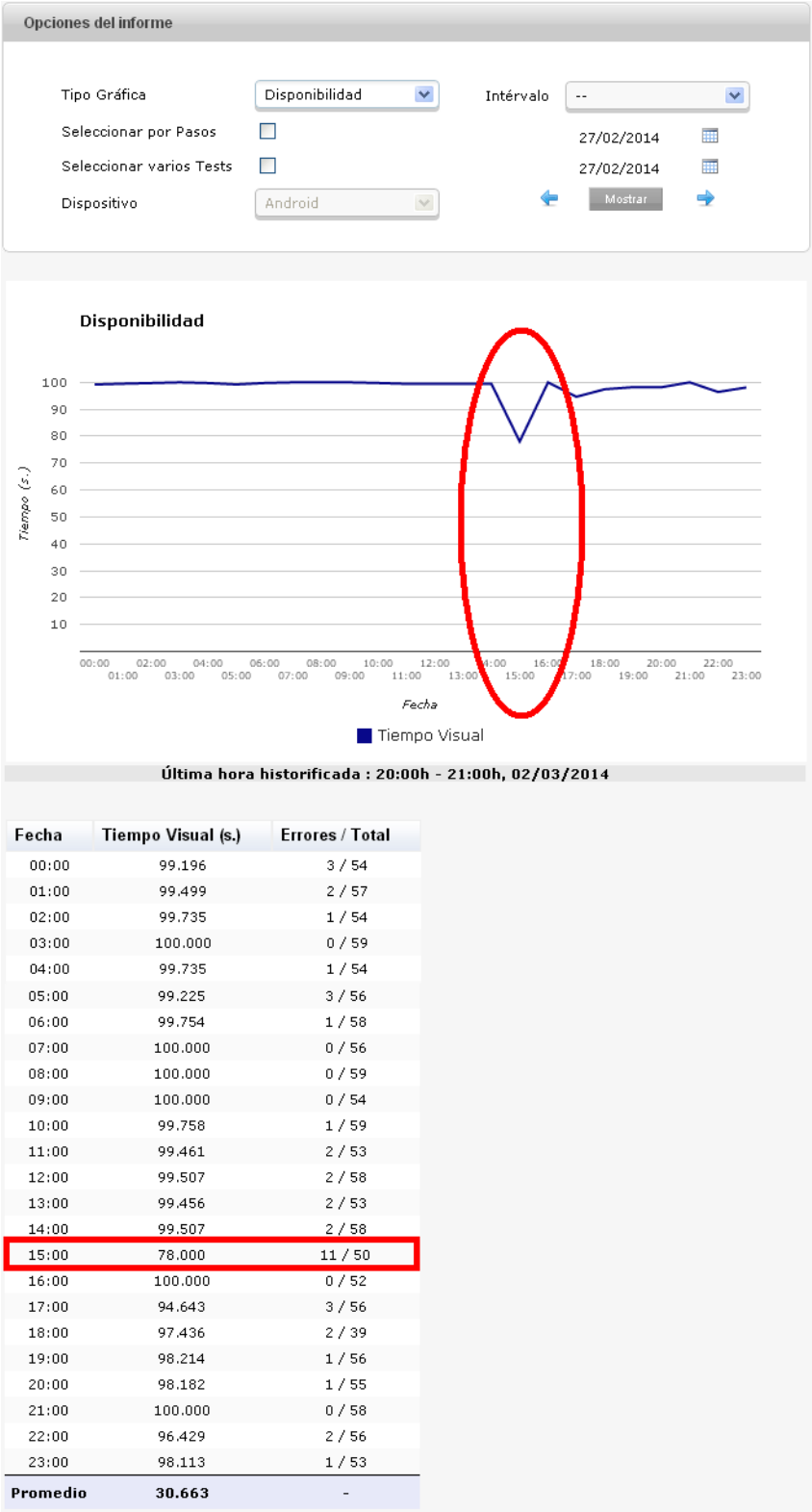
Fecha	Tiempo Visual (s.)	Errores / Total
00:00	29.994	3 / 54
01:00	30.126	2 / 57
02:00	29.996	1 / 54
03:00	31.361	0 / 59
04:00	30.077	1 / 54
05:00	30.359	3 / 56
06:00	29.990	1 / 58
07:00	29.281	0 / 56
08:00	29.860	0 / 59
09:00	30.095	0 / 54
10:00	30.045	1 / 59
11:00	30.691	2 / 53
12:00	30.595	2 / 58
13:00	30.667	2 / 53
14:00	29.901	2 / 58
15:00	38.482	11 / 50
16:00	30.961	0 / 52
17:00	30.545	3 / 56
18:00	31.107	2 / 39
19:00	29.966	1 / 56
20:00	30.181	1 / 55
21:00	31.168	0 / 58
22:00	30.117	2 / 56
23:00	30.336	1 / 53
Promedio	**30.663**	-

Figura 8.3: *Reporting Web*: Tiempo Visual.



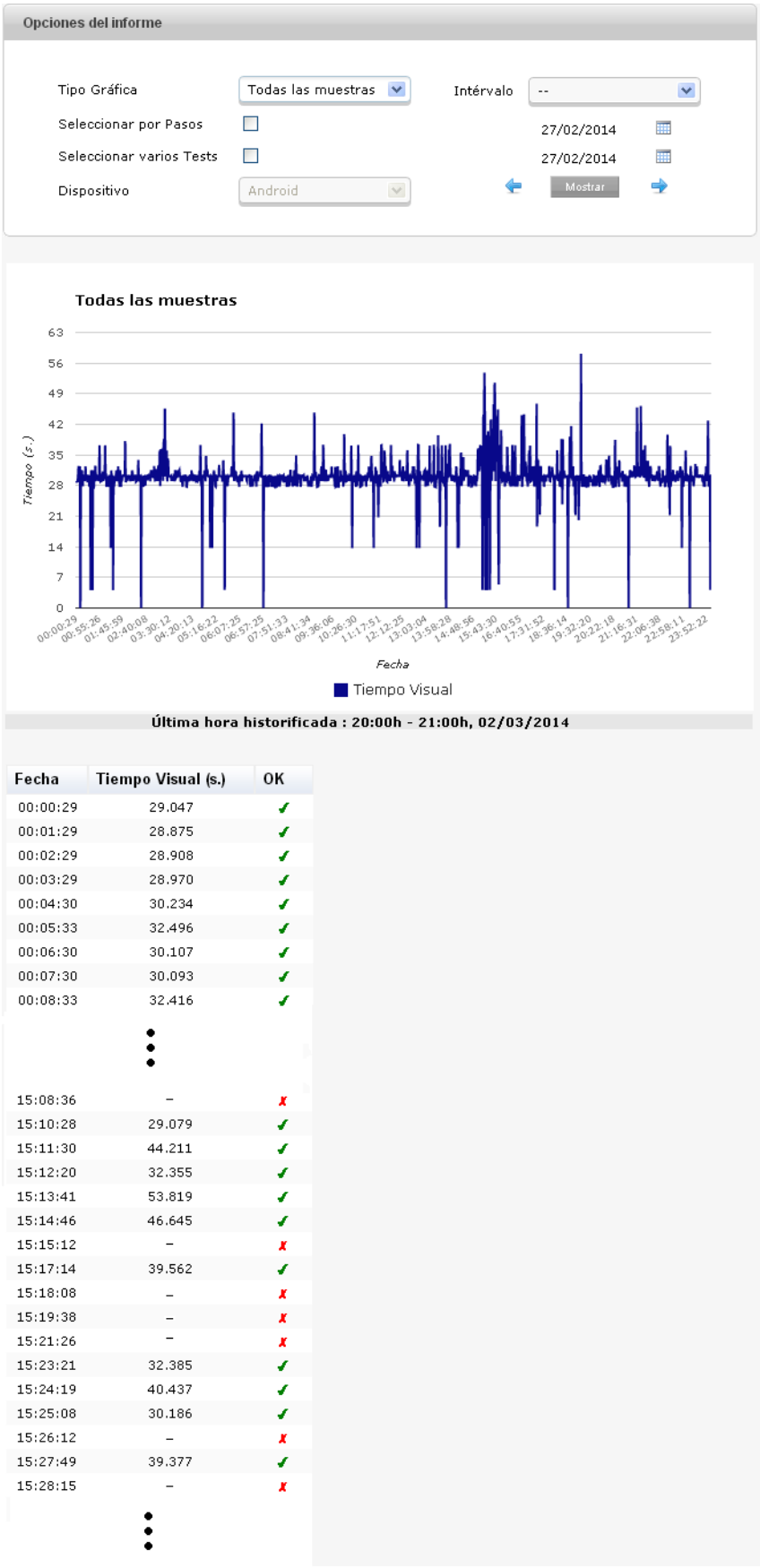
Fecha	1 - Obrir aplicació	Errores / Totales	2 - Clic Línia Oberta	Errores / Totales	3 - Escriure credencials	Errores / Totales	4 - Entrar	Errores / Totales	5 - Clic posició	Errores / Totales	6 - Clic comptes a la vista	Errores / Totales	7 - Tancar aplicació	Errores / Totales
00:00	4.113	1 / 54	5.386	2 / 53	4.502	0 / 51	4.712	0 / 51	3.534	0 / 51	3.104	0 / 51	4.641	0 / 51
01:00	4.110	0 / 57	5.331	1 / 57	4.507	0 / 57	4.920	1 / 57	3.498	0 / 56	3.118	0 / 56	4.641	0 / 56
02:00	4.111	1 / 54	5.171	0 / 53	4.507	0 / 53	4.837	0 / 53	3.531	0 / 53	3.199	0 / 53	4.641	0 / 53
03:00	4.106	0 / 58	6.054	0 / 58	4.506	0 / 58	5.130	0 / 58	3.486	0 / 59	3.442	0 / 59	4.636	0 / 59
04:00	4.116	1 / 54	5.409	0 / 53	4.509	0 / 53	4.749	0 / 54	3.507	0 / 54	3.154	0 / 54	4.633	0 / 54
05:00	4.095	0 / 55	5.201	1 / 56	4.505	0 / 55	4.834	2 / 55	3.619	0 / 53	3.464	0 / 53	4.640	0 / 53
06:00	4.114	1 / 58	5.228	0 / 58	4.506	0 / 58	4.875	0 / 58	3.477	0 / 58	3.152	0 / 58	4.638	0 / 58
07:00	4.124	0 / 56	5.124	0 / 56	4.505	0 / 56	4.520	0 / 56	3.350	0 / 56	3.019	0 / 56	4.639	0 / 56
08:00	4.131	0 / 59	5.378	0 / 59	4.506	0 / 59	4.453	0 / 59	3.408	0 / 59	3.339	0 / 59	4.644	0 / 59
09:00	4.135	0 / 54	5.467	0 / 54	4.507	0 / 54	4.636	0 / 54	3.542	0 / 54	3.171	0 / 54	4.637	0 / 54
10:00	4.081	0 / 58	5.542	0 / 58	4.504	0 / 58	4.654	1 / 59	3.498	0 / 58	3.121	0 / 58	4.645	0 / 58
11:00	4.113	0 / 53	5.979	0 / 53	4.503	0 / 53	4.850	2 / 53	3.535	0 / 51	3.058	0 / 51	4.652	0 / 51
12:00	4.092	0 / 57	5.765	0 / 58	4.508	0 / 58	4.801	2 / 58	3.664	0 / 56	3.127	0 / 56	4.638	0 / 56
13:00	4.109	1 / 53	6.321	0 / 52	4.506	0 / 52	4.596	0 / 52	3.307	1 / 52	3.185	0 / 52	4.643	0 / 52
14:00	4.086	0 / 57	5.476	0 / 57	4.504	0 / 58	4.625	2 / 58	3.471	0 / 56	3.100	0 / 56	4.639	0 / 56
15:00	4.156	0 / 50	11.045	5 / 50	4.507	0 / 45	6.982	4 / 45	3.484	2 / 41	3.673	0 / 40	4.633	0 / 40
16:00	4.136	0 / 52	6.363	0 / 52	4.505	0 / 52	4.651	0 / 52	3.501	0 / 52	3.148	0 / 52	4.656	0 / 52
17:00	4.084	0 / 56	5.841	1 / 56	4.502	0 / 55	4.835	0 / 55	3.555	1 / 55	3.086	1 / 54	4.641	0 / 54
18:00	4.116	1 / 39	5.923	0 / 38	4.503	0 / 38	5.232	0 / 38	3.460	0 / 39	3.233	1 / 39	4.640	0 / 38
19:00	4.112	0 / 56	5.535	0 / 56	4.508	0 / 56	4.575	0 / 56	3.533	0 / 56	3.068	1 / 56	4.636	0 / 55
20:00	4.129	1 / 55	5.457	0 / 54	4.505	0 / 55	4.866	0 / 55	3.400	0 / 55	3.189	0 / 55	4.635	0 / 55
21:00	4.114	0 / 58	5.623	0 / 58	4.506	0 / 58	5.249	0 / 58	3.650	0 / 58	3.390	0 / 58	4.636	0 / 58
22:00	4.132	0 / 56	5.556	0 / 56	4.503	0 / 56	4.671	1 / 56	3.489	0 / 55	3.132	1 / 55	4.634	0 / 54
23:00	4.139	1 / 53	5.541	0 / 52	4.505	0 / 52	4.976	0 / 53	3.483	0 / 53	3.046	0 / 53	4.646	0 / 53
Promedio	**4.115**	-	**5.821**	-	**4.505**	-	**4.885**	-	**3.499**	-	**3.197**	-	**4.640**	-

Figura 8.4: *Reporting Web*: Tiempo Visual por pasos.



Fecha	Tiempo Visual (s.)	Errores / Total
00:00	99.196	3 / 54
01:00	99.499	2 / 57
02:00	99.735	1 / 54
03:00	100.000	0 / 59
04:00	99.735	1 / 54
05:00	99.225	3 / 56
06:00	99.754	1 / 58
07:00	100.000	0 / 56
08:00	100.000	0 / 59
09:00	100.000	0 / 54
10:00	99.758	1 / 59
11:00	99.461	2 / 53
12:00	99.507	2 / 58
13:00	99.456	2 / 53
14:00	99.507	2 / 58
15:00	78.000	11 / 50
16:00	100.000	0 / 52
17:00	94.643	3 / 56
18:00	97.436	2 / 39
19:00	98.214	1 / 56
20:00	98.182	1 / 55
21:00	100.000	0 / 58
22:00	96.429	2 / 56
23:00	98.113	1 / 53
Promedio	**30.663**	-

Figura 8.5: *Reporting Web*: Disponibilidad.



Fecha	Tiempo Visual (s.)	OK
00:00:29	29.047	✓
00:01:29	28.875	✓
00:02:29	28.908	✓
00:03:29	28.970	✓
00:04:30	30.234	✓
00:05:33	32.496	✓
00:06:30	30.107	✓
00:07:30	30.093	✓
00:08:33	32.416	✓
⋮		
15:08:36	–	✗
15:10:28	29.079	✓
15:11:30	44.211	✓
15:12:20	32.355	✓
15:13:41	53.819	✓
15:14:46	46.645	✓
15:15:12	–	✗
15:17:14	39.562	✓
15:18:08	–	✗
15:19:38	–	✗
15:21:26	–	✗
15:23:21	32.385	✓
15:24:19	40.437	✓
15:25:08	30.186	✓
15:26:12	–	✗
15:27:49	39.377	✓
15:28:15	–	✗
⋮		

Figura 8.6: *Reporting Web*: Todas las muestras.

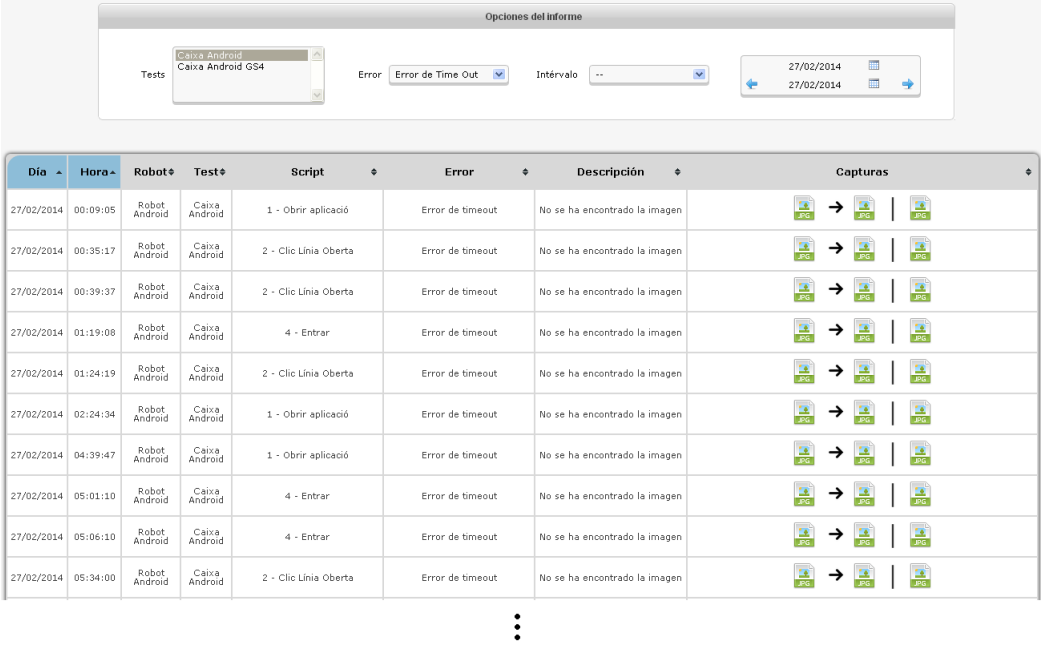


Figura 8.7: *Reporting Web*: Captura de errores.

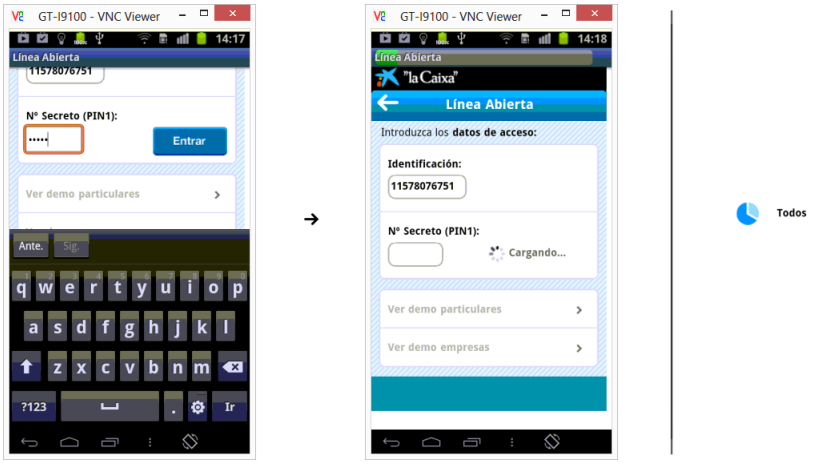


Figura 8.8: *Reporting Web*: Ejemplo capturas de una muestra errónea.

Figura 8.9: *Reporting Web*: Ejemplo capturas de una muestra errónea.

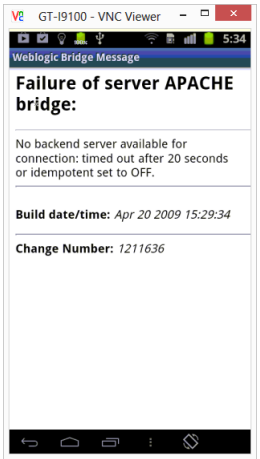


Figura 8.10: Ejemplo error de una aplicación móvil: error del servidor.

Figura 8.11: Ejemplo error de una aplicación móvil: operación no disponible.

Figura 8.12: Ejemplo error de una aplicación móvil: error de código.

Capítulo 9

Conclusiones finales

9.1. Puntos fuertes del proyecto

En este apartado se analizan los aspectos más destacables de este PFC, tanto desde el punto de vista académico como desde el empresarial, resaltando la aportación de la autora en la empresa NexTReT y en el sector.

Los puntos fuertes a nivel académico del sistema desarrollado son los siguientes:

- Se ha diseñado un sistema completo de monitorización pensando todos los elementos necesarios y las particularidades de cada uno.
- El agente, como aplicación Java, es eficiente, escalable y modular. Eso es debido a que previamente a la fase de programación se realizó una fase de diseño en la que se definió el funcionamiento y la arquitectura de la aplicación mediante diagramas de flujo y de clase, obteniendo con ello una aplicación regida por el paradigma de la programación orientada a objetos. Con ese buen diseño inicial no sólo se consiguió que la tarea de programación de código fuera sencilla, sino que se aseguró una reducción en el tiempo de mantenimiento posterior.
- El diseño de la Base de Datos relacional que contiene este proyecto se hizo meticulosamente, determinando tanto las entidades como los atributos necesarios, estableciendo las claves principales y relacionando las tablas de una forma lógica para la solución.

Por otro lado, se debe tener en cuenta que en este PFC se ha desarrollado el *core* de un servicio de monitorización de aplicaciones móviles que la empresa NexTReT está ofreciendo con el nombre de MASM. Así, se considera interesante mencionar los puntos fuertes de la solución en el ámbito empresarial:

- La competencia en el mercado es escasa y la existente ofrece sistemas que requieren de un esfuerzo económico considerable. La sencillez que se ha conseguido en el sistema implementado en este proyecto permite ajustar más los precios consiguiendo así una mejor relación calidad-precio.
- Entidades como *La Caixa* ya están actualmente interesadas en el servicio y su adquisición está en fase de negociación.
- La autora del presente PFC ha presentado MASM en la pasada edición del *Mobile World Congress*, ampliando así la presencia en el mercado y dando una visibilidad al servicio dentro del sector.
- La solución que se ha conseguido es independiente del Sistema Operativo del móvil, con lo que en caso de que un cliente solicitara el servicio en móviles que no fueran ni Android ni iOS, los que actualmente soporta MASM, la adaptación sería muy sencilla (recuérdese que el tipo de Sistema Operativo sólo influye en cómo realizar la conexión VNC con el móvil).
- MASM interactúa con todo tipo de aplicaciones, tanto nativas como basadas en web así como versiones híbridas, con lo que es capaz de monitorizar cualquier aplicación.
- Las alertas que genera MASM se pueden integrar en el sistema de monitorización del cliente, en caso de que tenga uno, facilitando así sus procedimientos.
- El servicio que se ofrece es altamente personalizable ya que el cliente puede escoger todas las opciones siguientes:
    - Sistema Operativo del móvil.
    - Aplicación o aplicaciones a monitorizar.
    - Pasos del test o de los tests.
    - Frecuencia de ejecución de los tests.

- Cuándo, cómo y quién debe recibir las alertas.
- Frecuencia e información de los informes a recibir.

9.2. Trabajos futuros

El sistema que se ha desarrollado en este PFC constituye una primera versión de lo que será un proyecto en constante evolución en la empresa NexTReT. Las mejoras que se preveen implementar son las siguientes:

- Mejoras funcionales:
  - Diferenciación de errores.
    Actualmente MASM detecta si una aplicación móvil falla, pero no el tipo de error que ha tenido. Es decir, la información que se proporciona en el *Reporting Web* es que el test ha fallado, pero sería deseable reportar también qué tipo de error ha tenido la aplicación: error de código, error de resolución de dns, error de conexión con el servidor, etc.
  - Información de las transacciones.
    Con tal de proporcionar más fiabilidad a los resultados enviados a la página web, sería deseable obtener datos de la navegación que se realiza en la aplicación móvil mediante un *sniffer* o una solución parecida. De esa forma el cliente vería en qué punto exacto su aplicación ha fallado, la última transacción realizada y si ha habido algún tipo de dato corrupto.
  - Servidor VNC iOS propio.
    A día de hoy MASM tiene un *hándicap* para monitorizar terminales iOS que es la dependencia del *jailbreak*. Para realizar la conexión VNC entre la máquina y el móvil es necesario que éste último tenga instalado un servidor VNC y la aplicación que cumple esa función forma parte de *Cydia*, el equivalente a la *Apple Store* para los terminales con *jailbreak*. Esto provoca que para cada nueva versión iOS que sale, MASM no es capaz de monitorizar un móvil con esa versión hasta que sale el *jailbreak* correspondiente. En cambio, si se desarrollara una aplicación iOS propia que sirviera de servidor VNC, no sería necesario realizar el *jailbreak* ya que ya no habría que acceder a *Cydia*, simplemente se desplegaría la aplicación propia.
  - Grabador de tests.
    Actualmente los *scripts* que constituyen los pasos que deben realizar los tests se implementan manualmente, con lo que se hacen uno a uno de forma personalizada, con el tiempo que ello conlleva. Sería deseable tener un módulo grabador que fuera capaz de convertir en código las acciones que un usuario realizara sobre un móvil, de forma que cualquier usuario pudiera crear o modificar un test. Disponer de ese módulo permitiría, por ejemplo, que los propios clientes gestionaran sus tests.

- Mejoras de accesibilidad:
  - *Reporting Web* versión móvil.
    Con el objetivo de dar mayor versatilidad a la visualización de resultados, sería conveniente desarrollar el *Reporting web* en versión móvil, es decir, que la propia web se visualizara de una manera conveniente en cualquier tipo de móvil o *tablet* o, incluso, disponer de una aplicación móvil. De esta manera los clientes podrían visualizar los resultados de sus tests en cualquier lugar y situación sin necesidad de un ordenador.
  - Administración de robots propia del cliente.
    MASM actualmente se ofrece como un servicio en el que el cliente no tiene que hacer nada, simplemente recibe la información correspondiente a las ejecuciones de los tests. Así, es NexTReT quien se encarga de la creación y modificación de los tests, así como de su distribución en los robots de los clientes. Sería una buena alternativa ofrecer una nueva modalidad en la que los clientes pudieran gestionar todo eso:

* Con la creación de un grabador de tests que convirtiera en código las acciones realizadas en un móvil, cualquier persona podría crear los tests. De esa forma, se evitaría la situación actual en la que el equipo de MASM de NexTReT es quien dedica tiempo en la creación de cada test. Así, con un grabador el cliente podría crear sus propios tests.
* La administración de los tests también podría realizarla el cliente en caso de que se creara una consola de gestión *online*. Con ella, se podría crear un nuevo test dentro de un robot, mover un test de un robot a otro, eliminar un test, etc. Todo esto actualmente lo hace NexTReT y, con una herramienta de ese tipo que se integrara, por ejemplo, en el propio *Reporting Web*, el cliente podría gestionar los robots contratados.

Glosario de términos y acrónimos

ADB Android Debug Bridge

API Application Programming Interface

BDD Behavior Driven Development

Casos de prueba En la ingeniería del software, los casos de prueba son un conjunto de condiciones o variables bajo las cuáles el analista determina si el requisito de una aplicación es parcial o completamente satisfactorio.

Circuito En este proyecto se utiliza este término para definir el conjunto de pasos automatizados que se realizan en una aplicación móvil. Equivalente al término "test".

DSL Domain Specific Language

Eclipse En programación, IDE de desarrollo. Es el que globalmente más se utiliza y el que se ha usado para desarrollar el presente proyecto.

Framework En programación, capa de abstracción que proporciona una metodología para el desarrollo de una aplicación.

GUI Graphical User Interface

IDE Integrated Development Environment

Jailbreak Proceso de suprimir algunas de las limitaciones impuestas por *Apple* en dispositivos que utilicen el Sistema Operativo iOS mediante el uso de kernels modificados.

JNI Java Native Interface

Librería En programación, conjunto de implementaciones funcionales que ofrece una interfaz bien definida para la funcionalidad que se invoca.

MASM Mobile Application Status Monitor

Open Source En el ámbito de la informática, se califica como *open source* a los programas que permiten el acceso a su código, lo que facilita modificaciones por parte de otros programadores ajenos a los creadores originales del *software* en cuestión.

PFC Proyecto Final de Carrera

Root En el contexto de terminales Android, permite a las aplicaciones acceder a partes del sistema operativo que normalmente están ocultas por el propio fabricante.

Test En este proyecto se utiliza este término para definir el conjunto de pasos automatizados que se realizan en una aplicación móvil. Equivalente al término "circuito".

Testing GUI Pruebas desde la Interfaz Gráfica de Usuario.

UI User Interface

UML Unified Modeling Language

UPC Universitat Politècnica de Catalunya

Índice de figuras

3.1. Realizar acciones sobre una aplicación móvil para comprobar su funcionamiento . . 21
3.2. Diagrama de flujo funcional de MASM. 22
3.3. Módulos funcionales de MASM. 23
3.4. Estructura lógica con la que se organizan los tests de MASM. 24
3.5. Panel de estado del Reporting web de MASM. 25
3.6. Diagrama de la arquitectura MASM. 26

4.1. UI Automation: Interfaz gráfica de *Automation*. 32
4.2. UI Automation: Ejemplo de *script* que genera eventos aleatorios. 33
4.3. UI Automation: Ejemplo de línea de *script* que obtiene los elementos de una aplicación. 34
4.4. MonkeyRunner: Cómo escoger uno de los móviles conectados por USB. 38
4.5. MonkeyRunner: Interfaz de Usuario capturada por *uiautomatorviewer*. 39
4.6. MonkeyRunner: Resultado ejecución *script dump.py*. 39
4.7. MonkeyRunner: Ejemplo de test escrito en *Java*. 40
4.8. Frank: Ejemplo de *script*. 42
4.9. Sikuli: Diagrama de la herramienta. 43
4.10. Sikuli: IDE de Sikuli. 44
4.11. Sikuli: Ejemplo *script* desde el IDE de Sikuli. 45
4.12. Sikuli: Ejemplo *script* desde un editor de texto. 45

5.1. Agente: Ejemplo contenido archivo de logs de errores de la aplicación. 53
5.2. Agente: Ejemplo contenido archivo de logs de errores de la aplicación. 53
5.3. Agente: Estructura del proyecto Java. 55
5.4. Agente: Código función *main* de la clase *Principal.java* 56
5.5. Agente: Código inicial de la función *initializeAgente* de la clase *Principal.java* . . . 61
5.6. Agente: Ventana de inicio (Izquierda: Windows XP, Derecha: Mac OS) 61
5.7. Agente: Interfaz gráfica (Izquierda: Windows XP, Derecha: Mac OS) 62
5.8. Agente: Botones PLAY y STOP (Izquierda: pausa, Derecha: ejecución) 62
5.9. Agente: Interfaz gráfica en la que se indica el test que se está ejecutando (Izquierda: Windows XP, Derecha: Mac OS) 62
5.10. Agente: Interfaz gráfica pestaña "Móviles Conectados" (Izquierda: Windows XP, Derecha: Mac OS) 63
5.11. Agente: Código *listener* botón PLAY de la clase *ParteGraficaAgente.java* 63
5.12. Agente: Código resumido de la función *createThreadToExecuteTests* de la clase *GestionEjecuciones Tests.java* 64
5.13. Agente: Código de la función *getNextTestToExecute* de la clase *GestionEjecuciones-Tests.java* 64
5.14. Agente: Código de la función *checkTimeCalendarioTest* de la clase *CalendarioTest.java* 65
5.15. Agente: Código de la función *setNextExec* de la clase *CalendarioTest.java* 65
5.16. Agente: Código de la función *executeTest* de la clase *GestionEjecuciones Tests.java* 66
5.17. Agente: Código de la función *connectToDevice* de la clase *ConexionConMovil.java* 66
5.18. Agente: Código resumido de la función *forwardPort* de la clase *ConexionConMovil.java* 67
5.19. Agente: Código resumido de la función *launchForwardPort* de la clase *Conexion-ConMovil.java* 68

5.20. Agente: Código resumido de la función *createResultsAndSendThem* de la clase *GestionResultadosTest.java* . . . 69
5.21. Agente: Código resumido de la función *sendTestImagesToFTPServer* de la clase *ServidorFTP.java* . . . 70
5.22. Agente: Código resumido de la función *sendImageToFTPServer* de la clase *ServidorFTP.java* . . . 70

6.1. *Web Service*: Ejemplo respuesta del *Web Service* a la función *GetIdRobotFromNameRobot*. . . . 73
6.2. Agente: Código resumido de la función *sendURLAndGetNodeResponse* de la clase *WebService.java*. . . . 80
6.3. *Web Service*: Lista de *if*'s para identificar el método a ejecutar. . . . 81
6.4. Código de las funciones *getIdRobotFromNameRobot* (arriba clase *WebService.java* del agente y abajo *Web Service*). . . . 82
6.5. Código de las funciones *getIdsTestsFromIdRobot* (arriba clase *WebService.java* del agente y abajo *Web Service*). . . . 83
6.6. Código de las funciones *getIdsCalendarisFromIdTest* (arriba clase *WebService.java* del agente y abajo *Web Service*). . . . 84
6.7. Código resumido de la funciones *getCalendarioTestClassFromIdCalendari* (arriba clase *WebService.java* del agente y abajo *Web Service*). . . . 85
6.8. Código de las funciones *getIdsPasosFromIdTest* (arriba clase *WebService.java* del agente y abajo *Web Service*). . . . 86
6.9. Código de las funciones *getTestBDFieldsFromIdTest* (arriba clase *WebService.java* del agente y abajo *Web Service*). . . . 87
6.10. Código de las funciones *getSOdeviceFromIdTerminal* (arriba clase *WebService.java* del agente y abajo *Web Service*). . . . 88
6.11. Código de la función *sendResultsToBBDD* de la clase *WebService.java* del agente. . 89
6.12. Código de la función *createSetMobileTestResultsURL* de la clase *WebService.java* del agente. . . . 89
6.13. *Web Service*: Código resumido de la función *setMobileTestResults*. . . . 90
6.14. *Web Service*: Código de la función *getSentenciaKoTablaResultats*. . . . 91
6.15. *Web Service*: Código de la función *getSentenciaOkTablaResultats*. . . . 91
6.16. *Web Service*: Código de la función *sendMobileResultsToBD*. . . . 91

7.1. Base de Datos: Ejemplo del contenido de la tabla Resultats_Mobile para un paso finalizado correctamente. . . . 97
7.2. Base de Datos: Ejemplo del contenido de la tabla Resultats_Mobile para un paso finalizado erróneamente. . . . 97
7.3. *Reporting Web*: Escoger uno o varios pasos sobre los que visualizar los resultados. . 98
7.4. *Reporting Web*: Ejemplo de la visualización de un error. . . . 98

8.1. *Reporting Web*: Panel de estado. . . . 104
8.2. *Reporting Web*: Opciones de visualización de los resultados de un test. . . . 104
8.3. *Reporting Web*: Tiempo Visual. . . . 105
8.4. *Reporting Web*: Tiempo Visual por pasos. . . . 106
8.5. *Reporting Web*: Disponibilidad. . . . 107
8.6. *Reporting Web*: Todas las muestras. . . . 108
8.7. *Reporting Web*: Captura de errores. . . . 109
8.8. *Reporting Web*: Ejemplo capturas de una muestra errónea. . . . 109
8.9. *Reporting Web*: Ejemplo capturas de una muestra errónea. . . . 109
8.10. Ejemplo error de una aplicación móvil: error del servidor. . . . 110
8.11. Ejemplo error de una aplicación móvil: operación no disponible. . . . 110
8.12. Ejemplo error de una aplicación móvil: error de código. . . . 110

Bibliografía

[1] UI Automation JavaScript Reference:
https://developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAutomationRef/_index.html

[2] UI/Application Exerciser Monkey:
http://developer.android.com/tools/help/monkey.html

[3] KeyEvent MonkeyRunner class Reference:
http://developer.android.com/reference/android/view/KeyEvent.html

[4] Monkey Talk | tfortesting:
http://tfortesting.wordpress.com/2012/09/07/setup-process-for-eclipse-and-monkey-talk/

[5] Testing with FoneMonkey:
http://www.gorillalogic.com/userfiles/fonemonkey/FoneMonkeyDevCon.pptx.pdf

[6] Cucumber Information:
http://cukes.info/

[7] UISpec Documentation:
https://code.google.com/p/uispec/wiki/Documentation

[8] Sikuli Documentation:
http://doc.sikuli.org/

[9] How Sikuli Works:
http://doc.sikuli.org/devs/system-design.html

Anexos

Anexo A

MonkeyRunner: *script dump.py*

```python
#! /usr/bin/env python
'''
Copyright (C) 2012  Diego Torres Milano
Created on Feb 3, 2012

@author: diego
'''

__version__ = '4.5.2'

import sys
import os
import getopt

try:
    sys.path.append(os.path.join(os.environ['ANDROID_VIEW_CLIENT_HOME'], 'src'))
except:
    pass

from com.dtmilano.android.viewclient import ViewClient

HELP = 'help'
VERBOSE = 'verbose'
IGNORE_SECURE_DEVICE = 'ignore-secure-device'
FORCE_VIEW_SERVER_USE = 'force-view-server-use'
DO_NOT_START_VIEW_SERVER = 'do-not-start-view-server'
DO_NOT_IGNORE_UIAUTOMATOR_KILLED = 'do-not-ignore-uiautomator-killed'
WINDOW = 'window'
UNIQUE_ID = 'uniqueId'
POSITION = 'position'
CONTENT_DESCRIPTION = 'content-description'
CENTER = 'center'
MAP = {'i':ViewClient.TRAVERSE_CITUI, UNIQUE_ID:ViewClient.TRAVERSE_CITUI,
       'x':ViewClient.TRAVERSE_CITPS, POSITION:ViewClient.TRAVERSE_CITPS,
       'd':ViewClient.TRAVERSE_CITCD, CONTENT_DESCRIPTION:ViewClient.TRAVERSE_CITCD,
       'c':ViewClient.TRAVERSE_CITC, CENTER:ViewClient.TRAVERSE_CITC,
       }

USAGE = 'usage: %s [OPTION]... [serialno]'
SHORT_OPTS = 'HVIFSkw:ixdc'
LONG_OPTS =  [HELP, VERBOSE, IGNORE_SECURE_DEVICE, FORCE_VIEW_SERVER_USE,
              DO_NOT_START_VIEW_SERVER,
              DO_NOT_IGNORE_UIAUTOMATOR_KILLED, WINDOW + '=',
              UNIQUE_ID, POSITION, CONTENT_DESCRIPTION, CENTER]
```

```
LONG_OPTS_ARG = {WINDOW: 'WINDOW'}
OPTS_HELP = {
    'H': 'prints this help',
    'V': 'verbose comments',
    'I': 'ignore secure device',
    'F': 'force view server use (even if UiAutomator present)',
    'S': 'don\'t start ViewServer',
    'k': 'don\'t ignore UiAutomator killed',
    'w': 'dump WINDOW content (default: -1, all windows)',
    'i': 'dump View unique IDs',
    'x': 'dump View positions',
    'd': 'dump View content descriptions',
    'c': 'dump View centers'
    }

def shortAndLongOptions():
    '''
    @return: the list of corresponding (short-option, long-option) tuples
    '''

    short_opts = SHORT_OPTS.replace(':', '')
    if len(short_opts) != len(LONG_OPTS):
        raise Exception('There is a mismatch between short and long options')
    t = tuple(short_opts) + tuple(LONG_OPTS)
    l2 = len(t)/2
    sl = []
    for i in range(l2):
        sl.append((t[i], t[i+l2]))
    return sl

def usage(exitVal=1):
    print >> sys.stderr, USAGE % progname
    print >> sys.stderr, "Try '%s --help' for more information." % progname
    sys.exit(exitVal)

def help():
    print >> sys.stderr, USAGE % progname
    print >> sys.stderr
    print >> sys.stderr, "Options:"
    for so, lo in shortAndLongOptions():
        o = '  -%c, --%s' % (so, lo)
        if lo[-1] == '=':
            o += LONG_OPTS_ARG[lo[:-1]]
        try:
            o = '%-34s %-45s' % (o, OPTS_HELP[so])
        except:
            pass
        print >> sys.stderr, o
```

```
    sys.exit(0)

# __main__
progname = os.path.basename(sys.argv[0])
try:
    opts, args = getopt.getopt(sys.argv[1:], SHORT_OPTS, LONG_OPTS)
    sys.argv[1:] = args
except getopt.GetoptError, e:
    print >>sys.stderr, 'ERROR:', str(e)
    usage()

kwargs1 = {VERBOSE: False, 'ignoresecuredevice': False}
kwargs2 = {'forceviewserveruse': False, 'startviewserver': True, 'autodump': False, 'ignoreuiautomatorkilled': True}
options = {WINDOW: -1}
transform = ViewClient.TRAVERSE_CIT
for o, a in opts:
    o = o.strip('-')
    if o in ['H', HELP]:
        help()
    elif o in ['V', VERBOSE]:
        kwargs1[VERBOSE] = True
    elif o in ['I', IGNORE_SECURE_DEVICE]:
        kwargs1['ignoresecuredevice'] = True
    elif o in ['F', FORCE_VIEW_SERVER_USE]:
        kwargs2['forceviewserveruse'] = True
    elif o in ['S', DO_NOT_START_VIEW_SERVER]:
        kwargs2['startviewserver'] = False
    elif o in ['k', DO_NOT_IGNORE_UIAUTOMATOR_KILLED]:
        kwargs2['ignoreuiautomatorkilled'] = False
    elif o in ['w', WINDOW]:
        options[WINDOW] = a
    else:
        transform = MAP[o]

vc = ViewClient(*ViewClient.connectToDeviceOrExit(**kwargs1), **kwargs2)
vc.dump(window=options[WINDOW])
vc.traverse(transform=transform)
```

Anexo B

MonkeyRunner: Función *dump* del *script ViewClient.py*

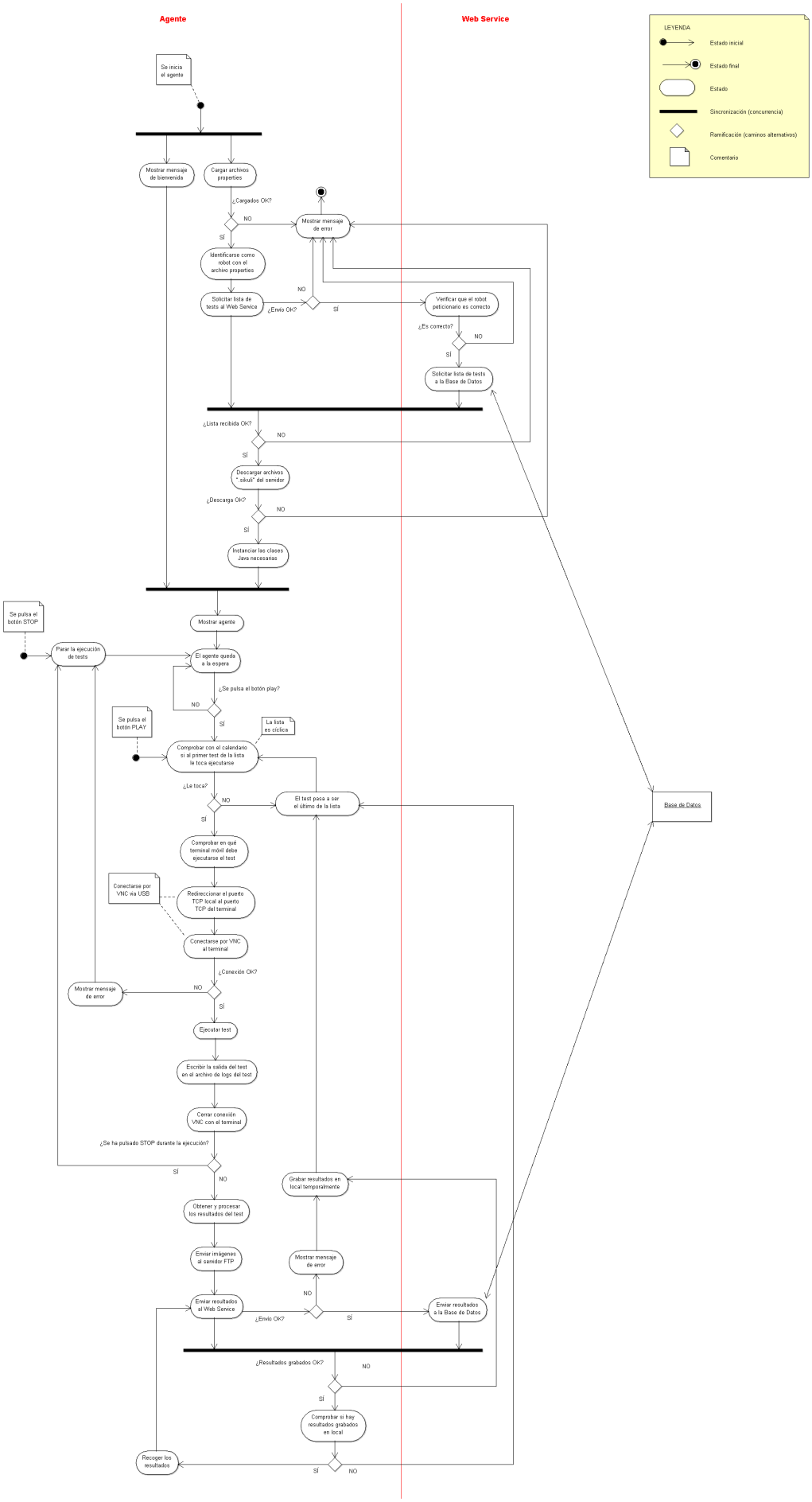
```python
def dump(self, window=-1, sleep=1):
        '''
        Dumps the window content.

        Sleep is useful to wait some time before obtaining the new content when something in the
        window has changed.

        @type window: int or str
        @param window: the window id or name of the window to dump.
                    The B{name} is the package name or the window name (i.e. StatusBar) for
                    system windows.
                    The window id can be provided as C{int} or C{str}. The C{str} should represent
                    and C{int} in either base 10 or 16.
                    Use -1 to dump all windows.
                    This parameter only is used when the backend is B{ViewServer} and it's
                    ignored for B{UiAutomator}.
        @type sleep: int
        @param sleep: sleep in seconds before proceeding to dump the content

        @return: the list of Views as C{str} received from the server after being split into lines
        '''

        if sleep > 0:
            time.sleep(sleep)

        if self.useUiAutomator:
            # NOTICE:
            # Using /dev/tty this works even on devices with no sdcard
            received = unicode(self.device.shell('uiautomator dump /dev/tty >/dev/null'), encoding='utf-8', errors='replace')
            if not received:
                raise RuntimeError('ERROR: Empty UiAutomator dump was received')
            if DEBUG:
                self.received = received
            if DEBUG_RECEIVED:
                print >>sys.stderr, "received %d chars" % len(received)
                print >>sys.stderr
                print >>sys.stderr, repr(received)
                print >>sys.stderr
            onlyKilledRE = re.compile('[\n\S]*Killed[\n\r\S]*',
```

```
                re.MULTILINE)
40              if onlyKilledRE.search(received):
41                  raise RuntimeError('''ERROR: UiAutomator output
                    contains no valid information. UiAutomator was
                    killed, no reason given.''')
42              if self.ignoreUiAutomatorKilled:
43                  if DEBUG_RECEIVED:
44                      print >>sys.stderr, "ignoring UiAutomator
                        Killed"
45                  killedRE = re.compile(
                    '</hierarchy>[\n\S]*Killed', re.MULTILINE)
46                  if killedRE.search(received):
47                      received = re.sub(killedRE, '</hierarchy>',
                        received)
48                  elif DEBUG_RECEIVED:
49                      print "UiAutomator Killed: NOT FOUND!"
50                  # It seems that API18 uiautomator spits this
                    message to stdout
51                  dumpedToDevTtyRE = re.compile(
                    '</hierarchy>[\n\S]*UI hierchary dumped to:
                    /dev/tty.*', re.MULTILINE)
52                  if dumpedToDevTtyRE.search(received):
53                      received = re.sub(dumpedToDevTtyRE,
                        '</hierarchy>', received)
54                  if DEBUG_RECEIVED:
55                      print >>sys.stderr, "received=", received
56              if re.search('\[: not found', received):
57                  raise RuntimeError('''ERROR: Some emulator
                    images (i.e. android 4.1.2 API 16 generic_x86)
                    does not include the '[' command.
58  While UiAutomator back-end might be supported 'uiautomator'
    command fails.
59  You should force ViewServer back-end.''')
60              self.setViewsFromUiAutomatorDump(received)
61          else:
62              if isinstance(window, str):
63                  self.list(sleep=0)
64                  found = False
65                  for wId in self.windows:
66                      try:
67                          if window == self.windows[wId]:
68                              window = wId
69                              found = True
70                              break
71                      except:
72                          pass
73                      try:
74                          if int(window) == wId:
75                              window = wId
76                              found = True
77                              break
```

```
78                      except:
79                          pass
80                      try:
81                          if int(window, 16) == wId:
82                              window = wId
83                              found = True
84                              break
85                      except:
86                          pass
87
88                  if not found:
89                      raise RuntimeError("ERROR: Cannot find
                        window '%s' in %s" % (window, self.windows))
90
91              s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
92              try:
93                  s.connect((VIEW_SERVER_HOST, self.localPort))
94              except socket.error, ex:
95                  raise RuntimeError("ERROR: Connecting to %s:%d:
                    %s" % (VIEW_SERVER_HOST, self.localPort, ex))
96              cmd = 'dump %x\r\n' % window
97              if DEBUG:
98                  print >>sys.stderr, "executing: '%s'" % cmd
99              s.send(cmd)
100             received = ""
101             doneRE = re.compile("DONE")
102             ViewClient.setAlarm(120)
103             while True:
104                 if DEBUG_RECEIVED:
105                     print >>sys.stderr, "    reading from
                        socket..."
106                 received += s.recv(1024)
107                 if doneRE.search(received[-7:]):
108                     break
109             s.close()
110             ViewClient.setAlarm(0)
111             if DEBUG:
112                 self.received = received
113             if DEBUG_RECEIVED:
114                 print >>sys.stderr, "received %d chars" % len(
                    received)
115                 print >>sys.stderr
116                 print >>sys.stderr, received
117                 print >>sys.stderr
118             self.setViews(received)
119
120             if DEBUG_TREE:
121                 self.traverse(self.root)
122
123         return self.views
```

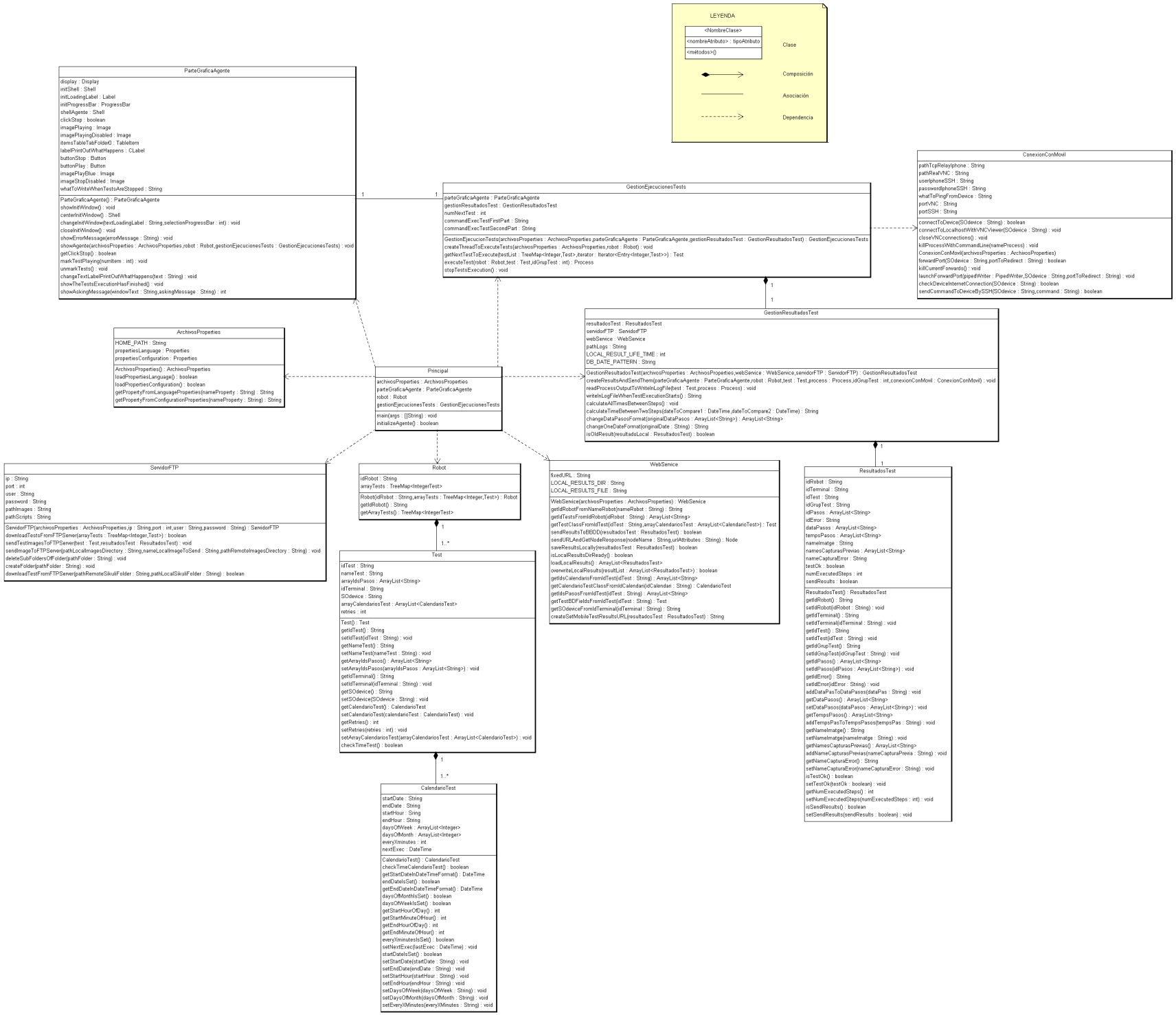
Anexo C

Agente: Diagrama de flujo


Agente
Web Service
LEYENDA
Estado inicial
Estado final
Estado
Sincronización (concurrencia)
Ramificación (caminos alternativos)
Comentario
Se inicia el agente
Mostrar mensaje de bienvenida
Cargar archivos properties
¿Cargados OK?
NO
SÍ
Mostrar mensaje de error
Identificarse como robot con el archivo properties
Solicitar lista de tests al Web Service
¿Envío OK?
NO
SÍ
Verificar que el robot peticionario es correcto
¿Es correcto?
NO
SÍ
Solicitar lista de tests a la Base de Datos
¿Lista recibida OK?
NO
SÍ
Descargar archivos "*.sikuli" del servidor
¿Descarga OK?
NO
SÍ
Instanciar las clases Java necesarias
Mostrar agente
Se pulsa el botón STOP
Parar la ejecución de tests
El agente queda a la espera
¿Se pulsa el botón play?
NO
SÍ
Se pulsa el botón PLAY
La lista es cíclica
Comprobar con el calendario si al primer test de la lista le toca ejecutarse
¿Le toca?
NO
El test pasa a ser el último de la lista
SÍ
Comprobar en qué terminal móvil debe ejecutarse el test
Conectarse por VNC via USB
Redireccionar el puerto TCP local al puerto TCP del terminal
Conectarse por VNC al terminal
¿Conexión OK?
NO
Mostrar mensaje de error
SÍ
Ejecutar test
Escribir la salida del test en el archivo de logs del test
Cerrar conexión VNC con el terminal
¿Se ha pulsado STOP durante la ejecución?
SÍ
NO
Obtener y procesar los resultados del test
Enviar imágenes al servidor FTP
Enviar resultados al Web Service
¿Envío OK?
NO
Mostrar mensaje de error
SÍ
Grabar resultados en local temporalmente
Enviar resultados a la Base de Datos
¿Resultados grabados OK?
NO
SÍ
Comprobar si hay resultados grabados en local
Recoger los resultados
SÍ
NO
Base de Datos

Anexo D

Agente: Diagrama de clases


LEYENDA
<NombreClase>
<nombreAtributo> : tipoAtributo
<métodos>()
Clase
Composición
Asociación
Dependencia
ParteGraficaAgente
display : Display
initShell : Shell
initLoadingLabel : Label
initProgressBar : ProgressBar
shellAgente : Shell
clickStop : boolean
imagePlaying : Image
imagePlayingDisabled : Image
itemsTableTabFolderO : TableItem
labelPrintOutWhatHappens : CLabel
buttonStop : Button
buttonPlay : Button
imagePlayBlue : Image
imageStopDisabled : Image
whatToWriteWhenTestsAreStopped : String
ParteGraficaAgente() : ParteGraficaAgente
showInitWindow() : void
centerInitWindow() : Shell
changeInitWindowTextLoadingLabel : String,selectionProgressBar : int) : void
closeInitWindow() : void
showErrorMessage(errorMessage : String) : void
showAgente(archivosProperties : ArchivosProperties,robot : Robot,gestionEjecucionesTests : GestionEjecucionesTests) : void
getClickStop() : boolean
markTestPlaying(numItem : int) : void
unmarkTests() : void
changeTextLabelPrintOutWhatHappens(text : String) : void
showTheTestsExecutionHasFinished() : void
showAskingMessage(windowText : String,askingMessage : String) : int
1
1
GestionEjecucionesTests
parteGraficaAgente : ParteGraficaAgente
gestionResultadosTest : GestionResultadosTest
numNextTest : int
commandExecTestFirstPart : String
commandExecTestSecondPart : String
GestionEjecucionesTests(archivosProperties : ArchivosProperties,parteGraficaAgente : ParteGraficaAgente,gestionResultadosTest : GestionResultadosTest) : GestionEjecucionesTests
createThreadToExecuteTests(archivosProperties : ArchivosProperties,robot : Robot) : void
getNextTestToExecute(testList : TreeMap<Integer,Test>,iterator : Iterator<Entry<Integer,Test>>) : Test
executeTest(robot : Robot,test : Test,idGrupTest : int) : Process
stopTestsExecution() : void
ConexionConMovil
pathTcpRelayIphone : String
pathRealVNC : String
userIphoneSSH : String
passwordIphoneSSH : String
whatToPingFromDevice : String
portVNC : String
portSSH : String
connectToDevice(SOdevice : String) : boolean
connectToLocalhostWithVNCViewer(SOdevice : String) : void
closeVNCconnections() : void
killProcessWithCommandLine(nameProcess) : void
ConexionConMovil(archivosProperties : ArchivosProperties)
forwardPort(SOdevice : String,portToRedirect : String) : boolean
killCurrentForwards() : void
launchForwardPort(pipedWriter : PipedWriter,SOdevice : String,portToRedirect : String) : void
checkDeviceInternetConnection(SOdevice : String) : boolean
sendCommandToDeviceBySSH(SOdevice : String,command : String) : boolean
1
1
GestionResultadosTest
resultadosTest : ResultadosTest
servidorFTP : ServidorFTP
webService : WebService
pathLogs : String
LOCAL_RESULT_LIFE_TIME : int
DB_DATE_PATTERN : String
GestionResultadosTest(archivosProperties : ArchivosProperties,webService : WebService,servidorFTP : ServidorFTP) : GestionResultadosTest
createResultsAndSendThem(parteGraficaAgente : ParteGraficaAgente,robot : Robot,test : Test,process : Process,idGrupTest : int,conexionConMovil : ConexionConMovil) : void
readProcessOutputToWriteInLogFile(test : Test,process : Process) : void
writeInLogFileWhenTestExecutionStarts() : String
calculateAllTimesBetweenSteps() : void
calculateTimeBetweenTwoSteps(dateToCompare1 : DateTime,dateToCompare2 : DateTime) : String
changeDataPasosFormat(originalDataPasos : ArrayList<String>) : ArrayList<String>
changeOneDateFormat(originalDate : String) : String
isOldResult(resultadoLocal : ResultadosTest) : boolean
ArchivosProperties
HOME_PATH : String
propertiesLanguage : Properties
propertiesConfiguration : Properties
ArchivosProperties() : ArchivosProperties
loadPropertiesLanguage() : boolean
loadPropertiesConfiguration() : boolean
getPropertyFromLanguageProperties(nameProperty : String) : String
getPropertyFromConfigurationProperties(nameProperty : String) : String
Principal
archivosProperties : ArchivosProperties
parteGraficaAgente : ParteGraficaAgente
robot : Robot
gestionEjecucionesTests : GestionEjecucionesTests
main(args : [](String)) : void
initializeAgente() : boolean
ServidorFTP
ip : String
port : int
user : String
password : String
pathImages : String
pathScripts : String
ServidorFTP(archivosProperties : ArchivosProperties,ip : String,port : int,user : String,password : String) : ServidorFTP
downloadTestsFromFTPServer(arrayTests : TreeMap<Integer,Test>) : boolean
sendTestImagesToFTPServer(test : Test,resultadosTest : ResultadosTest) : void
sendImageToFTPServer(pathLocalImagesDirectory : String,nameLocalImageToSend : String,pathRemoteImagesDirectory : String) : void
deleteSubFoldersOfFolder(pathFolder : String) : void
createFolder(pathFolder : String) : void
downloadTestFromFTPServer(pathRemoteSikuliFolder : String,pathLocalSikuliFolder : String) : boolean
Robot
idRobot : String
arrayTests : TreeMap<IntegerTest>
Robot(idRobot : String,arrayTests : TreeMap<Integer,Test>) : Robot
getIdRobot() : String
getArrayTests() : TreeMap<IntegerTest>
1
1..*
Test
idTest : String
nameTest : String
arrayIdsPasos : ArrayList<String>
idTerminal : String
SOdevice : String
arrayCalendariosTest : ArrayList<CalendarioTest>
retries : int
Test() : Test
getIdTest() : String
setIdTest(idTest : String) : void
getNameTest() : String
setNameTest(nameTest : String) : void
getArrayIdsPasos() : ArrayList<String>
setArrayIdsPasos(arrayIdsPasos : ArrayList<String>) : void
getIdTerminal() : String
setIdTerminal(idTerminal : String) : void
getSOdevice() : String
setSOdevice(SOdevice : String) : void
getCalendarioTest() : CalendarioTest
setCalendarioTest(calendarioTest : CalendarioTest) : void
getRetries() : int
setRetries(retries : int) : void
setArrayCalendariosTest(arrayCalendariosTest : ArrayList<CalendarioTest>) : void
checkTimeTest() : boolean
1
1..*
CalendarioTest
startDate : String
endDate : String
startHour : Sring
endHour : String
daysOfWeek : ArrayList<Integer>
daysOfMonth : ArrayList<Integer>
everyXminutes : int
nextExec : DateTime
CalendarioTest() : CalendarioTest
checkTimeCalendarioTest() : boolean
getStartDateInDateTimeFormat() : DateTime
endDateIsSet() : boolean
getEndDateInDateTimeFormat() : DateTime
daysOfMonthIsSet() : boolean
daysOfWeekIsSet() : boolean
getStartHourOfDay() : int
getStartMinuteOfHour() : int
getEndHourOfDay() : int
getEndMinuteOfHour() : int
everyXminutesIsSet() : boolean
setNextExec(lastExec : DateTime) : void
startDateIsSet() : boolean
setStartDate(startDate : String) : void
setEndDate(endDate : String) : void
setStartHour(startHour : String) : void
setEndHour(endHour : String) : void
setDaysOfWeek(daysOfWeek : String) : void
setDaysOfMonth(daysOfMonth : String) : void
setEveryXMinutes(everyXMinutes : String) : void
WebService
fixedURL : String
LOCAL_RESULTS_DIR : String
LOCAL_RESULTS_FILE : String
WebService(archivosProperties : ArchivosProperties) : WebService
getIdRobotFromNameRobot(nameRobot : String) : String
getIdTestsFromIdRobot(idRobot : String) : ArrayList<String>
getTestClassFromIdTest(idTest : String,arrayCalendariosTest : ArrayList<CalendarioTest>) : Test
sendResultsToBEDD(resultadosTest : ResultadosTest) : boolean
sendURLAndGetNodeResponse(nodeName : String,urlAttributes : String) : Node
saveResultsLocally(resultadosTest : ResultadosTest) : boolean
isLocalResultsDirReady() : boolean
loadLocalResults() : ArrayList<ResultadosTest>
overwriteLocalResults(resultList : ArrayList<ResultadosTest>) : boolean
getIdsCalendariosFromIdTest(idTest : String) : ArrayList<String>
getCalendarioTestClassFromIdCalendari(idCalendari : String) : CalendarioTest
getIdsPasosFromIdTest(idTest : String) : ArrayList<String>
getTestBDFieldsFromIdTest(idTest : String) : Test
getSOdeviceFromIdTerminal(idTerminal : String) : String
createSetMobileTestResultsURL(resultadosTest : ResultadosTest) : String
1
1
ResultadosTest
idRobot : String
idTerminal : String
idTest : String
idGrupTest : String
idPasos : ArrayList<String>
idError : String
dataPasos : ArrayList<String>
tempsPasos : ArrayList<String>
nameImatge : String
namesCapturasPrevias : ArrayList<String>
nameCapturaError : String
testOk : boolean
numExecutedSteps : int
sendResults : boolean
ResultadosTest() : ResultadosTest
getIdRobot() : String
setIdRobot(idRobot : String) : void
getIdTerminal() : String
setIdTerminal(idTerminal : String) : void
getIdTest() : String
setIdTest(idTest : String) : void
getIdGrupTest() : String
setIdGrupTest(idGrupTest : String) : void
getIdPasos() : ArrayList<String>
setIdPasos(idPasos : ArrayList<String>) : void
getIdError() : String
setIdError(idError : String) : void
addDataPasToDataPasos(dataPas : String) : void
getDataPasos() : ArrayList<String>
setDataPasos(dataPasos : ArrayList<String>) : void
getTempsPasos() : ArrayList<String>
addTempsPasToTempsPasos(tempsPas : String) : void
getNameImatge() : String
setNameImatge(nameImatge : String) : void
getNamesCapturasPrevias() : ArrayList<String>
addNameCapturasPrevias(nameCapturaPrevia : String) : void
getNameCapturaError() : String
setNameCapturaError(nameCapturaError : String) : void
isTestOk() : boolean
setTestOk(testOk : boolean) : void
getNumExecutedSteps() : int
setNumExecutedSteps(numExecutedSteps : int) : void
isSendResults() : boolean
setSendResults(sendResults : boolean) : void

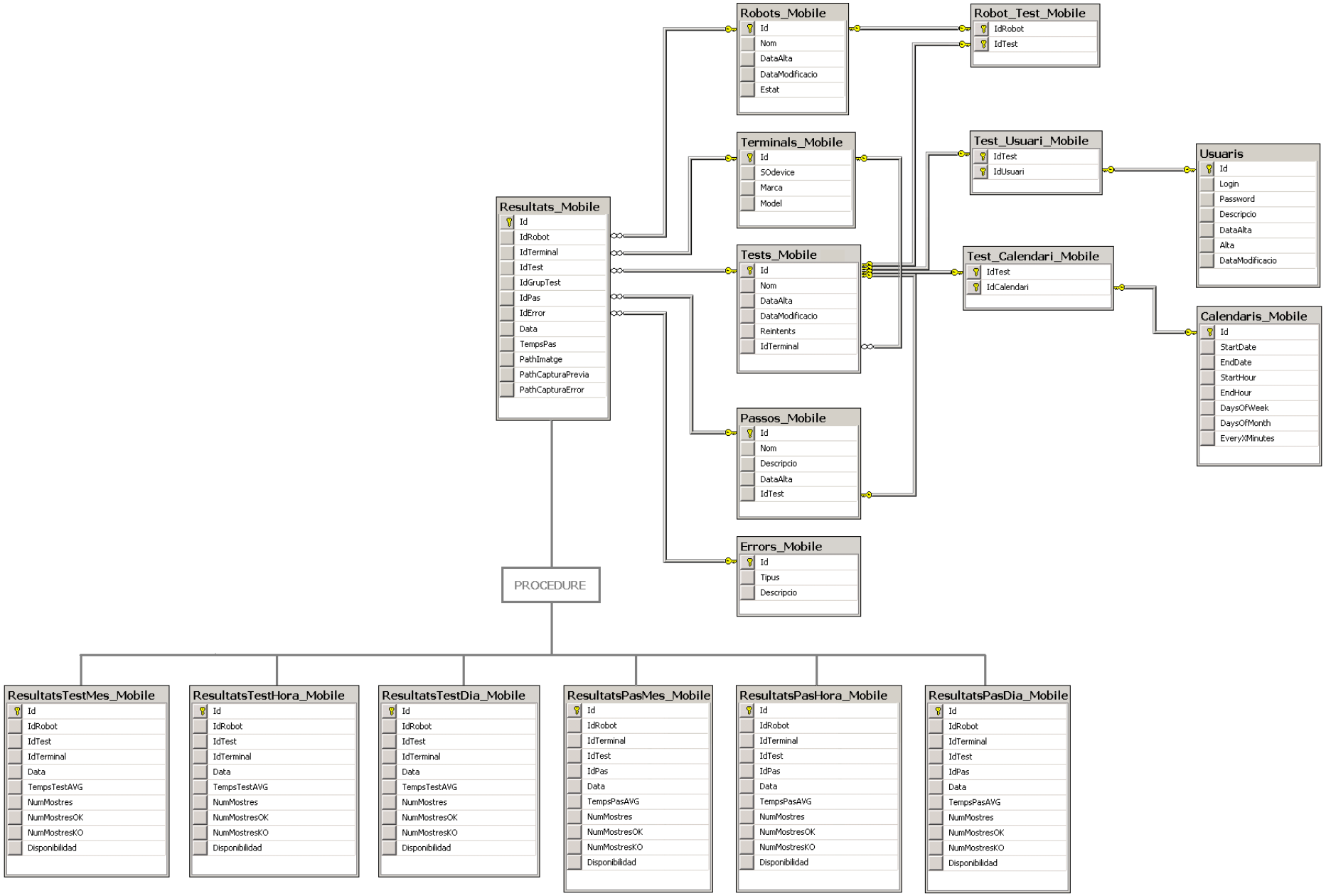
Anexo E

Agente: Contenido archivo *ConfigAgente.properties*

```
1   [Robot]
2
3   nameRobot = robot_piloto
4
5   [Rutas MASM]
6
7   pathLogs = C:/MASM/resultadosTests/logs/
8   pathImages = C:/MASM/resultadosTests/images/
9   pathScripts = C:/MASM/scripts/
10
11  [Sikuli]
12
13  pathSikuliScriptBat = C:/Program Files (x86)/Sikuli X/Sikuli-IDE.bat
14
15  [Conexión VNC]
16
17  pathTcpRelayIphone = C:/MASM/usbmuxd-1.0.8/python-client/tcprelay.py
18  pathRealVNC = C:/MASM/RealVNC
19
20  [Datos dispositivo iOS]
21
22  userIphoneSSH = root
23  passwordIphoneSSH = alpine
24
25  [Web Service]
26
27  urlWebService = http://192.168.7.111:8080/WS_20_Mobile/PeticionsXML?strXML=
28
29  [Servidor FTP Capturas e Imágenes]
30
31  ipFTPServer = 192.168.7.111
32  portFTPServer = 21
33  userFTPServer = Administrator
34  passwordFTPServer = P@ssw0rd
35
36  [Comprobar conectividad móvil]
37
38  whatToPingFromDevice = www.google.es
```

Anexo F

Diagrama de la Base de Datos


Robots_Mobile
Id
Nom
DataAlta
DataModificacio
Estat
Robot_Test_Mobile
IdRobot
IdTest
Terminals_Mobile
Id
SOdevice
Marca
Model
Test_Usuari_Mobile
IdTest
IdUsuari
Usuaris
Id
Login
Password
Descripcio
DataAlta
Alta
DataModificacio
Resultats_Mobile
Id
IdRobot
IdTerminal
IdTest
IdGrupTest
IdPas
IdError
Data
TempsPas
PathImatge
PathCapturaPrevia
PathCapturaError
Tests_Mobile
Id
Nom
DataAlta
DataModificacio
Reintents
IdTerminal
Test_Calendari_Mobile
IdTest
IdCalendari
Calendaris_Mobile
Id
StartDate
EndDate
StartHour
EndHour
DaysOfWeek
DaysOfMonth
EveryXMinutes
Passos_Mobile
Id
Nom
Descripcio
DataAlta
IdTest
Errors_Mobile
Id
Tipus
Descripcio
PROCEDURE
ResultatsTestMes_Mobile
Id
IdRobot
IdTest
IdTerminal
Data
TempsTestAVG
NumMostres
NumMostresOK
NumMostresKO
Disponibilidad
ResultatsTestHora_Mobile
Id
IdRobot
IdTest
IdTerminal
Data
TempsTestAVG
NumMostres
NumMostresOK
NumMostresKO
Disponibilidad
ResultatsTestDia_Mobile
Id
IdRobot
IdTest
IdTerminal
Data
TempsTestAVG
NumMostres
NumMostresOK
NumMostresKO
Disponibilidad
ResultatsPasMes_Mobile
Id
IdRobot
IdTerminal
IdTest
IdPas
Data
TempsPasAVG
NumMostres
NumMostresOK
NumMostresKO
Disponibilidad
ResultatsPasHora_Mobile
Id
IdRobot
IdTerminal
IdTest
IdPas
Data
TempsPasAVG
NumMostres
NumMostresOK
NumMostresKO
Disponibilidad
ResultatsPasDia_Mobile
Id
IdRobot
IdTerminal
IdTest
IdPas
Data
TempsPasAVG
NumMostres
NumMostresOK
NumMostresKO
Disponibilidad

Anexo G

Procedimiento MASM: Plataforma Robot

Rambla Catalunya, 33 – 08077 Barcelona
Tel. (+34) 932 541 530
Calle Fortuny, 3 – 28010 Madrid
Tel. (+34) 917 021 654
nextret@nextret.net www.nextret.net

Business IT Solutions

MASM

Plataforma Robots MASM

Irene García

Layer 8

Proyecto:	E34520
Fecha:	12/12/2013

Índice

1 Introducción **3**

2 Descripción de la plataforma **4**

2.1 Programas a instalar 4

2.1.1 Python 2.7 4

2.1.2 Java 7 4

2.1.3 Sikuli 5

2.1.4 iTunes 5

2.1.5 SDK de Android 6

2.1.6 VMLite Android App Controller 7

2.2 Sistema de ficheros 7

2.2.1 Carpeta “RealVNC” 8

2.2.2 Carpeta “usbmuxd-1.0.8” 8

1 Introducción

Este documento describe la plataforma necesaria para MASM. Esto es los programas y la estructura de ficheros necesaria para que el agente MASM se ejecute correctamente.

2 Descripción de la plataforma

2.1 Programas a instalar

Los siguientes puntos describen los programas necesarios para que el agente MASM se ejecute correctamente. Deben ser instalados como se describe a continuación.

2.1.1 Python 2.7

Necesario para poder realizar la conexión VNC con el terminal móvil por cable USB. La versión, además, debe ser la 2.7.

El instalador puede encontrarse en la ruta:

\\ntnxfs01\home$\Departaments\Innova\MASM\Plataforma_Robot\python-2.7.5.msi

Tan sólo debe hacerse doble clic y seguir los pasos. El programa se instalará en la ruta *"C:\Python27".*

Finalmente, se añadirá la ruta "C:\Python27" como variable de entorno, para ello:

- Panel de control > Sistema > Opciones Avanzadas > Variables de entorno

Ahí veremos que hay variables del usuario y variables del sistema. En las variables del sistema, clicaremos en "Nueva" y creamos la siguiente variable:

- Nombre de variable: **PYTHONPATH**
- Valor de variable: **C:\Python27**

Una vez creada, seleccionaremos la variable "path" y clicaremos en "Modificar". En el campo "valor de variable", iremos al final de la lista y añadiremos lo siguiente:

- **;%PYTHONPATH%**

Si el programa se ha instalado correctamente, al abrir una consola cmd y ejecutar "python", veremos que reconoce el comando.

2.1.2 Java

Necesario tanto para el agente MASM como para ejecutar los tests de Sikuli. Se instalarán dos versiones, la 6 y la 7. La 6 es necesaria para que Sikuli no de problemas, mientras que la 7 es la versión con la que se ha compilado el agente.

Java puede descargarse de la siguiente web:

http://www.java.com/es/download/

La versión que se debe usar por defecto es la 7, es decir, pondremos su ruta como variable de entorno. Instalaremos Java 7 en la ruta "C:\ Archivos de programa\Java\jre7". Una vez instalado, modificaremos las variables de entorno de la siguiente forma:

- Panel de control > Sistema > Opciones Avanzadas > Variables de entorno

Ahí veremos que hay variables del usuario y variables del sistema. En las variables del sistema, clicaremos en "Nueva" y creamos la siguiente variable:

- Nombre de variable: **JAVA_HOME**
- Valor de variable: **C:\ Archivos de programa\Java\jre7**

Una vez creada, seleccionaremos la variable "path" y clicaremos en "Modificar". En el campo "valor de variable", iremos al final de la lista y añadiremos lo siguiente:

- **;%JAVA_HOME%\bin**

Si el programa se ha instalado correctamente, al abrir una consola cmd y ejecutar "java", veremos que reconoce el comando.

2.1.3 Sikuli

Necesario para poder grabar i ejecutar los scripts de los tests. **Debe instalarse la versión de 32 o 64 bits según la máquina**. El instalador del ejemplo es de 32 bits.

El instalador puede encontrarse en la ruta:

\\ntnxfs01\home$\Departaments\Innova\MASM\Plataforma_Robot\Sikuli-X-1.0rc2-win32.exe

Tan sólo debe hacerse doble clic y seguir los pasos. El programa se instalará en la ruta *"C:\Archivos de Programa\Sikuli X".*

2.1.4 iTunes

Necesario para que la máquina reconozca los terminales móviles iOS. Sin este programa no sería posible conectarse por VNC a móviles iOS. **Debe instalarse la versión de 32 bits o 64 según la máquina**. El instalador del ejemplo es de 32 bits.

El instalador puede encontrarse en la ruta:

\\ntnxfs01\home$\Departaments\Innova\MASM\Plataforma_Robot\iTunes\32bits\iTunesSetup.exe

Tan sólo debe hacerse doble clic y seguir los pasos.

Si se ha instalado correctamente, al conectar a la máquina un terminal iOS por cable USB, el iTunes se abrirá automáticamente y reconocerá el dispositivo.

2.1.5 SDK de Android

Necesario para utilizar comunicarse con un móvil Android. **Debe disponerse de la versión de 32 bits o 64 según la máquina.**

El SDK no requiere de ninguna instalación, simplemente es necesario tener la carpeta "adt-bundle" en el ordenador. Las correspondientes carpetas tanto de 32 como de 64 bits pueden encontrarse en la ruta:

\\ntnxfs01\home$\Departaments\Innova\MASM\Plataforma_Robot\sdk_Android

También se puede descargar de la siguiente web:

https://developer.android.com/sdk/index.html

Se copiará la carpeta "adt-bundle" en la siguiente ruta:

C:\Archivos de Programa\MASM

A continuación se actualizarán las variables de entorno, para ello:

- Panel de control > Sistema > Opciones Avanzadas > Variables de entorno

Ahí veremos que hay variables del usuario y variables del sistema. En las variables del sistema, clicaremos en "Nueva" y creamos la siguiente variable:

- Nombre de variable: **SDKHOME**
- Valor de variable: **C:\Archivos de Programa\MASM\adt-bundle\sdk**

Una vez creada, seleccionaremos la variable "path" y clicaremos en "Modificar". En el campo "valor de variable", iremos al final de la lista y añadiremos lo siguiente:

- **;%SDKHOME%\platform-tools**

Si el programa se ha instalado correctamente, al abrir una consola cmd y ejecutar "adb", veremos que reconoce el comando.

2.1.6 VMLite Android App Controller

Necesario para utilizar un móvil Android como servidor VNC. Sin este programa no sería posible conectarse por VNC a móviles Android.

El instalador puede encontrarse en la ruta:

\\ntnxfs01\home$\Departaments\Innova\MASM\Plataforma_Robot\VMLiteAndroidAppControllerSetup.exe

También se puede descargar de la siguiente web:

http://www.vmlite.com/vaac/

Simplemente debe hacerse doble clic sobre el instalador y seguir los pasos. Por último pondremos un acceso directo al programa en el directorio del agente:

C:\Archivos de Programa\MASM

2.2 Sistema de ficheros

El agente MASM requiere un sistema de ficheros concreto en la máquina en la que se ejecuta para saber, por ejemplo, dónde ir a buscar los scripts o dónde dejar los resultados. Estas rutas se pueden modificar siempre que así se indique en el archivo de configuración al que el agente va a buscarlas.

El ejecutable del agente estará en la siguiente ruta:

C:\Archivos de Programa\MASM

En esa misma carpeta debe estar el archivo de configuración *"ConfigAgente.properties"*. Por defecto, el sistema de archivos es el siguiente:

2.2.1 Carpeta "logs"

Debe crearse esta carpeta para que se almacenen los logs de la aplicación MASM.

2.2.2 Carpeta "RealVNC"

En esta carpeta debe estar el ejecutable "vncviewer.exe" y todas aquellas conexiones VNC que previamente se hayan guardado en archivos ".vnc".

2.2.3 Carpeta "usbmuxd-1.0.8"

En esta carpeta hay una serie de scripts que son los encargados de redireccionar el puerto TCP de localhost de la máquina al puerto TCP de un terminal iOS.

Esta carpeta puede encontrarse en la ruta:

\\ntnxfs01\home$\Departaments\Innova\MASM\Plataforma_Robot

No requiere de ningún tipo de instalación, tan sólo debe copiarse y pegarse en el directorio indicado.

CONTROL DE VERSIONES

Versión	Autor	Fecha	Descripción modificaciones
0.0	IGF	12/12/13	Primera versión.
0.1	IGF	13/01/2013	Se actualizan las carpetas del agente.

Anexo H

Procedimiento MASM: Requisitos Terminal Móvil

Rambla Catalunya, 33 – 08077 Barcelona
Tel. (+34) 932 541 530
Calle Fortuny, 3 – 28010 Madrid
Tel. (+34) 917 021 654
nextret@nextret.net www.nextret.net

Business IT Solutions

MASM

Requisitos Terminal Móvil

Irene García

Layer 8

Proyecto:	E34520
Fecha:	12/12/2013

Índice

1 Introducción **3**

2 iOs **4**

2.1 Introducción 4

2.2 Restauración 4

2.2.1 Introducción 4

2.2.2 Modo Recuperación 5

2.2.3 Modo DFU 6

2.3 Jailbreak 8

2.3.1 iOS 6.0 hasta 6.1.1 8

2.4 Veency 9

2.5 OpenSSH 9

2.6 Inetutils 9

2.7 Wget 9

3 Android **10**

3.1 VMLite VNC Server 10

1 Introducción

Este documento describe cómo preparar un terminal móvil para que pueda ser objeto de pruebas con el agente MASM.

2 iOs

2.1 Introducción

Para el caso de terminales con Sistema Operativo iOS, es necesario realizar un *jailbreak* ya que por defecto iOS tiene bastantes restricciones y no permite utilizar el terminal como servidor VNC ni como servidor SSH, dos cosas que necesitamos para MASM.

Antes de realizar el *iailbreak*, en caso de que el terminal se haya usado previamente, se recomienda restaurarlo.

2.2 Restauración

2.2.1 Introducción

En este apartado se explica cómo restaurar un terminal iOS, concretamente se da el ejemplo de un iPhone.

Hay dos modos de hacerlo, los modos recuperación y DFU (Device Firmware Update) del iPhone nos permiten restaurar el firmware del dispositivo mediante iTunes. Visualmente podemos diferenciar uno del otro porque en el primero (modo recuperación) se muestra en la pantalla del iPhone el logo de iTunes mientras que en el segundo (modo DFU) la pantalla se mantiene en negro.

Recovery Mode DFU Mode

La principal diferencia entre ambos modos es iBoot (bootloader (gestor de arranque) de todos los dispositivos con iPhone OS). Mientras que el modo recuperación emplea el iBoot durante el proceso de restauración el modo DFU evita el iBoot. Durante el proceso de restauración, iBoot comprueba que se está instalando una versión del firmware igual o

superior a la que está instalada. Es por ello que si queremos realizar un downgrade deberemos poner el iPhone en modo DFU.

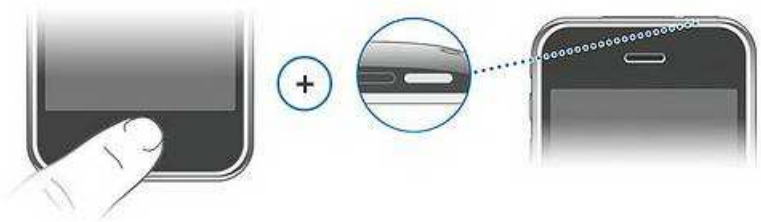
2.2.2 Modo Recuperación

Para poner el iPhone en modo recuperación, deben seguirse los siguientes pasos:

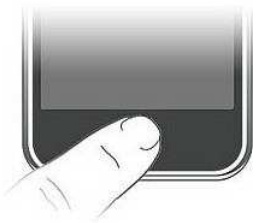
1. Desconectar el cable USB del iPhone, pero dejando el otro extremo conectado al puerto USB del ordenador.

2. Apagar el dispositivo: mantener pulsado el botón de encendido durante unos segundos hasta que aparezca el regulador rojo y, a continuación, arrastrar el regulador. Esperar hasta que el iPhone se apague.

Si no se puede apagar el dispositivo utilizando el regulador, mantener pulsados el botón de encendido y el botón de inicio al mismo tiempo. Cuando el dispositivo se apague, soltar los botones de reposo/activación e inicio.



3. Estando el iPhone ya apagado, mantener pulsado el botón de inicio y conectar el cable USB. Al conectar el cable USB al dispositivo, éste debería encenderse.



4. Mantener pulsado el botón de inicio hasta que se muestre la siguiente imagen en el iPhone.

5. En iTunes aparecerá un mensaje indicando se ha encontrado un dispositivo en modo recuperación. Se acepta el mensaje y se clica en “Restaurar”.

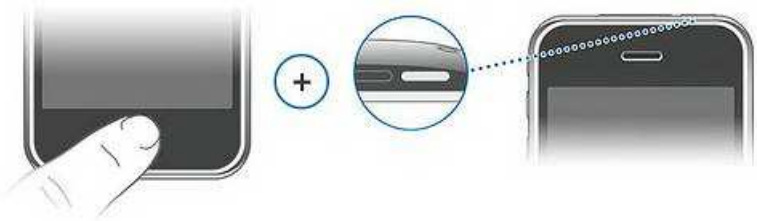
2.2.3 Modo DFU

Para poner el iPhone en modo DFU seguiremos los siguientes pasos:

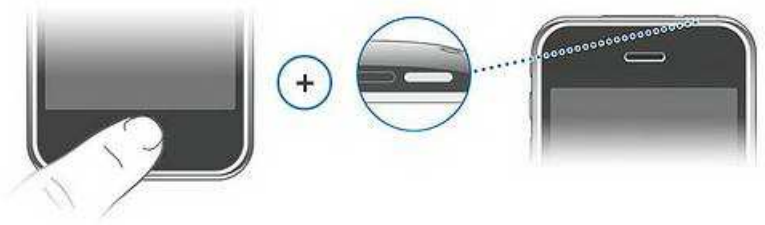
1. Abrir iTunes y conectar el iPhone al ordenador.

2. Apagar el dispositivo: mantener pulsado el botón de encendido durante unos segundos hasta que aparezca el regulador rojo y, a continuación, arrastrar el regulador. Espera hasta que el iPhone se apague.

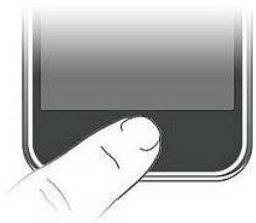
Si no se puede apagar el dispositivo utilizando el regulador, mantener pulsados el botón de encendido y el botón de inicio al mismo tiempo. Cuando el dispositivo se apague, soltar los botones de reposo/activación e inicio.



3. Estando el iPhone ya apagado, pulsar los botones de inicio y encendido simultáneamente y mantenerlos pulsados.



4. Transcurridos exactamente 10 segundos, soltar el botón de encendido y mantener pulsado el de inicio hasta que iTunes diga que ha encontrado un dispositivo en modo recuperación.



La pantalla debería mantenerse totalmente en negro.

5. Se acepta el mensaje de iTunes y se clica en “Restaurar”.

2.3 Jailbreak

2.3.1 iOS 6.0 hasta 6.1.1

A continuación se describe cómo hacer *jailbreak* en **iOS 6.0, 6.0.1, 6.0.2, 6.1 y 6.1.1**. Pasos a seguir:

1. Accederemos a http://www.evasi0n.com y descargaremos el ejecutable de acorde con el sistema operativo de nuestro ordenador. La versión para Windows puede encontrarse en la siguiente ruta:

   I:\Departaments\Innova\MASM\Jailbreak iOS\evasi0n-win-1.5.3\evasi0n.exe

2. Se recomienda realizar una copia de seguridad del terminal por si pasara cualquier cosa. Para ello sincronizaremos el iPhone/iPad/iPod con iTunes.

3. Es recomendable quitar el bloqueo con código antes de empezar a hacer el proceso. Para ello iremos a *"Ajustes > General > Bloqueo con código"* y lo desactivaremos. También es recomendable cerrar por completo iTunes y cualquier programa de Apple.

4. Ejecutar evasi0n, el programa detectará el modelo y versión de nuestro dispositivo:

5. Si lo detecta correctamente, clicamos en "*jailbreak*". El proceso de jailbreak se dividirá en dos partes. Tras realizarse la primera, el dispositivo se reiniciará y al encenderse de nuevo, veremos que aparecerá un icono nuevo en el iPad. Para iniciar el segundo paso, el programa nos pedirá que abramos esa nueva aplicación. Pocos segundos después dará por finalizado el proceso reiniciando el dispositivo y haciendo aparecer la aplicación de *Cydia*. Esta aplicación es la equivalente a la oficial "App Store", sirve para descargarse aplicaciones que oficialmente Apple no permite.

6. Si teníamos activado el bloqueo con código, lo volvemos a activar.

2.4 Veency

Una vez hecho el Jailbreak, veremos que se ha creado una aplicación llamada *Cydia*. Accederemos a *Cydia* y des de ahí instalaremos la aplicación *Veency*. Con esta aplicación podremos utilizar el terminal iOS como servidor VNC.

2.5 OpenSSH

Accederemos a *Cydia* y des de ahí instalaremos la aplicación *OpenSSH*. Con esta aplicación podremos utilizar el terminal iOS como servidor SSH.

2.6 Inetutils

Accederemos a *Cydia* y des de ahí instalaremos la aplicación *Inetutils*. Con esta aplicación dispondremos de comandos como “ping” en la comunicación SSH con el iOS.

2.7 Wget

Accederemos a *Cydia* y des de ahí instalaremos la aplicación *Wget*. Con esta aplicación dispondremos del comando “wget” en la comunicación SSH con el iOS.

3 Android

En el caso de Android no es necesario realizar Jailbreak para usar el terminal como servidor VNC.

3.1 VMLite VNC Server

Para utilizar un Android como servidor VNC, se instalará la aplicación de "Play Store" llamada "VMLite VNC Server".

¡Atención! Es importante asegurarse de que el puerto que utiliza es el **5900** (mirarlo en "Settings" dentro de la aplicación)

3.2 Conectar Android a PC

En el móvil, ir a "opciones de desarrollador" dentro de "ajustes" y activar las opciones "Permanecer activo" y "Depuración de USB". A continuación, conectar un cable usb entre el móvil y el ordenador y comprobar si el PC reconoce correctamente el móvil. Para ello, se ejecutará la siguiente línea en una consola cmd:

- *adb devices*

Si el PC reconoce correctamente el móvil, veremos algo como lo siguiente:

```
List of devices attached
cd1898ff        device
```

A la izquierda el número de serie del móvil y a la derecha "device". Si la lectura es distinta, se procederá a realizar lo siguiente:

a) Si no reconoce ningún móvil:

```
List of devices attached
```

Esto quiere decir que falta algún *driver* que haga que el pc reconozca el móvil como tal.

b) Si pone "offline" en vez de "device":

```
List of devices attached
cd1898ff        offline
```

Se hará lo siguiente:

1. Desenchufar y enchufar el cable que conecta el móvil con el pc.
2. En el móvil, ir a "opciones de desarrollador" dentro de "ajustes" y desactivar y activar la opción "Depuración de USB".

3. Reiniciar el *adb* en el ordenador. Para ello ejecutar lo siguiente en una consola cmd:

   - *adb kill-server*
   - *adb start-server*

4. Tras los comandos anteriores aparecerá un mensaje en el móvil. Se debe seleccionar la opción “Permitir siempre desde este ordenador”.

CONTROL DE VERSIONES

Versión	Autor	Fecha	Descripción modificaciones
0.0	IGF	12/12/13	Primera versión.

Anexo I

Procedimiento MASM: Grabar Tests Sikuli

Rambla Catalunya, 33 – 08077 Barcelona
Tel. (+34) 932 541 530
Calle Fortuny, 3 – 28010 Madrid
Tel. (+34) 917 021 654
nextret@nextret.net www.nextret.net

Business IT Solutions

MASM

Grabar Tests Sikuli

Irene García

Layer 8

Proyecto:	E34520
Fecha:	14/11/2013

Índice

1 Introducción **3**

2 Cómo crear o modificar un test **4**

2.1 Estructura de un test 4

2.2 Requisitos previos 4

2.2.1 Instalación Sikuli 4

2.3 Procedimiento 5

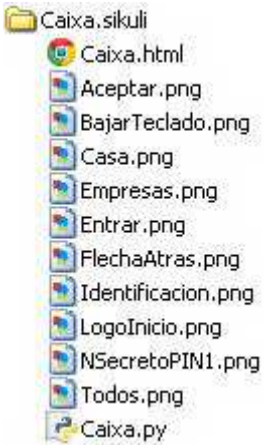
1 Introducción

Este documento describe cómo realizar los tests que el agente MASM reproduce.

2 Cómo crear o modificar un test

2.1 Estructura de un test

El agente MASM reproduce tests con la ayuda de la tecnología de procesado de imágenes que aporta Sikuli. Esta herramienta necesita tener en un mismo directorio (una carpeta acabada en ".sikuli") un script en python que describa los pasos a realizar y todas aquellas imágenes que utilice. Un ejemplo sería el siguiente:



En el ejemplo también puede verse un archivo html. Este archivo lo crea el IDE de Sikuli al compilar el script, pero no aporta nada en cuanto a poder ejecutar el script desde el agente MASM, tan sólo es una forma diferente de ver el test. Por ejemplo, si miráramos el archivo "Caixa.py" veríamos:

click("Entrar.png")

Mientras que en el archivo "Caixa.html" veríamos:

2.2 Requisitos previos

Para crear o modificar un test no es imprescindible tener instalado el programa Sikuli. Podrían capturarse las imágenes manualmente y editar los scripts python en cualquier editor de texto. No obstante, el IDE de Sikuli facilita la tarea, sobretodo para capturar las imágenes, además de crear el archivo html comentado anteriormente.

2.2.1 Instalación Sikuli

Debe instalarse la versión de 32 o 64 bits según la máquina. El instalador del ejemplo es de 32 bits.

El instalador puede encontrarse en la ruta:

I:\Departaments\Innova\MASM\Plataforma Robot\Sikuli-X-1.0rc2-win32.exe

Tan sólo debe hacerse doble clic y seguir los pasos. El programa se instalará en la ruta "C:\Archivos de Programa\Sikuli X".

2.3 Procedimiento

Hasta el momento no se ha creado una plantilla python con la que realizar los tests, con lo que de forma temporal los tests se realizan copiando un test existente y creando un archivo python nuevo con la misma estructura.

CONTROL DE VERSIONES

Versión	Autor	Fecha	Descripción modificaciones
0.0	IGF	14/11/13	Primera versión.

Anexo J

Procedimiento MASM: Añadir calendario a un test

Rambla Catalunya, 33 – 08077 Barcelona
Tel. (+34) 932 541 530
Calle Fortuny, 3 – 28010 Madrid
Tel. (+34) 917 021 654
nextret@nextret.net www.nextret.net

Business IT Solutions

MASM

Añadir calendario a un test

Irene García

Layer 8

Proyecto:	E34520
Fecha:	14/11/2013

Índice

1 Introducción **3**

2 Cómo añadir calendario a un test **4**

1 Introducción

En este documento se explica cómo añadir calendarios a un test. Concretamente se especifica la nomenclatura a utilizar en la Base de Datos.

2 Cómo añadir calendarios a un test

El agente de MASM es capaz de interpretar distintos calendarios para cada test una vez éstos están en la Base de Datos. Actualmente no se ha desarrollado la consola de gestión con la que especificar los calendarios con una interfaz gráfica, con lo que los campos se introducen manualmente a la Base de Datos.

La tabla que define las características de cada calendario es la llamada "Calendaris_Mobile" y tiene los siguientes campos:

- Id: este campo es autoincremental, con lo que no se debe especificar.
- StartDate: día del año en el que empieza a ejecutarse el test *(dd/MM/yyyy)*
- EndDate: día del año en el que deja de ejecutarse el test *(dd/MM/yyyy)*
- StartHour: hora del día en la que empieza a ejecutarse el test *(mm:mm)*
- EndHour: hora del día en la que deja de ejecutarse el test *(mm:mm)*
- DaysOfWeek: días de la semana en los que se ejecuta el test
- DaysOfMonth: días del mes en los que se ejecuta el test
- EveryXminutes: cada cuántos minutos se ejecuta el test

Los posibles valores de "DaysOfWeek" son los números de la semana, es decir, del 1 al 7 que corresponderían con lunes y domingo respectivamente. Los posibles valores de "DaysOfWeek" son los números del mes, es decir, del 1 al 31. En ambos campos pueden especificarse varios días separados por comas. Por ejemplo, en el campo "DaysOfWeek" de lunes a viernes sería "1,2,3,4,5".

Por otro lado, el campo "EveryXminutes" sólo admite un único número.

Todos los campos de la tabla "Calendaris_Mobile" admiten valores nulos (admiten no especificar el campo). Si no se especifica, se entiende que se asigna su valor por defecto. Los valores por defecto de cada campo son:

- StartDate: Empezará a ejecutarse en el momento en que se cree.
- EndDate: Se ejecutará sin fecha fin.
- StartHour: Empezará a ejecutarse a las 00:00h.
- EndHour: Se ejecutará hasta las 23:59h.
- DaysOfWeek: Se ejecutará todos los días de la semana.
- DaysOfMonth: Se ejecutará todos los días del mes.
- EveryXminutes: Se ejecutará continuamente.

A continuación se muestran unos ejemplos de cómo definir calendarios. Las imágenes corresponden a los campos de la tabla "Calendaris_Mobile".

a) Ejecutar un test de lunes a viernes cada 5 minutos:

StartDate	EndDate	StartHour	EndHour	DaysOfWeek	DaysOfMonth	EveryXminutes
NULL	NULL	NULL	NULL	1,2,3,4,5	NULL	5

b) Ejecutar un test sábados y domingos cada 10 minutos:

StartDate	EndDate	StartHour	EndHour	DaysOfWeek	DaysOfMonth	EveryXminutes
NULL	NULL	NULL	NULL	6,7	NULL	10

c) Ejecutar un test de 9h a 18h una vez por minuto:

StartDate	EndDate	StartHour	EndHour	DaysOfWeek	DaysOfMonth	EveryXminutes
NULL	NULL	09:00	18:00	NULL	NULL	1

Para asignar un calendario a un test, se debe añadir una entrada a la tabla "Test_Calendari_Mobile" de la Base de Datos. Esta tabla tiene dos campos:

- IdTest: hace referencia al campo "Id" de la tabla "Tests_Mobile"
- IdCalendari: hace referencia al campo "Id" de la tabla "Calendaris_Mobile"

Por ejemplo, en la siguiente imagen se estaría asignado el calendario con Id=1 al test con Id=2:

IdTest	IdCalendari
2	1

Un test con Id=X puede configurarse de una de estas 3 formas:

a) Un único calendario. La tabla "Test_Calendari_Mobile" sólo tendrá una entrada en la que haya un Id=X. Por ejemplo, en la imagen anterior el test con Id=2 sólo tiene un calendario asignado.

b) Más de un calendario. La tabla "Test_Calendari_Mobile" tendrá más de una entrada en la que haya un Id=X. Por ejemplo, en la siguiente imagen el test con Id=2 tiene dos calendarios asignados (el de Id=1 y el de Id=3):

IdTest	IdCalendari
2	1
2	3

c) Ningún calendario. En la tabla “Test_Calendari_Mobile” el Id=X no aparecerá en ninguna entrada. En este caso el agente ejecutará el test sin ningún tipo de restricción (si es el único test asignado a un agente, se ejecutará continuamente).

CONTROL DE VERSIONES

Versión	Autor	Fecha	Descripción modificaciones
0.0	IGF	29/01/14	Primera versión.