UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Aplicación y revisión de baterías de pruebas automatizadas:

Una herramienta de apoyo para la clasificación de resultados

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor:Adrián Moreno González

Tutor: Patricio Letelier Torres

2015-2016

Agradecimientos

A Patricio Letelier, mi tutor, por darme la oportunidad de realizar este trabajo y las prácticas en la empresa, y a Alejandro Del Rincón, empleado de la empresa, por su paciencia para enseñarme, su apoyo y su ayuda para conseguir que la herramienta fuera eficiente.

Resumen

El presente trabajo aborda la problemática de la clasificación de los resultados de la ejecución de baterías de pruebas automatizadas, proponiendo la utilización de herramientas de apoyo para agilizar esta tarea, la cual se mostró como un potencial cuello de botella en el proceso de Aseguramiento de Calidad (QA). Ha sido realizado en el contexto de una práctica en una empresa, en la cual el autor estaba integrado en el equipo de testeo automatizado, e incluye tanto el estudio de las posibles aproximaciones al desarrollo de herramientas de apoyo, la implementación y el estudio del impacto de la herramienta en el proceso de testeo.

Palabras clave: pruebas automatizadas, pruebas de aceptación, gestión de fallos

Abstract

This paper addresses the problem of classification of the results of automated tests' runs , proposing the use of support tools to agilize this task, which was shown as a potential bottleneck in the process of Quality Assurance (QA). It was made in the context of an internship in a company, in which the author was integrated into the automation testing team, and includes both the study of possible approaches to the development of support tools, his implementation and the study of the tool's impact in the testing process.

Keywords : automatic testing, acceptance tests, fail management.

Tabla de contenidos

Índice de figuras

Ilustraciones

Tablas

1. Introducción

1.1 Motivación

La calidad del software se ha convertido en los últimos años un factor prioritario para las empresas del sector, como muestra el hecho de que la inversión en este apartado se ha visto aumentada de una media del 18% del presupuesto de un proyecto en 2012 a un 35% en 2015 [1] y se espera que llegue al 40% antes del año 2018. Con ello las empresas pretenden proteger la imagen corporativa, uno de los factores clave para atraer a nuevos clientes y fidelizar a los existentes, así como crear una conciencia de calidad, frente a la tendencia recurrente hasta unos años atrás de descuidar este aspecto debido a los compromisos y limitaciones de tiempo y presupuesto, y garantizar también la experiencia final de usuario, lo cual, recordemos que es, en última instancia, aquello que da valor al producto en esencia.

En este aspecto, han surgido nuevas herramientas y metodologías para facilitar, mejorar y hacer más eficiente este aspecto. El testeo automatizado siempre se ha perfilado como uno de los elementos capaces de revolucionar esta tarea, puesto que el testeo es, por su propia definición como una secuencia de instrucciones a seguir que debe devolver un resultado conocido, una tarea computable y, por tanto, susceptible de ser automatizada. Además, soluciona el problema de la falta de imparcialidad de las personas, que, a pesar de la experiencia y profesionalidad que puedan tener, suelen ser menos críticas con el trabajo que realizan ellos mismos o sus compañeros.

El tipo de testeo al que nos referimos aquí es el llamado testeo funcional, aquel que se encarga de verificar tanto la apropiada aceptación de datos como su procesamiento y recuperación, y la adecuada implementación de las reglas de negocio que debe seguir el sistema.[7] En este caso, para generar estas pruebas funcionales se hace uso de metodologías ágiles para la definición de las llamadas Pruebas de Aceptación (PA) y así comprobar la cobertura de los requisitos por parte de la aplicación, para después extraer de estas definiciones las de las propias pruebas funcionales.

Todo esto pretende transformar la idea obsoleta del llamado “happy testing”(testeo arbitrario y sin metodología) en un proceso acotado, repetible y que permita detectar cuanto antes la mayor parte de los errores. La consecución de este objetivo supondría un cambio de paradigma en el testeo de aplicaciones, además de una mejora notable en la calidad del software producido.

El contexto en el que se realiza el presente trabajo es el de una empresa de software que desarrolla y mantiene un ERP para el sector socio-sanitario, además de otros productos complementarios. El producto lleva más de 10 años de desarrollo y mantenimiento, lo cual ha permitido que su funcionalidad sea muy amplia. En este sentido, la empresa tiene establecido un proceso de calidad, integrado en el proceso del desarrollo de software, que incluye la ejecución de baterías o suites de pruebas de regresión automatizadas sobre las sucesivas versiones del producto, para garantizar que los cambios no introducen defectos en la funcionalidad.

Este proceso, el cual lleva asociada una considerable inversión de recursos, presenta sin embargo el inconveniente que las pruebas no devuelven unos resultados todo lo útiles que deberían ser. De las pruebas fallidas, sólo un pequeño número resulta en fallos reales de la aplicación, la mayor parte se deben a fallos provocados por la infraestructura sobre la cual se ejecutan las pruebas, así como cambios menores en la interfaz que requieren que se revisen constantemente la correspondencia entre la representación en el código de las pruebas de los objetos de la Interfaz de Usuario (IU) de la aplicación y la propia interfaz de la aplicación, lo que se conoce como fallos de mapeo.

Estos fallos suponen un problema doble. En primer lugar provocan un posible enmascaramiento de fallos reales, puesto que si una prueba no se ejecuta por completo no se cubre toda la funcionalidad que está prevista. Si un falso fallo provoca la detención de la prueba antes de tiempo, no podemos asegurar que no haya un fallo de aplicación en la parte de la prueba que no se ha ejecutado. En segundo lugar, obliga a realizar una ardua tarea de revisión y reparación de estos fallos, que en realidad no pertenecen a la aplicación; tiempo que no se dedica a la revisión y estudio de los fallos reales. Además, en el momento de iniciar este trabajo no había ningún método para discernir rápidamente un fallo real de uno falso, todo recaía en la experiencia del equipo de desarrollo. Por tanto, tampoco era posible priorizar la resolución de los fallos reales en detrimento de los falsos.

En definitiva, estos fallos comprometen los resultados de todo el proceso de QA, ya que recortan la cobertura de los casos de prueba y consumen tiempo de revisión de los fallos reales, con lo que en ocasiones no se pueden resolver a tiempo y llegan a la versión de producción. La empresa ha invertido muchos recursos para reducir el número de falsos fallos, desgraciadamente, se trata de un problema intrínseco que no puede ser resuelto por completo. Por tanto, la creación y uso de algún tipo de herramienta que permita clasificar los resultados se muestra como una de las vías para, en primer lugar, priorizar los fallos reales, y en segundo lugar, estudiar los fallos falsos para así reducirlos de una forma más óptima. Esta solución podría compararse con el cambio de paradigma que supusieron los entornos distribuidos en los sistemas de información de hoy

en día: además de intentar evitar la caída de los nodos (falsos fallos, en nuestro caso), se usan técnicas automáticas para detectar estas caídas y mitigarlas, y así hacer que sea transparente para el usuario final.

1.2 Objetivos

El desarrollo del presente trabajo tiene como objetivo abordar la problemática del estudio de las posibles formas de hacer más eficiente y efectiva la tarea de estudio de los resultados de la ejecución de las pruebas automatizadas mediante herramientas de apoyo. Su función sería de criba y predicción, más o menos precisa, de la probabilidad de un fallo sea real o falso sin entrar al detalle de la ejecución. Los objetivos de estas herramientas, serían:

- **Priorizar la resolución de los fallos de aplicación,** en detrimento de los falsos. Esto permitiría a los testers comenzar el análisis de los fallos que tienen una mayor probabilidad a priori de ser debidos a fallos de aplicación, para así reportarlos cuanto antes, en lugar de realizar el análisis siguiendo un orden arbitrario.
- **Identificar y categorizar los falsos fallos.** Esto permitiría analizar las causas que provocan los falsos fallos, y así solucionarlas de una forma más certera. Es decir, que si se descubre que la mayor parte de los falsos fallos son, por ejemplo, debido a la falta de memoria física de las máquinas de entorno de pruebas, se podría aumentar esta característica para reducir el número de fallos.
- **Dar una base para la gestión automática de los falsos fallos.** Algunos fallos falsos tienen una estrategia de resolución automatizable, si se conocen las causas del mismo. Por ejemplo, la mayor parte de los fallos de infraestructura son arbitrarios, y la estrategia más sencilla y efectiva para resolverlos a corto plazo consiste en repetir la prueba. Si se detecta que un fallo tiene una alta probabilidad de ser un fallo de este tipo, podría programarse la herramienta de ejecución para que repitiera la prueba automáticamente y desechara el fallo falso.

1.3 Estructura del trabajo

El presente trabajo se estructura de la siguiente forma:

El capítulo 1. contiene la introducción, en la cual se explica las circunstancias que motivan su desarrollo, se exponen los objetivos del mismo y se muestra su estructura.

En el capítulo 2. mostramos el estado del arte de la revisión de suites describiendo el proceso de aseguramiento de calidad que hay implantado en la empresa. También incluye una exposición de las posibles herramientas similares a la que se desarrolla más adelante, que pueden servir de base o de las cuales se pueden adoptar ciertas ideas.

En cuanto al capítulo 3. ilustra la problemática actual del testeo automatizado en estudio, es decir, detalla con cifras y datos el tamaño del problema al cual nos enfrentamos y que intentamos solventar, para así poder compararlos con los datos que arroje el proceso de testeo después de la implantación de la herramienta.

El capítulo 4. se centra en el diseño de la herramienta, que en nuestro caso se materializa como una extensión de la funcionalidad de una herramienta ya existente, para cumplir con los objetivos marcados. En él se exponen varias alternativas de diseño, se enumeran los puntos fuertes y las debilidades de cada uno y en la última parte se justifica y muestra el algoritmo escogido para ser implementado, y se detalla el proceso de implementación de la misma.

El capítulo 5. muestra y estudia el efecto que la herramienta tiene en el proceso de testeo, tanto ejecutando pruebas sobre resultados ya estudiados sin la herramienta como sobre las nuevas versiones del producto que se desarrollan tras la puesta en marcha de la herramienta, y evalúa si cumple con las expectativas.

Por último, el capítulo 6. contiene una recapitulación sobre todo el trabajo realizado, para después presentar las conclusiones finales extraídas del trabajo realizado, así como una serie de ideas sobre cómo enfocar el trabajo futuro a realizar en este tema.

2. Estado del arte. Revisión de suites

En este apartado vamos a abordar la problemática y los procedimientos actuales del proceso de revisión de resultados de pruebas. El proceso de testeo aquí mostrado lleva un trabajo acumulado a lo largo de varios años y, a juzgar por la escasa información disponible respecto a los problemas encontrados en su realización, podemos afirmar que probablemente se trate de uno de los más avanzados a día de hoy en cuanto a aseguramiento de la calidad mediante la ejecución de pruebas automatizadas.

2.1 Descripción del proceso

El presente trabajo se desarrolla, como ya se ha adelantado, en el contexto de una empresa de desarrollo de software que desde hace más de 10 años desarrolla un ERP, un producto software de gestión para el sector socio-sanitario. Como cabe esperar, en estos diez años el software ha alcanzado un tamaño considerable y la empresa está compuesta por un gran número de desarrolladores dedicados tanto a este producto como a otras herramientas complementarias.

El proceso de desarrollo utilizado en la empresa utiliza metodologías ágiles para realizar un desarrollo iterativo e incremental, según el cual aproximadamente una vez al mes se libera una nueva versión del producto y se inicia el desarrollo de la siguiente, que se ha ido perfilando, es decir, se ha acordado qué cambios y nuevos requisitos introduce en el software durante la elaboración de la versión anterior. La fase en la cual se introducen nuevas funcionalidades en el software se denomina fase de desarrollo, y cuando esta acaba, se inicia un plazo para corregir fallos sin introducir nuevas funcionalidades antes de que la versión llegue al cliente, denominado fase de preproducción. No entraremos en más detalles del proceso de desarrollo, puesto que este trabajo se centrará en el proceso de testeo que va asociado.

Este proceso de testeo consiste en tres tareas principales, que se desarrollan dependiendo de las distintas fases del proceso de desarrollo: ejecución y revisión de baterías de pruebas automatizadas; refactorización de las pruebas ya existentes y automatización de nuevas pruebas. Pasamos a detallar en qué consiste y en qué momento se desarrolla cada una de estas tareas:

- **Ejecución y revisión de pruebas.** Las baterías de pruebas automatizadas completas (es decir, aquellas que incluyen todas o casi todas las pruebas disponibles) se ejecutan de dos formas: en el momento que el proceso de desarrollo se pasa a la fase de preproducción, cuando se lanza la batería completa para comprobar los fallos de la versión antes de que se libere, y en la fase de desarrollo puede realizarse alguna ejecución aunque no sea prioritario resolver los fallos, para adelantar trabajo futuro.

  También hay suites parciales que se ejecutan bajo demanda del equipo de desarrollo cuando se realiza un cambio importante. Un caso especial de estas suites parciales es la suite de versión, que se compone de todas aquellas pruebas en las cuales interviene una funcionalidad modificada en la versión y que es más probable que resulte afectada. Esta suite de versión también se ejecuta en el período de desarrollo. Ejecutar una suite implica realizar después una revisión de los fallos en busca de fallos de aplicación, diferenciándolos de los fallos en las pruebas provocados por otras causas. Es en esta tarea en la que se centra el presente trabajo.

- **Refactorización de pruebas existentes.** Esta tarea se realiza principalmente en el periodo de desarrollo, aunque también se realiza en parte en preproducción a la vez que se corrigen o actualizan algunas pruebas fallidas. Consiste en modificar la composición interna de las pruebas para que se realicen en menos tiempo, se reutilicen más elemento o, en general, se mejore cualquier aspecto de la calidad interna de las pruebas.

- **Automatización de nuevas pruebas.** Esta tarea se realiza en el periodo de desarrollo, en el cual se reciben o se diseñan nuevas especificaciones de pruebas de sistema (PS) y se crean nuevas pruebas automatizadas. El tiempo dedicado a esta tarea es muy variable, ya que es la tarea que se realiza cuando no es necesario revisar ninguna suite ni realizar ninguna refactorización importante de las pruebas, por ser estas otras tareas más prioritarias.

Una vez conocido el proceso de testeo automatizado, vamos a proceder a definir algunos conceptos que ya se han introducido, pero que aún no se han explicado debidamente, a la par que presentamos el detalle del proceso de revisión de pruebas. Para ello, iremos definiendo las unidades más básicas (las pruebas automatizadas) para llegar a las baterías completas y el software utilizado en la empresa para la tarea de revisión de las mismas.

Denominamos prueba automatizada a un script en un lenguaje de programación concreto (Visual Basic en nuestro caso) que interpreta un motor de ejecución de pruebas automatizadas (Rational Functional Testing, en el caso de estudio). Estos scripts básicamente incluyen instrucciones de

interacción con la IU, como pulsar botones o introducir datos en tablas, y comprobaciones de los datos con los valores esperados, por ejemplo en las etiquetas y tablas de la aplicación o en los ficheros que puede generar en ciertos momentos.

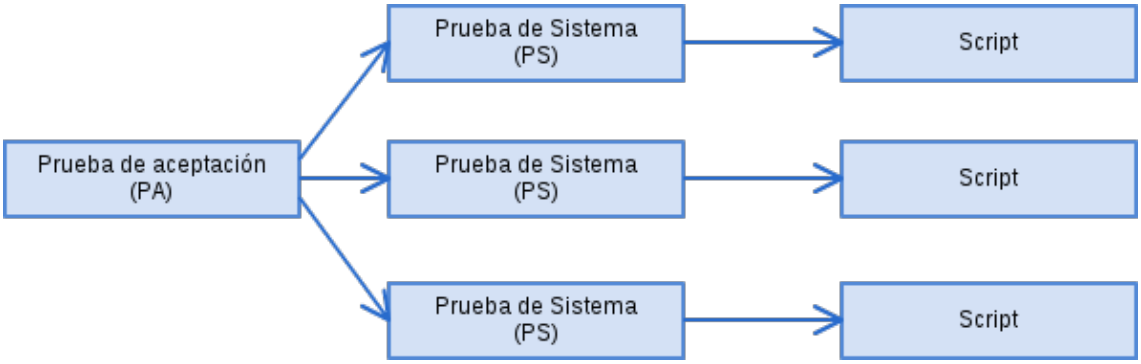

Ilustración 1: Relación entre Prueba de Aceptación, Prueba de Sistema y Script

Cada script es diseñado a partir de la definición de una prueba de sistema (PS), que recoge las instrucciones (en lenguaje natural) necesarias para comprobar el funcionamiento externo de una determinada funcionalidad. Como muestra la figura 1, cada PS lleva asociada una prueba de aceptación (PA), que es una definición más general de una determinada funcionalidad. Por ejemplo, si dado un conjunto de datos tenemos una PA que define cómo se exportan esos datos, podemos tener tantas PS específicas como formatos posibles de exportación tengamos. Aclaramos que una PA puede tener asociadas múltiples PS, pero no al contrario.

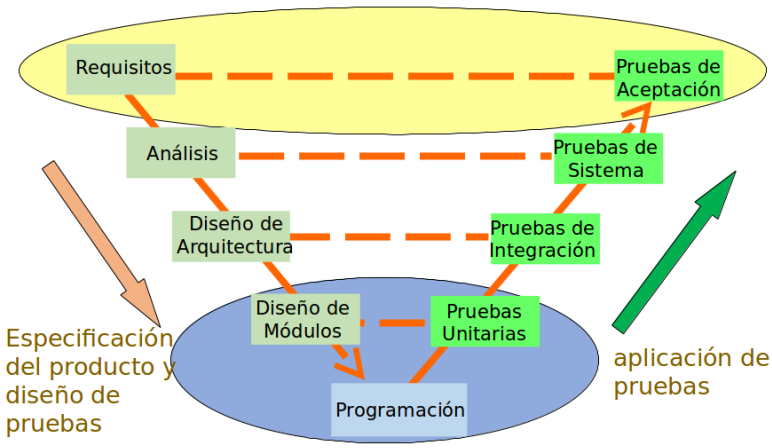

Ilustración 2: Modelo V: Relación entre especificación del producto y aplicación de pruebas

En esta ilustración 2 podemos contextualizar los tipos de pruebas respecto de los niveles de abstracción del diseño de software. Así, una prueba de aceptación comprueba que se satisfacen los requisitos acordados con el

cliente, mientras que una prueba de sistema comprueba que se haya implementado el modelo especificado por los analistas. En el caso de estudio no se entra a nivel de pruebas de integración ni unitarias, puesto que no entra dentro de las pruebas que se automatizan en la empresa.

La revisión de los resultados de las baterías de pruebas es una tarea compleja, debido a las múltiples causas que pueden provocar un fallo, y costosa, debido a la ingente cantidad de información que se maneja. Cada prueba genera un registro o log del resultado de la misma. Este log consiste en un documento HTML en el que se muestra información relativa a cada una de las comprobaciones que se realizan durante la misma, como el que se observa en la ilustración 3:

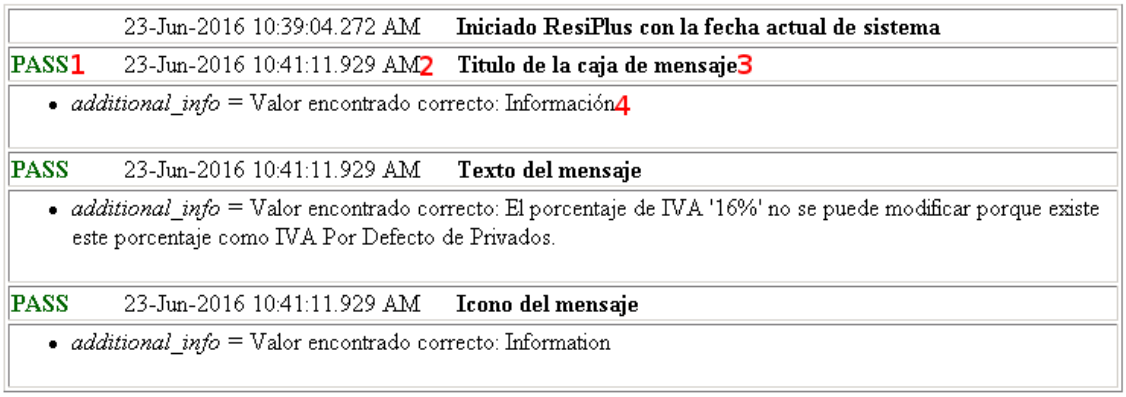

Ilustración 3: Ejemplo de log de una prueba

Por cada mensaje emitido por las comprobaciones que realiza el motor de ejecución de pruebas automatizadas se añade una nueva fila en la tabla del documento. Podemos observar que un mensaje consiste de: (1) el resultado de la comprobación (PASS, FAIL o WARN), en el caso de que no se trate de un mensaje de información, como el primero, que sólo informa de la configuración con la que se inicia la aplicación; (2) la fecha y hora en la cual se efectuó la comprobación; (3) una breve descripción de la comprobación y (4) un espacio opcional donde se muestra información adicional. En este apartado podemos encontrarnos desde un mensaje describiendo el contenido de un campo cuyo valor se compruebe, hasta un volcado de la pila de llamadas y una captura de pantalla en el caso de que se produzca una excepción. Una prueba se considera pasada si no hay ningún mensaje de WARN (alerta) o FAIL (fallo), pasada con aviso si hay al menos un mensaje de aviso y fallada si hay al menos un mensaje de fallo.

Denominamos suite o batería a un conjunto de pruebas, que suelen agruparse siguiendo diferentes criterios, los más comunes son por módulos de la aplicación o por pruebas susceptibles de fallos en una versión concreta. En cuanto al número de pruebas por suite, en el caso de estudio el tamaño es muy variable, pero generalmente las suites parciales (que sólo comprueban determinados módulos de la aplicación) no suelen superar las 400 pruebas, mientras que la batería total (que incluye todas las pruebas

automatizadas creadas para la aplicación que no han sido descartadas) contiene más de 4000 pruebas y se encuentra en crecimiento constante. Por tanto, resulta evidente la necesidad de organizar adecuadamente esos datos, así como extraer la información relevante de los registros, aunque estos sigan accesibles si se desea un mayor nivel de detalle.

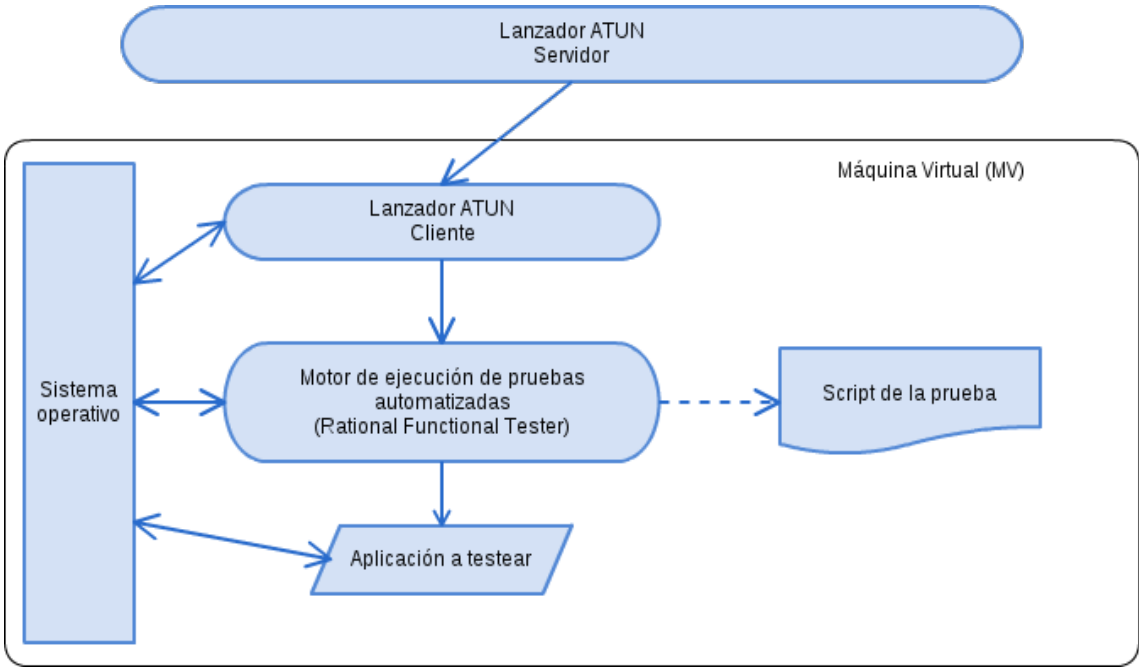

Ilustración 4: Esquema de la ejecución de una prueba en el entorno de máquinas virtuales

Como muestra la ilustración 4, las suites se ejecutan sobre un conjunto de máquinas virtuales (MV) dedicadas, que incluyen el sistema operativo, el motor de ejecución de las pruebas automatizadas, la versión de la aplicación que se desea testear y la versión cliente del lanzador de pruebas, que recopila y guarda en base de datos los resultados. Este lanzador, llamado “Lanzador ATUN”, es una aplicación ad-hoc creada en la empresa. La parte servidor de la misma permite crear, gestionar y lanzar las baterías de pruebas, y acceder al histórico de los resultados, además de otras utilidades, como gestionar las MV, reportar fallos de aplicación, etc. De todas estas funcionalidades, para la creación y desarrollo de la herramienta nos centraremos únicamente en la pestaña llamada histórico de resultados (histórico de logs).

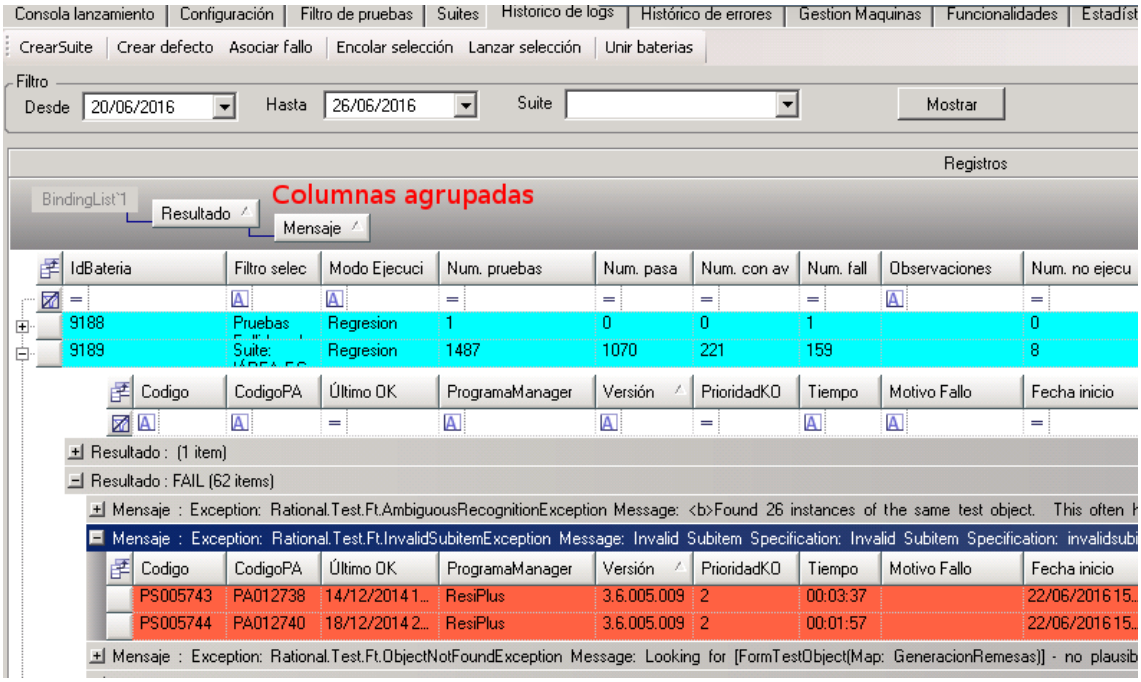

Ilustración 5: Captura del Lanzador ATUN. Pestaña de histórico de logs

Como vemos en la captura de la ilustración 5, el lanzador nos muestra un gran número de datos de cada prueba, una vez hemos seleccionado un intervalo de tiempo y hemos pulsado el botón “Mostrar”. También permite agrupar los resultados según algunos campos, arrastrando una columna al espacio señalado en la captura como “Columnas agrupadas”. En este ejemplo se ha seguido el procedimiento habitual para facilitar la visualización, agrupando jerárquicamente los siguientes campos:

- **Resultado:** El resultado de la prueba: PASS si ha pasado, WARN si ha pasado con aviso o FAIL si ha fallado. También pueden encontrarse los estados <ENCOLADO> si la prueba aún no se ha ejecutado, o vacío en el caso de que se encuentre en ejecución.
- **Mensaje:** El mensaje aquí se corresponde con el primer mensaje de error emitido en las pruebas falladas (la descripción de la primera fila del documento HTML que tiene el mensaje FAIL) o el primer aviso en las pruebas pasadas con aviso. Para ver el resto de mensajes de error es posible hacer doble click en la fila y acceder al documento que se ha expuesto antes.

El hecho de agrupar por el resultado permite filtrar las pruebas falladas de las pruebas con aviso o no ejecutadas. Por otra parte, un mismo fallo generalmente emitirá el mismo mensaje en todas las pruebas que a las que afecte (aunque un mismo mensaje puede ser provocado por distintos fallos), y cuanto más se repita ese mensaje, a más pruebas afecta, lo que lo vuelve más prioritario.

El resto de campos, que no se han agrupado, se muestran en la fila del registro. Los más relevantes son:

- **Código:** El código de la prueba de sistema (PS), identifica a la prueba de forma unívoca.
- **CódigoPA:** El código de la PA asociada a la PS.
- **PrioridadKO:** Se considera más prioritario revisar los fallos nuevos, es decir, aquellas pruebas que pasaron la última ejecución, ya que esto puede implicar un fallo introducido en la última versión del software, mientras que aquellas que ya fallaban anteriormente puede que se deba a un fallo ya reportado, puede deberse a un problema menor que requiera una solución muy compleja, puede que la prueba sea candidata a ser desechada debido a que a perdido utilidad, etc. A las pruebas que en la última ejecución dieron OK se les asigna prioridad 1 y a las que fallaron se les asigna prioridad 2.
- **Último OK:** En el proceso de testeo automatizado se llegó a acumular una gran cantidad de pruebas que fallaban siempre por causas ajenas a la aplicación y que no eran fallos de aplicación. Por ejemplo, algunas pruebas no funcionaban cuando se ejecutaban en las MVs pero al ejecutarlas en local o manualmente, sí que funcionaban. Para priorizar aquellas que empezaron a fallar se introdujo este campo, que indica la fecha del último PASS de la prueba. Este campo es complementario al anterior, ya que a una prueba que fallara dos veces seguidas se le asignaba la misma prioridad que una que fallara desde hace dos años. De hecho, una de las mejoras para el proceso consistió en eliminar de la suite total las pruebas que fallaban desde una fecha anterior a dos años atrás, para mejorar la legibilidad de los datos.
- **Motivo fallo:** Una vez se ha averiguado la causa del fallo, el tester puede introducir la causa en este campo. Las posibles causas son:
  - Fallo de entorno: Error del entorno en el cual se ejecuta la prueba (Error del sistema, falta de memoria, etc.)
  - Prueba mal diseñada: Ocurre cuando el diseño de la prueba no se corresponde con el funcionamiento correcto de la aplicación.
  - Fallo de aplicación: El único tipo de fallo real, aquel que es provocado por un funcionamiento incorrecto de la aplicación.
  - Script mal programado: Ocurre cuando el script de la prueba no se corresponde con el diseño o contiene errores.
  - Fallo de script debido a un cambio de funcionalidad: Similar al anterior, pero aquí no se debe a un error del programador del script sino a que debe ser actualizado.

- Fallo de análisis: Aquellos errores provocados por un fallo durante la fase de análisis.
- Falso FAIL: Fallo genérico, no debido a error de aplicación.

  En la práctica a los fallos de aplicación se les asigna un defecto y se abre una incidencia para que sean resueltos y el resto se asignan como “Falso FAIL”. Esto es debido a que el grupo de trabajo es pequeño (4 personas) y la comunicación puede ser directa. En el caso de que el grupo de trabajo aumentara puede ser interesante hacer una clasificación más exhaustiva.

Como se puede ver, los datos básicos que se utilizan para hacer la primera criba y separar o priorizar los posibles fallos de aplicación de los falsos son básicamente el resultado y el mensaje. También utilizamos la estrategia de priorizar los mensajes más repetidos, ya que puede que estén provocados por el mismo defecto y que se puedan resolver de golpe. Por ejemplo, un fallo en el final de un proceso muy específico probablemente sólo afectará a una PS, sin embargo, un fallo en el lanzador de la aplicación afectará a todas. El que el mensaje se repita con mucha frecuencia no implica que el fallo sea de aplicación, ya que puede ser un problema de mapeo de un control muy utilizado, pero el hecho de solucionar un falso fallo que afecta a varias PS mejora notablemente la cobertura y permite descubrir fallos de aplicación en zonas que debido a ese fallo no se han llegado a explorar.

A pesar de estas y otras estrategias utilizadas para realizar esta tarea, la probabilidad de acertar o fracasar en la priorización de los fallos acaba dependiendo del azar, lo que pone de manifiesto la necesidad de usar un método más científico para resolver este problema.

2.2 Ejemplos de herramientas similares

Mostraremos ahora las soluciones que podemos encontrar en el mercado para la ejecución y visualización de resultados de baterías de pruebas automatizadas. En otras palabras, se expondrán las alternativas comerciales al ya comentado lanzador ATUN que se utiliza en la empresa del estudio.

Destacar que en este apartado ha sido excepcionalmente difícil encontrar herramientas similares, ya que es un área que aún no está muy explotada y las opciones que ofrece el mercado no parecen orientadas a la ejecución masiva de pruebas automatizadas. Intuimos que la mayor parte del mercado aún no utiliza baterías tan extensas como las aquí expuestas para testear su software, sin embargo, no dudamos de más pronto que tarde la ejecución de suites de pruebas automáticas se hará una tarea crucial e indispensable en el desarrollo de software.

Rational Functional Test Manager (IBM)[1]

El motor de ejecución de pruebas automatizadas Rational Functional Tester incluye un software para visualización de resultados llamado Test Manager[4]. Esta herramienta ofrece una escueta visualización de resultados, dividiendo los posible resultados de la prueba en pasadas, falladas, con aviso y con mensaje de información, para obtener más información acerca del resultado de la prueba resulta necesario estudiar detenidamente el detalle de cada una de las pruebas, que tienen asociadas un log idéntico al mostrado al comienzo de este capítulo, ya que en la empresa del caso de estudio se utiliza Rational Functional Tester para la ejecución de los scripts de pruebas automatizadas que ya hemos ido comentando.

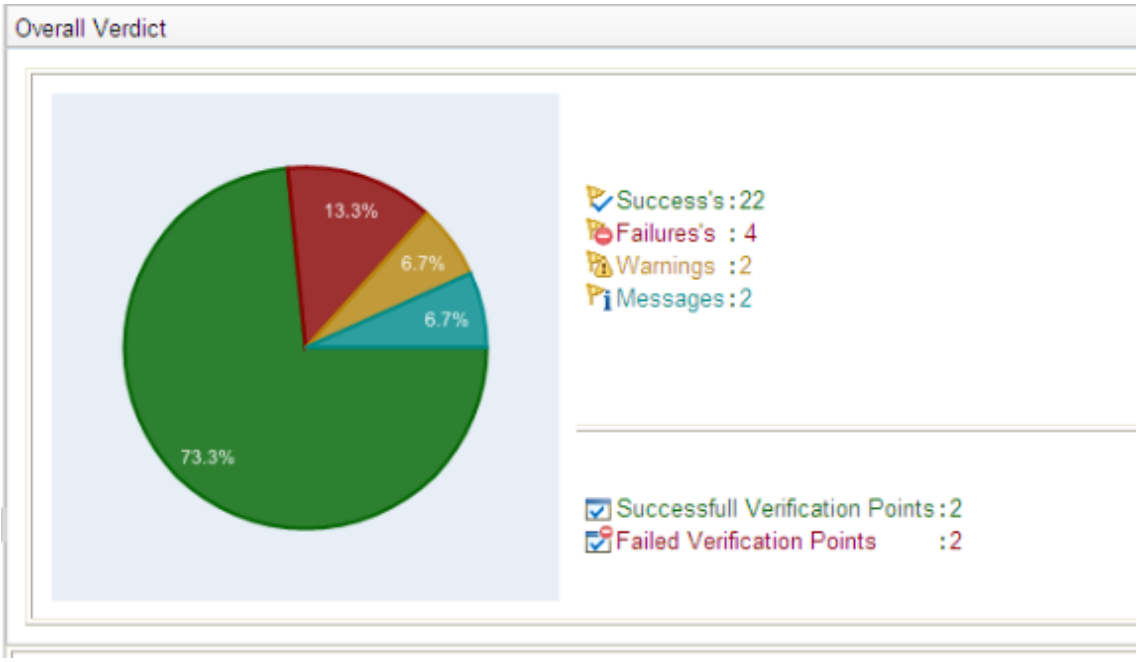

Ilustración 6: Rational Test Manager (IBM)

Rational Quality Manager (IBM)[2]

La versión renovada del Test Manager que IBM ofrece es el Rational Quality Manager, una solución para la gestión de la calidad del software que incluye, entre muchas otras funcionalidades, un visor de fallos de baterías de pruebas[5]. Sin embargo este visor es idéntico al utilizado por el Rational Test Manager, excepto en el tipo de diagrama mostrado, que ahora admite distintos usuarios ejecutando pruebas. No incluye ningún tipo de novedad respecto a la clasificación de los fallos, igual que su predecesor. En apariencia se ha potenciado la gestión de requisitos, así como el seguimiento integral del proceso de calidad del software en detrimento del apartado de testeo automatizado.

1. http://www-03.ibm.com/software/products/en/functional

2. http://www-03.ibm.com/software/products/es/ratiqualmana

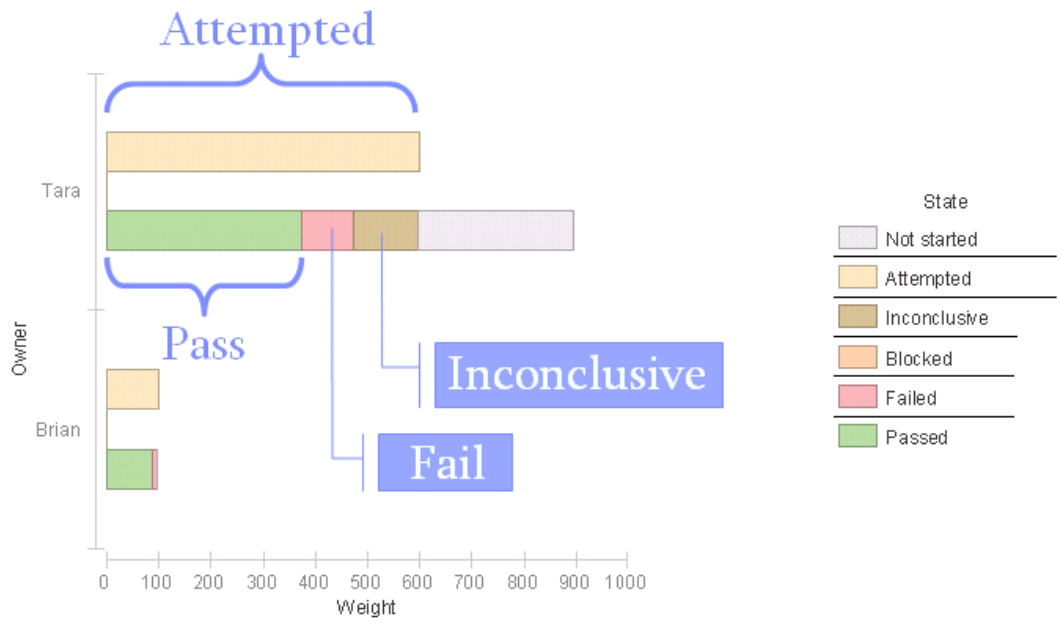

Ilustración 7: Rational Quality Manager (IBM)

Microsoft Test Manager (Microsoft)[3]

En último lugar, tenemos la solución de Microsoft, Microsoft Test Manager. Este software ofrece una ventana de resultados de ejecución de baterías (“test plan” es como se denomina en este entorno) algo más completa. Incluye estadísticas acerca de los motivos de los fallos y del análisis de los mismos [6]. Sin embargo, esta información debe ser introducida manualmente, sin apoyo de ninguna herramienta automatizada. Por tanto, vemos que en este aspecto, tampoco aporta ninguna solución similar a la que se propone en este documento.

3. https://msdn.microsoft.com/es-es/library/jj635157.aspx

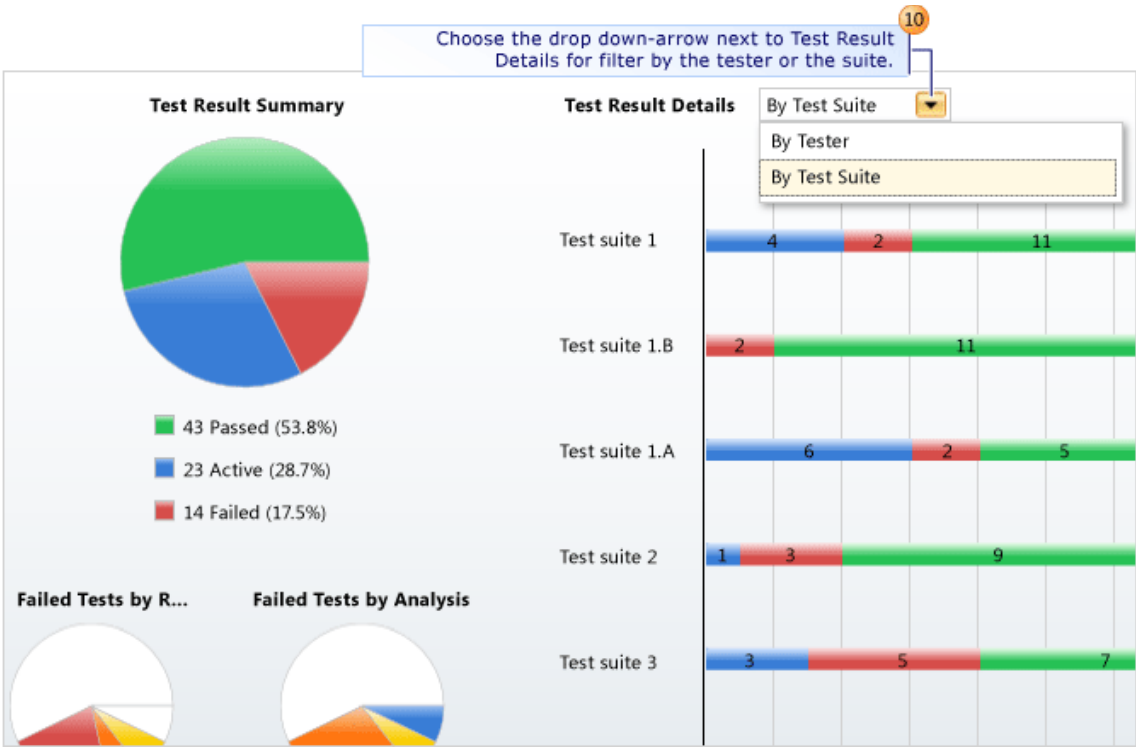

Ilustración 8: Microsoft Test Manager (Microsoft)

En definitiva, las herramientas comerciales encontradas no ofrecen ninguna funcionalidad de apoyo a la tarea de revisión de suites. Ni siquiera parecen orientados a esta tarea, más bien casi todas se orientan hacia la gestión del trabajo de los testers manuales y la revisión global de grupos pequeños de pruebas automatizadas, cuyos resultados no se cuestionan, sino que se asume que son correctos. Esta falta de funcionalidad hace que el desarrollo de la herramienta se inicie desde cero, sin poder contar con otra herramienta en la que basarse.

La herramienta que vemos que ofrece una funcionalidad más avanzada en el terreno que nos ocupa parece sin duda el lanzador ATUN, por su capacidad para clasificar y revisar grandes cantidades de datos. Este hecho y la disponibilidad del código de dicha herramienta para ser modificado, pues recordemos que es un proyecto propio de la empresa, hacen que se tome la decisión de crear la herramienta como una extensión de este lanzador, añadiendo la funcionalidad requerida a la ya existente.

3. Estado inicial del proceso de calidad

En este capítulo expondremos el estado de la problemática de revisión de suites en el momento anterior al inicio del desarrollo de la herramienta, con el fin de justificar con datos las razones que motivaron el desarrollo del presente trabajo y permitir realizar la comparación entre antes y después de la elaboración del mismo, para comprobar la efectividad de la herramienta. Sobre el estado inicial del mismo hay que decir que el autor del trabajo se incorporó junto a otro trabajador a la empresa, con lo que se aumentó el equipo de trabajo de un trabajador a tres. Esto, unido a las pequeñas mejoras que se fueron introduciendo en el proceso, explica la mejora paulatina de los resultados mostrados.

Las siguientes gráficas muestran la evolución de los resultados de las suites aplicadas, en los cuatro meses anteriores al inicio del desarrollo de la herramienta, en junio del año 2016:

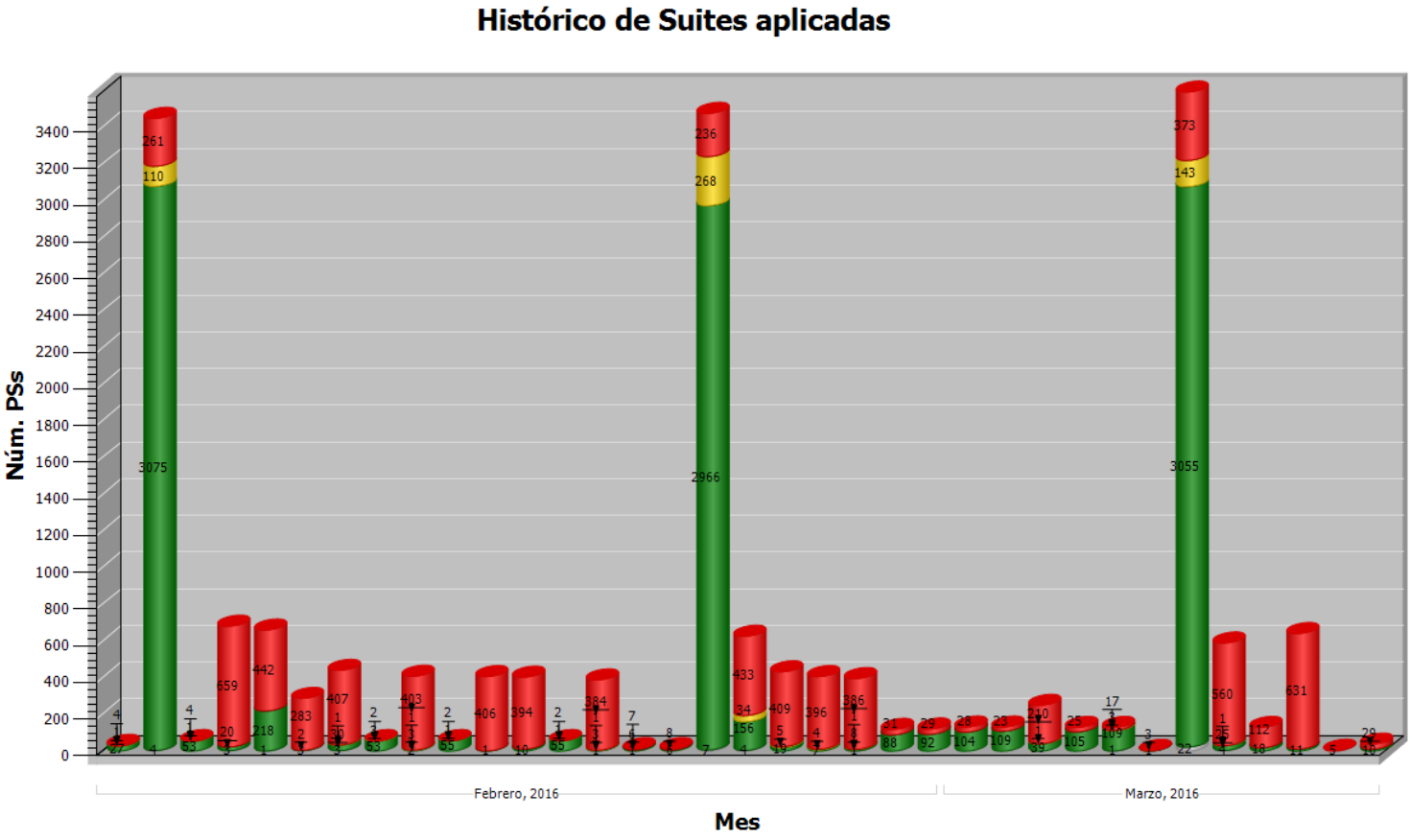

Ilustración 9: Resultados de suites. Febrero-Marzo 2016

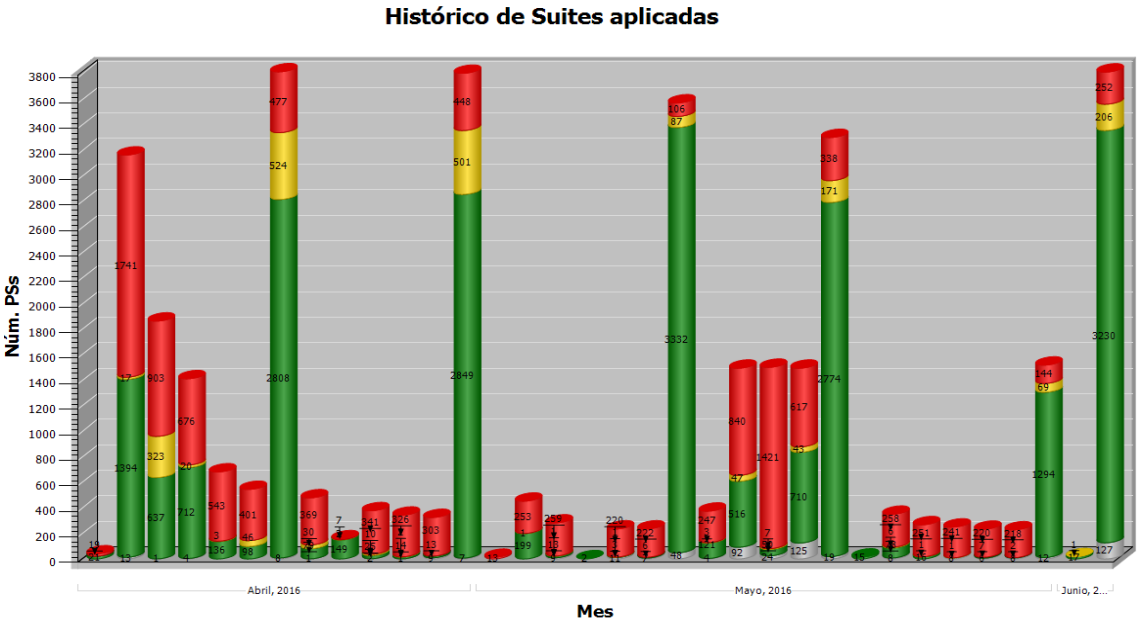

Ilustración 10: Resultados de suites. Abril-Mayo 2016

Estas gráficas ilustran de manera fiel el proceso ya descrito en el apartado anterior para la ejecución de suites: las suites con mayor número de PS (suites totales) se lanzan cuando la versión que está en desarrollo pasa al periodo de preproducción, es decir, antes de que se libere, cuando se dejan de añadir funcionalidades y se inicia la búsqueda exhaustiva de fallos para evitar que lleguen a los clientes. Las suites que siguen a estas suelen tratarse de relanzamientos de las pruebas falladas. Las suites de tamaño medio son, en su mayoría, suites lanzadas bajo demanda para probar determinadas partes de la aplicación.

Se aprecia aquí el volumen de datos manejado, del orden de miles de pruebas y cientos de fallos que se han de revisar. Además podemos observar una suite lanzada a primeros de abril que arrojó más de 1700 fallos. Estas situaciones no son habituales pero tampoco son aisladas: ocurren cuando un fallo, sea del tipo que sea, afecta a un gran número de pruebas, ya sea porque afecta a un elemento vital del programa, o porque se trata de un fallo arbitrario que ocurre con una frecuencia extremadamente alta.

Es en este tipo de casos cuando una adecuada herramienta de clasificación contribuiría en gran medida a agilizar la tarea de revisión de las suites, pues el volumen de datos de fallos aumenta considerablemente, junto con la importancia de resolverlos con rapidez, pues está afectando notablemente al funcionamiento de la aplicación, si finalmente se trata de

un fallo de aplicación, o a la cobertura del proceso de calidad, en el problema del enmascaramiento de fallos de aplicación por falsos fallos que ya se ha comentado.

A continuación, y para complementar la información anterior se muestran las estadísticas de reportes de fallos detectados en testeo automatizado y creación de nuevas pruebas automatizadas hasta el inicio del desarrollo de la herramienta. Es decir, los fallos reales de aplicación detectados a través de las automatizaciones. Los colores reflejan la severidad del fallo, más rojo implica mayor severidad, Hay que aclarar que hasta el mes de abril no se estableció correctamente la metodología de reporte en la herramienta y los datos hasta esa fecha pueden no ser completamente correctos:

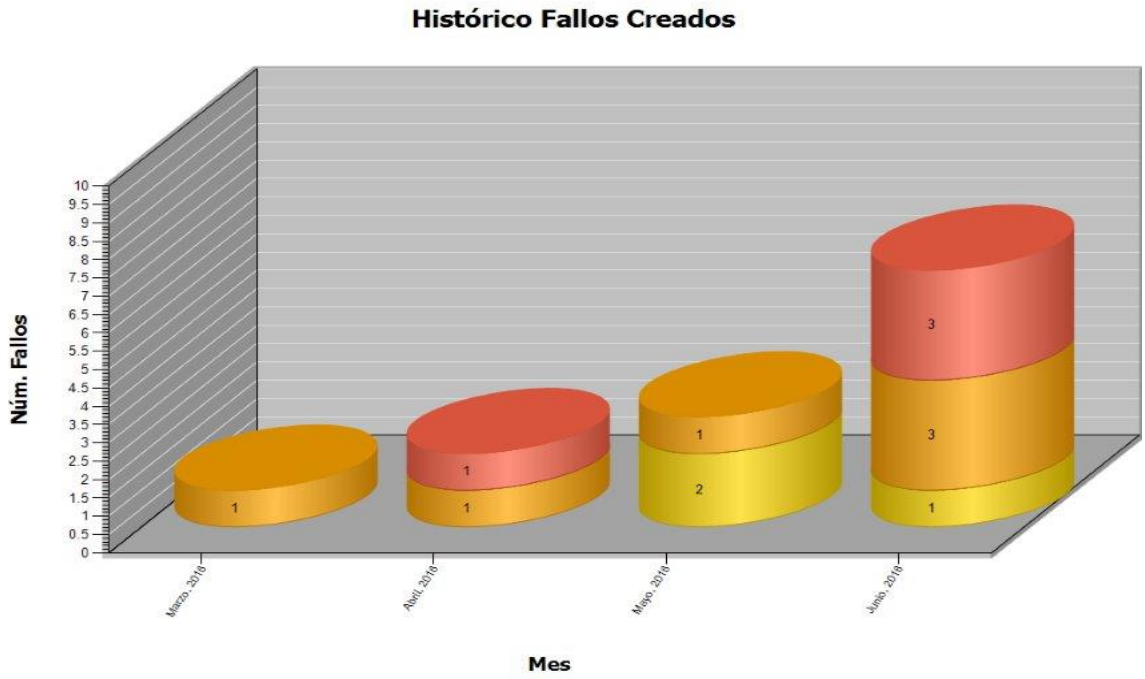

Ilustración 11: Fallos detectados en Testeo Automatizado. Febrero 2016-Junio 2016

Los resultados muestran que, por una parte, del ingente número de fallos de las pruebas, sólo una ínfima parte resultan en fallos reales de aplicación, aunque hay que recordar que un sólo fallo de aplicación puede afectar a un gran número de pruebas (el mes de febrero no se muestra ya que no se registraron fallos). Hemos de recordar que, como ya hemos recalcado anteriormente, el objetivo último del proceso es la detección de estos fallos, y estos resultados son los que los administradores tomarán como referencia para decidir, o no, invertir en el apartado de testeo automatizado. Por tanto, la necesidad de mejorarlos se hace patente.

Por otra parte, se observa una mejora notoria a lo largo del tiempo. Esto se explica por la experiencia del equipo de trabajo y la implantación de una metodología clara para el reporte de fallos. Aquí se muestra el potencial de introducir en una herramienta el conocimiento que los testers obtienen a través de la experiencia, para así reducir el tiempo de formación de los nuevos trabajadores y, así, acortar la espera para ver los beneficios del aumento de personal, o mitigar los problemas derivados de la sustitución de un miembro por otro con menos experiencia, en su caso.

Sin embargo, estos datos por sí solos no permiten aclarar si el número de fallos detectados es grande o pequeño, ya que no tenemos con qué compararlo. Para ello, a continuación se muestran los fallos detectados por testeo manual en el mismo periodo:

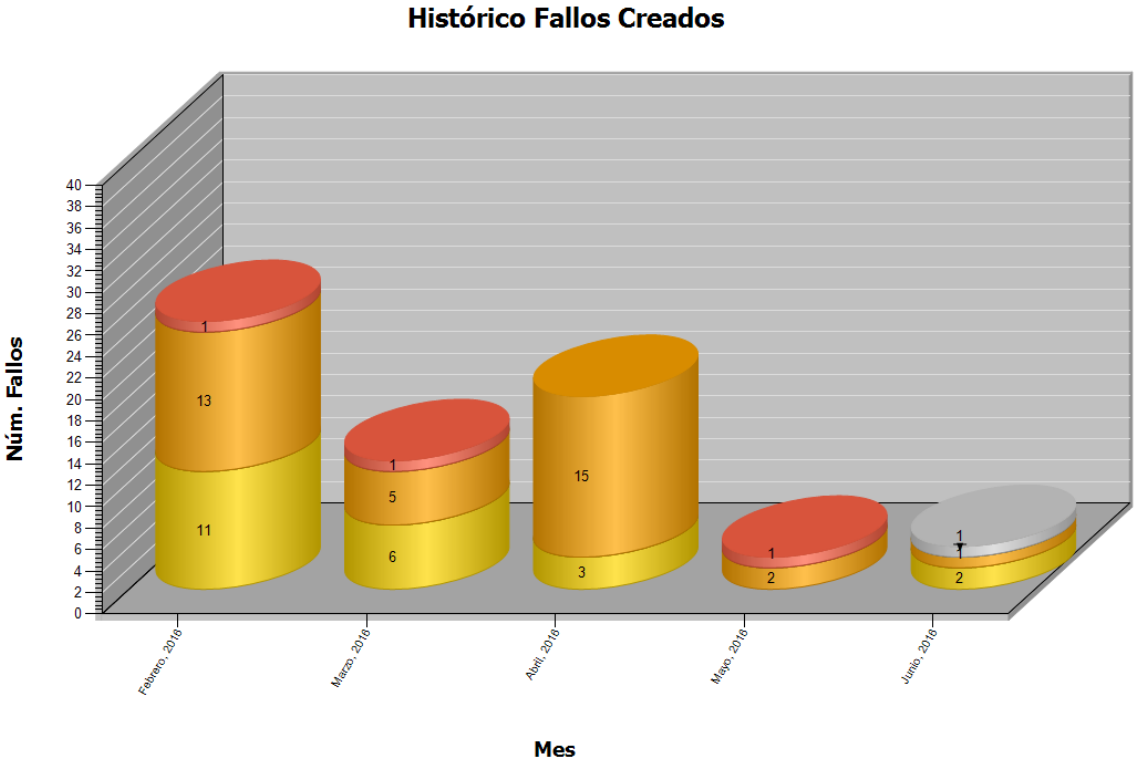

Ilustración 12: Fallos detectados en Testeo Automatizado. Febrero 2016-Junio 2016

Estos resultados desprenden varias conclusiones interesantes. La primera es la gran variabilidad entre unos meses y otros (aunque, como ya se ha dicho antes, durante los primeros meses es probable que algunos fallos detectados en testeo automatizado se clasificaran incorrectamente como detectados en testeo manual). Esto es debido principalmente a la propia naturaleza del desarrollo de software. Unas versiones introducen más cambios que otras, y esos cambios, a su vez, en ocasiones introducen un número de errores superior que otras, que a su vez pueden ser más o menos fáciles de detectar. De una forma o de otra, en los dos últimos meses

se observa que los resultados del testeo manual y automatizado son muy similares, lo que nos indica que, con la metodología adecuada, el testeo automatizado es, al menos, tan efectivo como el manual en cuanto al número de fallos detectados.

Aquí es importante marcar las diferencias entre el testeo manual y el automatizado, ya que, en contra de lo que pueda parecer, no son dos sistemas excluyentes, sino complementarios. Los test automatizados son pruebas de aceptación que han sido definidas, ejecutadas manualmente y se ha diseñado implementado y testeado un script que realiza la prueba, es decir, llevan un proceso de creación bastante laborioso. Es por esto que la mayor parte de las pruebas de este tipo que se realizan son de regresión, es decir, comprueban que los cambios no introducen errores en las partes de la aplicación que ya funcionaban correctamente. Los test manuales pueden ser de regresión o de aceptación, pero como para su creación no es necesario implementar un script, pueden comprobar las nuevas funcionalidades, que no ha dado tiempo a automatizar. Por tanto, una versión que introduzca una nueva funcionalidad muy compleja pero que no afecte a otras partes del sistema es probable que introduzca fallos detectables por testeo manual, mientras que una modificación de una funcionalidad ya existente es susceptible de provocar un gran número de fallos detectables por las pruebas automatizadas.

Como conclusión, podemos decir que, a pesar de que las estadísticas hay que tomarlas con mucha cautela, el testeo automatizado es una tarea cuyo éxito depende en gran medida de la destreza y experiencia del equipo de trabajo en realizar su labor, como ocurre con el testeo manual, aunque en menor medida. En este caso, además vemos que uno de los mayores escollos a la hora de realizar esta tarea consiste en lidiar con el ingente número de falsos fallos. Por tanto, el uso de una herramienta de apoyo a la clasificación de resultados no sólo permitiría facilitar la tarea a los testers, sino que además permitiría mejorar los resultados y hacerlos más independientes del equipo de trabajo.

4. Diseño e implementación de la herramienta.

En el este apartado presentaremos distintas posibilidades de diseño de la herramienta, la cual, puesto que va a formar parte del ya mostrado lanzador ATUN, va a consistir en una extensión de éste. Una vez expuestas las alternativas de diseño, las analizaremos para mostrar la implementación que finalmente se realizará. Sin embargo, aunque sólo se implemente una de las propuestas, la intención del presente trabajo es ilustrar distintas opciones de diseño que puedan cubrir distintas necesidades en otros casos de estudio, es decir, que las soluciones que aquí descartemos podrían ser las adecuadas en aquellos entornos en los cuales la capacidad de inversión en QA, el porcentaje de falsos fallos en las suites o la cantidad de datos sobre testeo automatizado disponible sean diferentes.

4.1 Diseño de la extensión.

En primer lugar, acotaremos que los datos de entrada posibles para el sistema son los registros de fallos que ya hemos explicado en apartados anteriores. Éstos consisten de documentos HTML que pueden incluir o no capturas de pantalla de los errores, además de los otros campos que muestra el lanzador que presentamos en el apartado 2.1 . Para simplificar y hacer el proceso más sencillo, se podría utilizar únicamente el primer mensaje de error de la prueba, de esta forma no sería necesario acceder a los ficheros de las pruebas puesto que esta información ya la recoge el lanzador en uno de sus apartados. La salida del sistema consistirá en un indicador de la probabilidad de falso fallo y un indicador de probabilidad de fallos de aplicación, que pueden ser en forma de porcentaje, número entero o mediante una escala nominal discreta (Alta, Media o Baja, por ejemplo).

La problemática de la clasificación de pruebas puede resolverse mediante el llamado reconocimiento de patrones o formas, con el cual el sistema va modelando qué patrones siguen los elementos que se clasifican en un cierto grupo. Para ello, es necesario entrenar al sistema y disponer de una buena base de datos ya clasificados. Una alternativa más sencilla consistiría en implementar las heurísticas que ya aplican manualmente los trabajadores de la empresa para agilizar su tarea. Aquí plantearemos ambas alternativas, y una tercera consistente en combinar estas estrategias para obtener un método mixto.

Diseño mediante implementación de heurísticas

Esta opción de diseño tiene la ventaja de ser la más sencilla, ya que sólo trata de automatizar los métodos que los testers han ido desarrollando para realizar su tarea con una mayor rapidez y eficacia. Estos métodos, si bien no son completamente fiables ni tienen por qué tener una base totalmente científica o teórica, en la práctica se han demostrado muy útiles para apoyar agilizar la tarea que nos ocupa.

Las posibilidades de optimización en este apartado son prácticamente infinitas, y probablemente la propia tarea de revisión de suites haga que surjan nuevas estrategias de resolución. A continuación se muestran aquellas que, por sus características de eficacia y simplicidad de implementación, se muestran más prometedoras para formar parte de la extensión a desarrollar.

- **Descartar los fallos que funcionan en la segunda ejecución.** Si un error no se repite bajo las mismas circunstancias (misma aplicación, mismo script) es obvio que no puede deberse a un fallo de aplicación. Son fallos debidos al entorno en el que se ejecutan las pruebas, por ejemplo, debido a que aparezca un mensaje del sistema recordando una actualización. Estos fallos son tan comunes que la herramienta ATUN lleva incorporada una opción para relanzar las pruebas que fallan automáticamente, creando una nueva batería con llamada “Pruebas fallidas de la suite XXXX”, donde XXXX es el nombre de la suite original. Para implementar esta heurística bastaría con revisar el resultado de las siguientes ejecuciones de la prueba fallada y ver si hay alguna que haya pasado. En este caso podríamos ir un paso más allá y clasificarla directamente como “Falso FAIL”, ya que no hay posibilidad de que se trate de un fallo de aplicación.
- **Descartar los fallos que en la segunda ejecución fallan en un punto posterior de ejecución.** Esta estrategia es similar a la anterior, ya que sigue el mismo principio. Si bajo las mismas circunstancias, una prueba falla primero en un punto A y después en uno posterior B, el fallo del punto A será un fallo falso. La cuestión es averiguar cuándo una aplicación falla en un punto distinto, y, una vez descubierto, cuál de los dos puntos de fallo es anterior. En muchos registros se muestra una traza de la pila de ejecución y en ella es posible observar en qué línea del script se produjo el error. El problema radica en que esta información no está siempre y que muchas de las pruebas heredan de otras, por tanto, no es trivial averiguar si primero se ejecutará la línea 20 del script o la línea 25 de la base.

  Un método mucho más sencillo consiste en comparar los tiempos de las ejecuciones. Aquella que tarde más tiempo en fallar será la

candidata a fallo de aplicación. Para contemplar las posibles diferencias debidas al entorno, debemos considerar un cierto margen de error, una diferencia mínima de tiempos para aplicar esta heurística. También asumiremos que dos fallos distintos arrojan mensajes distintos, para no aplicar esta estrategia sobre dos ejecuciones con el mismo mensaje.

- **Asumir que los fallos en los que la prueba finaliza normalmente son candidatos a fallo real.** Los mensajes de error que no son debido a una excepción, sino a que fallan las comprobaciones, suelen ser más proclives a deberse a fallos reales que no de falsos fallos. La razón es que la probabilidad de haber introducido incorrectamente los datos de entrada (lo cual provocaría un resultado diferente) sin provocar un final inesperado de la prueba es muy baja, así como la probabilidad de leer incorrectamente los datos de salida. Los únicos casos en los que suele ocurrir esto es cuando, por ejemplo, salta un mensaje de error y se esperaba un mensaje de aviso, entonces se muestra que hay una discrepancia en el mensaje encontrado, pero no hay excepción.

  Este tipo de mensajes se diferencian de los otros en que no contienen la palabra “Exception” y que explican las diferencias entre el resultado esperado y el real. Con estas condiciones, podríamos asignar a estos fallos una probabilidad superior a la normal de ser fallos de aplicación.

- **Asumir que los fallos con mensaje de error del sistema son candidatos a falso fallo.** Aquellos mensajes de error que incluyen referencias a excepciones provocadas por el sistema suelen estar relacionados con fallos en el entorno donde se realizan las pruebas (falta de memoria, error de permisos, etc.). Por ello, para este tipo de mensajes podríamos asignar una probabilidad superior a la normal de tratarse de fallos de entorno.

- **Crear una lista blanca y una lista negra de mensajes con alta/baja probabilidad de ser fallos de aplicación.** Las distintas etapas de desarrollo van mostrando que algunos fallos tienen tendencia a reaparecer por la propia naturaleza de la aplicación, debido a que a ciertas partes les afectan más los cambios entre versiones o que ciertas pruebas resultan menos estables por distintas causas (p.e. su complejidad). Este conocimiento se puede modelar mediante una simple lista de mensajes o palabras clave que se sabe tienden a esconder fallos falsos (lista negra) o fallos de aplicación (lista blanca). El que un mensaje dado se corresponda con algún elemento de estas listas aumentará las probabilidades de que se trate de un fallo de aplicación o de entorno. Esto, además, permite al equipo de trabajo compartir el conocimiento práctico de forma

sencilla y facilita la incorporación de nuevos miembros al equipo, que se benefician de la experiencia acumulada por otros miembros.

La lista de heurísticas puede extenderse tanto como se quiera, pero con la aplicación de la mayoría de estas reglas ya podría hacerse una primera criba para probar la eficacia del uso de este tipo de herramientas. Como se trata de aproximaciones y conocimiento difuso, lo más conveniente sería utilizar una escala discreta de probabilidad de fallo de aplicación de 5 niveles, por ejemplo, “Mínima, baja, normal, alta, máxima”, en la cual inicialmente a todos los fallos se le asigna una probabilidad normal y después cada una de las reglas aumenta o disminuye esa probabilidad, o bien alguna escala numérica similar, por ejemplo, una con números enteros en la que el valor medio sea 0 y el signo positivo o negativo implique una mayor o menor probabilidad de la normal, respectivamente.

Diseño mediante reconocimiento de formas

Como ya se ha expuesto anteriormente, el problema de, dado un registro de error, predecir si se trata de un error de aplicación o no, es un problema que puede ser resuelto mediante técnicas de reconocimiento de formas. Aunque hemos acordado utilizar únicamente el mensaje de error (una cadena de texto) para simplificar, podría estudiarse incluir toda la información del documento HTML e incluso las capturas de pantalla. Incluir estas imágenes permitirían que el sistema llegara a detectar los diálogos del sistema que en ocasiones aparecen y provocan fallos en las pruebas, aunque complicaría considerablemente el diseño del clasificador.

Para este diseño se podría usar cualquier software diseñado para este propósito, como MALLET[2]. Este software en concreto permite clasificar utilizando diversas distribuciones de probabilidad, entre ellas: máxima entropía, árboles de decisión, “Naïve Bayes”, etc. En nuestro caso la opción más interesante a priori sería el uso del clasificador de mediante máxima entropía, que es también utilizado en sistemas de traducción automática. La razón es que, según este principio, “la distribución de probabilidad menos sesgada que se le puede atribuir a un sistema estadístico es aquella en la que dadas unas ciertas condiciones fijas maximiza la entropía, esto es, aquella en la que la desinformación es máxima.”[3] Sin embargo, cabe la posibilidad de probar distintos clasificadores y escoger aquel que dé mejores resultados.

Todos los clasificadores ya citados, requieren de un cierto entrenamiento para su funcionamiento. No puede haber aprendizaje sin estos datos, ya que a partir de ellos se deducen los patrones que posteriormente se utilizan para clasificar los datos de entrada. El tamaño de la base de datos de entrenamiento condicionará en gran medida la calidad del clasificador y la velocidad a la que mejore el mismo. Por tanto, esta opción es inviable sin un buen conjunto de datos ya clasificado correctamente para alimentar el sistema. Además hay que tener en cuenta que estos datos deben provenir

de las propias pruebas realizadas en el entorno en el que se trabaje, no siendo posible adquirirlos, como sí ocurre en otros contextos, ya que los patrones que sigan los mensajes pueden variar.

Esta opción sería mucho más fiable que la anterior, ya que sigue principios estadísticos fiables y demostrados, y además con el tiempo aumenta su experiencia y va mejorando, por lo que sería la solución ideal de disponer de los medios necesarios. Además, si en el futuro se desea ampliar la clasificación, por ejemplo, distinguiendo los posibles fallos de entorno de los fallos de mapeo, el sistema a utilizar sería el mismo con algunas ligeras variaciones y un reentrenamiento moderado del clasificador.

En este caso la escala a utilizar debe ser continua, en forma de probabilidad, siendo además el resultado natural de las herramientas ya mencionadas. Esto implica una precisión mucho mayor en los resultados, así como la posibilidad de tratar ese resultado computacionalmente para obtener una medida mucho más fiable de la posible efectividad de la extensión. Si se descubre que la precisión media de las estimaciones es muy baja, podría razonarse que la extensión no cumple las expectativas y así replantearse el sistema propuesto, con el otro diseño la valoración depende de la impresión personal que el equipo de trabajo, de forma más subjetiva.

Diseño mixto

La última opción de diseño que vamos a exponer consiste en intentar combinar lo mejor de cada uno de los diseños anteriores, es decir, la simplicidad y la independencia de una base de datos de entrenamiento grande de la implementación de las heurísticas con la fiabilidad y la capacidad de mejora continua del aprendizaje automático. Para ello, la manera más sencilla es mezclar los resultados.

Podríamos diseñar un sistema en el cual por una parte se aplicaran las heurísticas para obtener la primera aproximación, por otra parte se calcule la probabilidad mediante un clasificador y después se realice la suma ponderada de ambas cantidades. Cómo realizar la ponderación depende en gran medida de la fiabilidad del aprendizaje automático. Si es muy fiable, podríamos tomar directamente el valor que nos devuelva y luego añadir o quitar algunos puntos (moviéndonos en un margen pequeño, del orden del 10-20%) en función de las heurísticas que hemos comentado. Si, por el contrario, el clasificador no ha sido aún lo suficientemente entrenado y no confiamos en los resultados que arroja, podríamos tomar como base el resultado de la aplicación de las heurísticas y subir o bajar un nivel en la escala si el valor devuelto por el clasificador difiere lo bastante.

En cualquier caso, probablemente el mejor planteamiento de esta opción sea como un diseño transitorio, un proceso en el que partimos de usar el método de aplicación de heurísticas y en el que progresivamente se va

dando más importancia al resultado de los clasificadores. Así en un primer momento, cuando aún no se dispone de una base de datos lo bastante grande, utilizamos el primer método para crear datos fiables y vamos alimentando al clasificador para mejorar su precisión y darle cada vez más importancia. Sería la opción ideal si se desea obtener la máxima fiabilidad pero no se dispone de esta base de datos, de esta forma mientras se generan estos datos podemos aprovechar las ventajas de la herramienta.

4.2 Implementación de la extensión

Tras evaluar cuidadosamente cada una de las alternativas de diseño propuestas y tener en cuenta todas las circunstancias particulares del entorno en el que se desarrolla el presente trabajo, se optó por desarrollar una versión del diseño basado en heurísticas. Las razones que motivaron esta decisión fueron varias, las más destacables fueron que era importante mostrar lo antes posible la efectividad de la extensión para justificar los recursos que se invertían en ella y que, a pesar de contar con una base de datos de registros de fallo de gran tamaño, los datos no estaban clasificados correctamente en la mayoría de los casos, pues esta tarea debía ser realizada manualmente y en ocasiones se omitía este paso, pues el grupo de trabajo era lo suficientemente pequeño para que la comunicación fuera directa.

Tampoco se incluyó en el diseño final la heurística que trataba de la creación de una lista blanca y una lista negra. La razón fue que las primeras pruebas delataron que la mayor parte de las pruebas que se podrían incluir en estas listas ya recibían la probabilidad adecuada, es decir, se clasificaban correctamente. El uso de listas quizá habría ayudado a hacer más explícita esta clasificación, aumentando el valor absoluto del resultado de la probabilidad correspondiente, pero puesto que el calibrado, es decir, la escala que se utiliza para determinar mayores o menores prioridades, no era tan prioritario como el signo, y dado que las listas debían ser compartidas entre los testers, lo que complicaba el diseño, la idea se descartó.

El resultado de todo este trabajo consiste en unas pequeñas modificaciones en el lanzador de pruebas automatizadas ATUN, que ya se mostró anteriormente. A nivel de interfaz, la única diferencia, como se aprecia en la ilustración 13, es la adición de dos nuevas columnas a los registros de las pruebas con la prioridad estimada de que se trate de un fallo de aplicación (1. ProbabilidadFA) y de un fallo del entorno (2. ProbabilidadFE).

Mensaje : Exception: Rational.Test.Ft.WindowActivateFailedException Message: window.activatefailed.unknown (9 items)

Codigo	CodigoPA	Último OK	ProgramaManager	Versión	PrioridadKO	ProbabilidadFA 1	ProbabilidadFE 2	Tiempo
PS005880	PA002920	14/07/2016 12:4...	ResiPlus	3.6.006.013	1	-10	10	00:02:26
PS001940	PA002751	15/07/2016 09:45...	ResiAgenda	3.6.006.013	2	-10	10	00:00:41
PS002257	PA002847	12/07/2016 10:20...	ResiAgenda	3.6.006.013	2	-2	2	00:00:43

Ilustración 13: Lanzador ATUN después de ser modificado.

En cuanto a las modificaciones en la capa de lógica de la aplicación, del conjunto total de las clases implementadas para la aplicación, sólo fue necesario modificar una. Esta clase, llamada ResultadoPruebaHistorico, representa toda la información referente a una ejecución de una prueba dentro de una batería, en otras palabras, contiene la información necesaria

para rellenar cada una de las líneas de la pestaña “Histórico de logs” que ya hemos mostrado. Procedemos a enumerar algunos de los atributos y métodos utilizados en las modificaciones realizadas, con el fin de que sean más comprensibles:

IdBateria: Entero que contiene el identificador de la batería a la que pertenece el resultado.

IdBateriaPadre: Entero que representa el padre de la batería a la que pertenece el resultado, si lo hubiera. Una batería es padre de otra si la segunda se generó a partir de la primera. Esto ocurre cuando se relanza un subconjunto de pruebas de una batería, generalmente fallos que se desea comprobar si se han solucionado o se trata de fallos de entorno.

CodigoDefecto: Cadena de caracteres que contiene el código del defecto o fallo asociados al resultado. Cuando se detecta que hay un fallo de aplicación, se crea un defecto para notificarlo. Si es nulo indica que no tiene defecto asociado, luego no ha sido clasificado como fallo de aplicación (aunque puede ocurrir más adelante).

IdPSistema: Entero que contiene el código de la PS que se ejecutó.

GetResultado(): Método que devuelve un enumerado de la clase “ResultadoPruebaHistorico”, que representa los posibles resultados de la prueba y puede tomar los valores “PASS”, “FAIL”, “WARN”, “UNKNOWN” o “NULL”.

TiempoTotal: Entero que contiene el tiempo total que tardó en ejecutarse la prueba, en segundos.

Mensaje: Cadena de caracteres que contiene el mensaje que devolvió la prueba, que ya se ha explicado anteriormente.

ProbabilidadFA: Entero que representa la probabilidad de que el resultado sea fruto de un fallo de aplicación.

ProbabilidadFE: Entero que representa la probabilidad de que el resultado sea fruto de un fallo de entorno o de mapeo, similar al anterior.

Asimismo, se hace uso de dos métodos de la clase de utilidades DataClassesLogs, que accede a la base de datos:

GetBateriaPadre(int idBateria): Devuelve la raíz del árbol de baterías padre-hijo al que pertenece la que recibe como argumento, es decir, la propia batería si no tiene padre, la batería padre si ésta no tiene padre, etc.

ATUNObtenerBateriasHijas(int idBateria): Devuelve el conjunto de baterías que son descendientes de la batería dada. (hijos, nietos, etc.)

A continuación se muestran los diagramas de flujo del algoritmo. Se ha dividido la funcionalidad en dos métodos, el primero carga los datos de la base de datos y se los pasa al segundo, que los analiza y asigna los valores adecuados a los resultados.

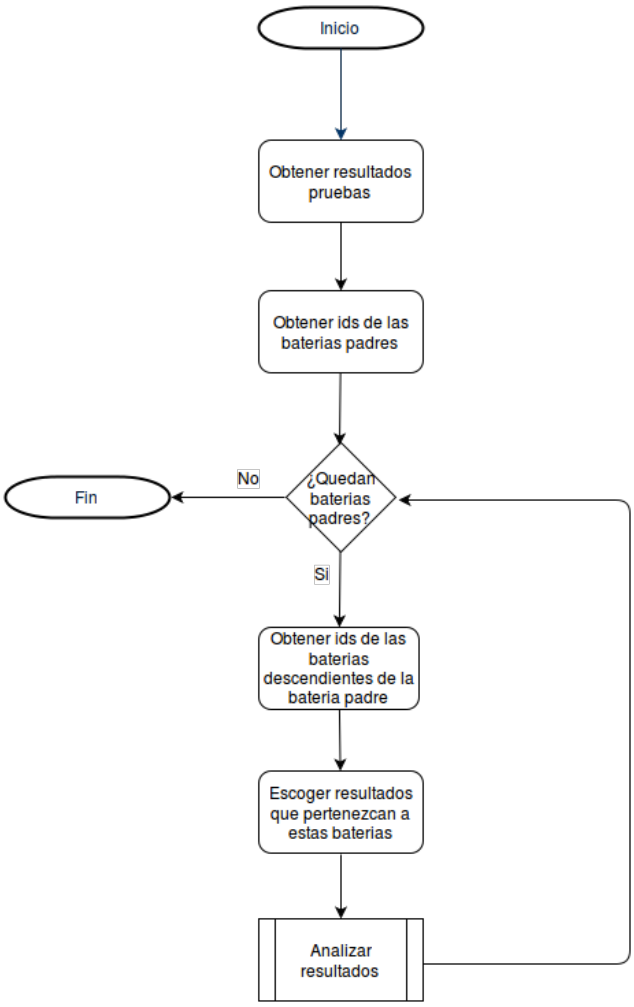

Ilustración 14: Diagrama de flujo del método CalcularProbabilidadFA

Seguidamente, se muestra el código de dicho método. Está implementado en C# con el framework .NET, al igual que el resto de la aplicación. Este método es llamado al final de la operación de carga de resultados del histórico, en el método GetResultadosHistorico de la misma clase, ResultadoPruebaHistorico.

```
public static void CalcularProbabilidadFA(IEnumerable<ResultadoPruebaHistorico> resultados)
{
    using (DataClassesLogsDataContext dbLogs = new DataClassesLogsDataContext())
    {
        // Obtenemos el listado de los padres de las baterias
        var bateriasPadres = resultados.Select(x => x.IdBateria).Distinct().Select(
                    x => dbLogs.GetBateriaPadre(x)).Distinct().ToList();

        foreach (var bateriaPadre in bateriasPadres)
        {
            // Para cada bateria, obtenemos los descendientes de su padre
            var idBateriasHijas = dbLogs.ATUNObtenerBateriasHijas(bateriaPadre).Select(x => x.IDBateria);

            List<int?> idsBaterias = idBateriasHijas.ToList();
            idsBaterias.Add(bateriaPadre);

            // Escogemos los resultados de las baterias que son familia de la que estamos analizando
            var resultadosBateriaPadreYHijas = resultados.Where(x => idsBaterias.Contains(x.IdBateria)).ToArray();

            AnalizarResultadosBateria(resultadosBateriaPadreYHijas);
        }
    }

}
```

Ilustración 15: Código de CalcularProbabilidadFA

A continuación se muestra el diagrama de flujo y el código del método AnalizarResultadosBateria, la subrutina a la cual se llama en el método anterior. Esta subrutina primero ordena los resultados por IDPSistema, para tener las distintas ejecuciones de la misma prueba en secciones continuas dentro del array, después lo recorre primero analizando y luego asignando valores a cada sección.

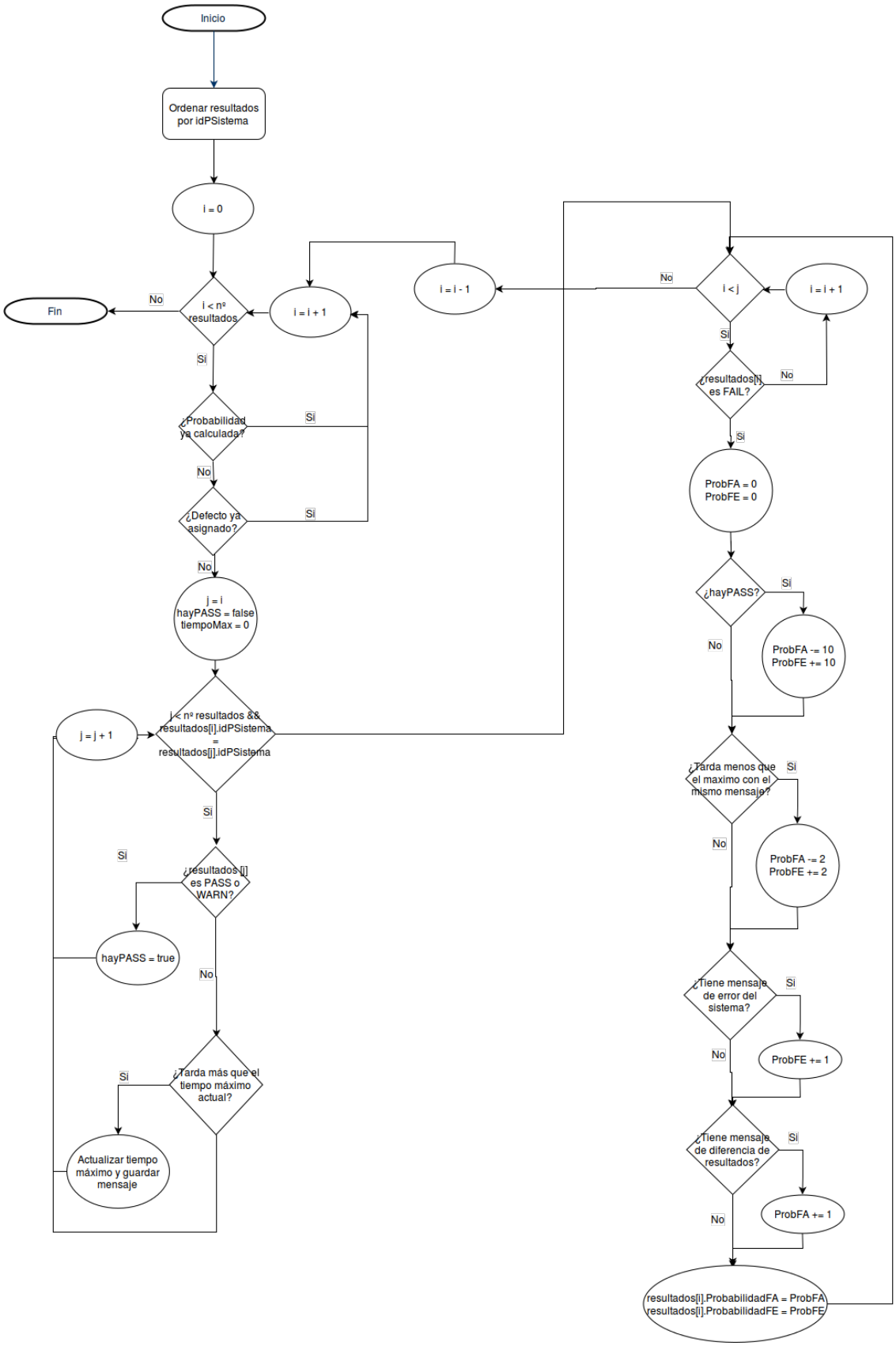

Ilustración 16: Diagrama de flujo del método AnalizarResultados

Como se observa en la ilustración 16, el algoritmo va recorriendo el conjunto de resultados y, si no tienen defecto asignado o la probabilidad ya calculada, pasa a analizarlos. Puesto que están ordenados por código de PS, tenemos contiguos las ejecuciones de las mismas pruebas. Estas secciones contiguas se recorren buscando alguna ejecución pasada o, en su defecto, el tiempo de ejecución más largo. Después se asignan las probabilidades adecuadas, muy alta probabilidadFE (+10) y muy baja probabilidadFA (-10) si hay un PASS, alta probabilidadFE (+2) y muy baja probabilidadFA (-2) si hay una ejecución con un mensaje diferente y mayor tiempo de ejecución, se aumenta la probabilidadFE (+1) si el mensaje es de error del sistema y se aumenta la probabilidaFA (+1) si el mensaje es de diferencia de resultados.

```
private static void AnalizarResultadosBateria(ResultadoPruebaHistorico[] resultadosBat)
{
    // Ordenamos por IDPSistema.
    resultadosBat = resultadosBat.OrderBy(o => o.IdPSistema).ToArray();

    for (int i = 0; i < resultadosBat.Length; i++)
    {
        if (!(resultadosBat[i].CodigoDefecto == null) || !(resultadosBat[i].ProbabilidadFA == null))
        {
            // si ya tiene asignado código defecto o ya se ha calculado la prob. no hacemos nada.
            // (saltamos a la siguiente iteracion).
            continue;
        }

        // Vamos a tratar ahora las múltiples ejecuciones de una prueba, que, como hemos ordenado
        // por ID de la prueba, estarán contiguas en la lista resultadosBat.

        bool hayPASS = false;
        long? tiempoMaximo = 0;
        String mensajeTiempoMaximo = "";
        int j;

        // Iteramos desde la primera posicion de la seccion hasta que el IdPSistema cambie.
        for (j = i; j < resultadosBat.Length && resultadosBat[j].IdPSistema == resultadosBat[i].IdPSistema; j++)
        {

            hayPASS = hayPASS || resultadosBat[j].GetResultado().Equals(ResultadoPruebaAutomatizada.PASS) ||
                                 resultadosBat[j].GetResultado().Equals(ResultadoPruebaAutomatizada.WARN);

            // Descartamos las pruebas no finalizadas,
            // o con resultado UNKNOWN, que pueden haberse quedado en bucle sin avanzar.
            if (!hayPASS && resultadosBat[j].GetResultado().Equals(ResultadoPruebaAutomatizada.FAIL) &&
                                 resultadosBat[j].TiempoTotal > tiempoMaximo)
            {
                // Tiempo máximo -> punto de ejecución más avanzado alcanzado.
                tiempoMaximo = resultadosBat[j].TiempoTotal;
                mensajeTiempoMaximo = resultadosBat[j].Mensaje;
            }
        }
```

Ilustración 17: Código del método AnalizarResultados (parte 1)

```
// Y volvemos a recorrerla (j se quedó como el primero de la sección siguiente)
for (; i < j; i++)
{
    if (!resultadosBat[i].GetResultado().Equals(ResultadoPruebaAutomatizada.FAIL))
        continue;
    int ProbFA = 0;
    int ProbFE = 0;
    if (hayPASS)
    {
        ProbFA -= 10;
        ProbFE += 10;
    }
    if (resultadosBat[i].Mensaje != null)
    {
        // Para el tiempo maximo, se considera que si una prueba tiene el mismo mensaje que
        // la que haya dado tiempo maximo, es el mismo fallo aunque tarde menos.
        if (!hayPASS && !resultadosBat[i].Mensaje.Equals(mensajeTiempoMaximo))
        {
            ProbFA -= 2;
            ProbFE += 2;
        }
        if (resultadosBat[i].Mensaje.Contains("ystem"))
        {
            // Mensaje de error del sistema
            ProbFE += 1;
        }
        if (resultadosBat[i].Mensaje.StartsWith("Mensaje"))
        {
            // Mensaje de diferencia de resultados
            ProbFA += 1;
        }
    }

    resultadosBat[i].ProbabilidadFA = ProbFA + "";
    resultadosBat[i].ProbabilidadFE = ProbFE + "";
}
// El último bucle deja a i apuntando al primer elemento de la siguiente sección
// hay que decrementarlo para que la guarda del bucle no haga que se salte uno.
i--;
```

Ilustración 18: Código del método AnalizarResultados (parte 2)

En las ilustraciones 17 y 18 se muestra el código de la extensión. Es la implementación en C# del algoritmo ya explicado, con las peculiaridades del lenguaje (p.e. el uso del método Equals para comparar objetos). Las líneas de comentario (que comienzan por //) son comentarios añadidos para ayudar a entender las instrucciones.

5. Resultados de uso de la extensión

Tras el desarrollo de la herramienta, inmediatamente se incorporó al proceso continuo de aseguramiento de la calidad (QA) para comprobar su efectividad. También se aprovechó el histórico de resultados para probar, conociendo de antemano los fallos de aplicación existentes, el porcentaje de acierto de la herramienta. En este capítulo mostraremos estos resultados y los analizaremos, primero lo realizados sobre el histórico, posteriormente los ejecutados sobre las nuevas versiones, es decir, con la herramienta ya incorporada al proceso, y en último lugar expondremos algunas conclusiones y evaluaremos si la extensión cumple los objetivos marcados.

5.1 Pruebas sobre el histórico de pruebas.

Como ya se ha comentado, el histórico de las pruebas ejecutadas sobre las distintas versiones del producto incluía los resultados de las pruebas automatizadas de los últimos años. Esto permitió visualizar de nuevo estas pruebas con los nuevos campos de probabilidad (ProbabilidadFA y ProbabilidadFE) y comprobar si los fallos que se habían detectado como fallos de aplicación o de entorno el sistema le asignaba una probabilidad acorde.

Para realizar estas pruebas se escogieron varias suites de gran tamaño (>3000 pruebas) de las últimas versiones del software antes del inicio del desarrollo de la prueba, entre marzo y junio del año 2016. Los resultados de las suites son muy variados entre sí, tanto en lo que a número de fallos de entorno y aplicación se refiere como en cuanto al tiempo que se dedicó a revisar la suite, ya que en ciertas ocasiones, cuando hay un número anormal de fallos por la misma causa, se espera a que éste se resuelva y se relanza la suite completa. De cualquier modo, estos datos siempre resultan interesantes puesto que recrean las condiciones normales de ejecución de la aplicación con la ventaja de saber de antemano qué fallos son de aplicación y cuales no. La siguiente tabla muestra los resultados de estas pruebas, sin tener en cuenta el valor absoluto sino el signo del resultado. Queda para trabajo futuro el calibrado de la herramienta, pues requeriría un estudio más exhaustivo de qué causas o heurísticas son más fiables.

Fecha suite	Nº pruebas	Nº falladas	FA	Prob. FA > 0	Aciertos en FA	Prob. FA = 0
10/06/16	3626	179	7	41	3	7
01/06/16	3628	415	47	67	15	324
07/05/16	3580	358	14	5	3	99
29/04/16	3815	628	51	76	8	299
07/04/16	3159	2132	3	38	2	1203
10/03/16	3592	1744	7	10	1	92

Fecha suite	Nº pruebas	Nº falladas	FE	Prob. FE > 0	Aciertos en FE	Prob. FE = 0
10/06/16	3626	179	172	129	122	8
01/06/16	3628	415	368	9	8	406
07/05/16	3580	358	344	238	236	103
29/04/16	3815	628	577	259	255	369
07/04/16	3159	2132	2129	926	925	1231
10/03/16	3592	1744	1737	1651	1645	93

Tabla 1: Resultados del testeo de la herramienta sobre el histórico de suites

Antes de analizar los datos, tengamos en cuenta algunas consideraciones, con el fin de interpretarlos correctamente:

- Las columnas “Aciertos en FE” y “Aciertos en FA” se corresponden con aquellas pruebas cuya probabilidad era mayor que cero y finalmente eran fallos de entorno o aplicación, respectivamente, es decir, el resultado de cruzar los datos de las dos columnas anteriores.
- Los fallos de aplicación son las pruebas que fallan debido a fallos de aplicación, recordemos que un mismo fallo de aplicación puede provocar múltiples fallos en las pruebas. Los fallos de entorno se han calculado automáticamente restando estas pruebas falladas al total de fallos. Aunque es posible que haya fallos de aplicación no descubiertos, ya que nunca se comprueban todas las pruebas, esta posibilidad es muy improbable, ya que se trata de versiones del software que ya están siendo usadas por los clientes y cualquier fallos sería reportado por ellos mismos.
- Para extraer estos datos se ha modificado el algoritmo para que incluya los fallos ya reportados, y no les aplique la heurística según la cual se disminuye la probabilidad de fallo de aplicación si la aplicación funciona en una ejecución posterior. La razón es muy simple: si una prueba con fallo reportado pasa a funcionar cuando

antes no lo hacía, se debe a que el fallo ha sido reparado, no a que se trataba de un fallo de entorno. Sin embargo, para un uso normal, el funcionamiento original de la aplicación, según el cual no se calcula la probabilidad si ya ha sido reportado un fallo de aplicación, es el correcto, puesto que sólo nos interesa calcular la probabilidad de aquellas pruebas de las que desconocemos la causa de su fallo.

Teniendo esto en cuenta, lo primero que nos llama la atención es la variabilidad de los datos. Estas diferencias se notan ya en los datos de números de fallos de pruebas y pruebas falladas por fallo de aplicación, como ya se expuso en el capítulo 3, por tanto, es lógico y coherente que los resultados de nuestra herramienta muestren también esta variabilidad. Las razones de esto son, principalmente, el hecho de que un mismo fallo de aplicación provoque un número arbitrario de errores en las pruebas, que, unido al hecho de que el número de fallos de aplicación detectados en cada suite es también muy variable, provoca resultados extremadamente dispares. Destacar que las suites con un gran número de fallos (>1000) suelen deberse a fallos en las partes vitales del producto a testear, que impiden la ejecución de un gran porcentaje de las pruebas.

Salvando estas dificultades para analizar los resultados, podemos decir que superan las expectativas. En primer lugar, observamos que la extensión ha arrojado resultados, es decir, ha devuelto una probabilidad distinta de cero en más del 90% de las pruebas en todas las suites excepto en una, que se puede considerar un caso aislado, ya que vemos que es una batería cuyo número de fallos supera con creces el habitual. La razón por la que no se ha descartado es porque, sin embargo, este tipo de resultados se dan con una frecuencia no despreciable y era interesante saber cómo respondía la extensión. Esto es un signo de que las heurísticas escogidas eran adecuadas, pues un número mayor de pruebas sin resultado indicarían que estas eran excesivamente específicas. Por tanto, podemos afirmar que esta primera versión es lo suficientemente general para lo que se requería.

Pasemos ahora al porcentaje de acierto de las pruebas con resultado. Para analizar estos resultados sencilla y correctamente, se adjunta la siguiente tabla, en la cual hemos calculado la probabilidad de encontrar un fallo de aplicación sobre el total de fallos (tal y como se haría sin aplicar la herramienta) y sobre los fallos que la herramienta ha señalado como candidatos a fallo de aplicación.

Nº pruebas	Nº falladas	FA	Prob. Acierto SIN herramienta	Prob. FA > 0	Aciertos	Prob. Acierto CON herramienta
3626	179	7	3,91%	41	3	7,32%
3628	415	47	11,33%	67	15	22,39%
3580	358	14	3,91%	5	3	60,00%
3815	628	51	8,12%	76	8	10,53%
3159	2132	3	0,14%	38	2	5,26%
3592	1744	7	0,40%	10	1	10,00%

Nº pruebas	Nº falladas	FE	Prob. Acierto SIN herramienta	Prob. FE > 0	Aciertos	Prob. Acierto CON herramienta
3626	179	172	96,09%	129	122	94,57%
3628	415	368	88,67%	9	8	88,89%
3580	358	344	96,09%	238	236	99,16%
3815	628	577	91,88%	259	255	98,46%
3159	2132	2129	99,86%	926	925	99,89%
3592	1744	1737	99,60%	1651	1645	99,64%

Tabla 2: Comparativa entre probabilidad de acierto evaluando un fallo con y sin herramienta

Como muestran los datos, la diferencia de usar o no la herramienta es notoria. En cuanto a los fallos de aplicación, se comprueba que la herramienta aumenta las probabilidades de encontrar fallos de aplicación, lo que provocará que estos se encuentren más rápidamente. La diferencia va desde una mejora del 20% (8,12% frente al 10,53% en la segunda batería) hasta la espectacular cifra del 1500% (3,91% frente al 60% en la tercera batería). Así podemos asegurar que la extensión garantiza encontrar fallos de aplicación en las pruebas a las que asigna una probabilidad positiva en un menor tiempo que si no se utiliza.

Por lo que respecta a los fallos de entorno, en todos los casos conseguimos porcentajes de aciertos cercanos al 100%, y en todos los casos excepto uno se mejoran los resultados con el uso de la herramienta, por lo que también se puede afirmar que se cumplen los objetivos en este apartado. Las sucesivas versiones de la extensión deberán intentar alcanzar ese 100% de acierto, al menos a partir de cierto valor de probabilidad. Esto permitiría aplicar políticas de resolución de fallos de entorno automáticas, tal y como ya se ha comentado. Por ejemplo, si se detecta que el 100% de los fallos cuyo valor de ProbabilidadFE es superior o igual a 10 resultan ser auténticos fallos de entorno, se puede programar una repetición de la prueba sin necesidad de intervención del usuario.

Estos resultados son alentadores pero pueden resultar engañosos, debido a la gran cantidad de fallos de entorno en comparación con los de aplicación.

Para estudiar la cantidad de aciertos en este último tipo de fallo sobre los totales, mostramos una última tabla más:

Nº pruebas	Nº falladas	FA	Aciertos	Fallos	% Acierto sobre FA totales
3626	179	7	3	4	42,86%
3628	415	47	15	32	31,91%
3580	358	14	3	11	21,43%
3815	628	51	8	43	15,69%
3159	2132	3	2	1	66,67%
3592	1744	7	1	6	14,29%

Tabla 3: Aciertos y fallos en estimación de fallos de aplicación

Estos datos muestran que, a pesar de la mejora en el tiempo dedicado a encontrar fallos gracias a la herramienta, una vez revisados aquellos candidatos a fallo de aplicación, debemos continuar revisando aquellos que no lo son, ya que la mayoría de los fallos no han sido detectados. Esto implica que el éxito de la extensión es sólo parcial, pues esta segunda parte del análisis no se realizaría en menos tiempo del actual. Está aquí una de las prioridades de cara a futuras mejoras, ya que esta cifra debería aumentar, al menos, al 50% en la mayoría de casos para ser considerada aceptable. Mientras esto no ocurra, esta mejora no afectará a un número suficiente de fallos de aplicación reales y, por tanto, no acelerará una parte suficiente del trabajo total.Hasta ahora nos hemos centrado en analizar la aplicación sin tener en cuenta el valor absoluto arrojado por la herramienta. La razón es que para realizar una calibración correcta de la misma habría que tener en cuenta la experiencia de uso de la aplicación, es decir, cómo interpreta un usuario los resultados y comprobar si es la que intenta transmitir la herramienta, además de que de momento el número de heurísticas aplicadas es lo suficientemente pequeña como para que haya diferencias notables y conocidas de antemano entre unas y otras, con lo cual se puede asignar un valor comparativo aproximado. Tampoco es una prioridad en esta primera versión el que los resultados tengan una calibración perfecta, pues es algo que es sencillo de corregir. Sin embargo, también se ha dedicado un pequeño tiempo a estudiar estos datos.

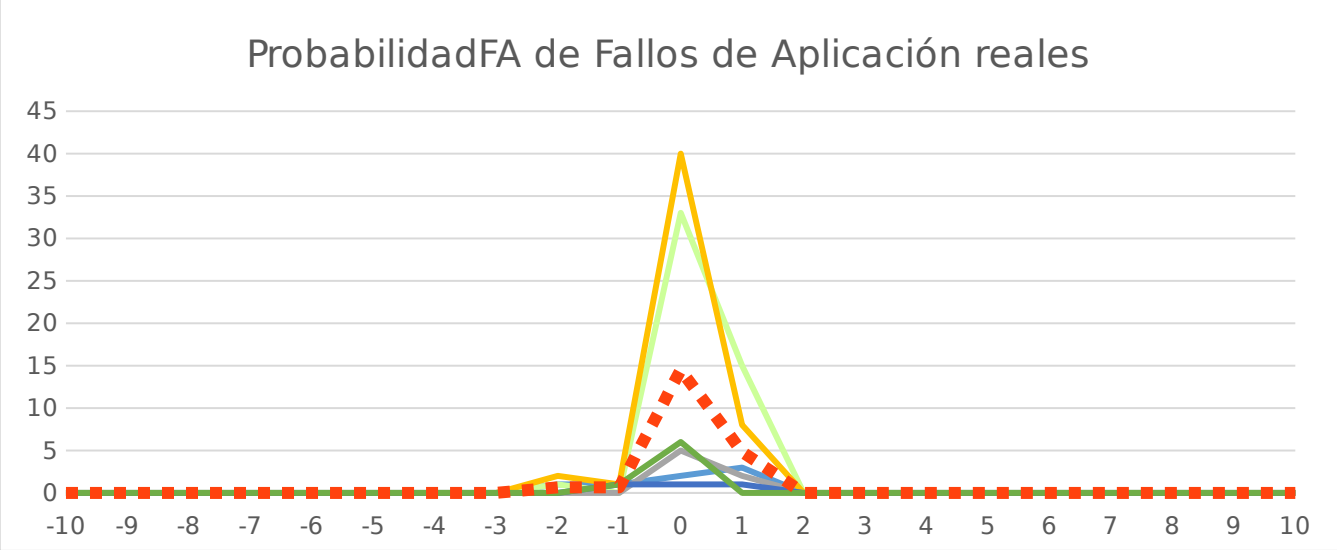

Ilustración 19: Resultados de la ProbabilidadFA calculada de los fallos de aplicación detectados

En esta gráfica se muestran los valores numéricos que la extensión asignó a cada una de las suites que hemos utilizado para el estudio y la media de estos valores, que se muestra como la línea roja discontinua. Como se puede apreciar, además de las ya mencionadas diferencias abismales entre unas suites y otras, los resultados se centran alrededor del cero, debido a que, tal y cómo se observó, la gran mayoría de los resultados obtenían este resultado ya que no estaban contenidos en ninguna heurística. Además de eso, se observa que hay un mayor peso en los valores positivos que en los negativos, aunque no se llega a ningún valor cuyo valor absoluto se diferencie en más de dos unidades del cero. En general, es un resultado bueno, puesto que si algunos fallos de aplicación tuvieran una prioridad muy baja, correríamos el riesgo de descartarlos incorrectamente.

Con estos matices, se puede considerar que los datos que arroja esta parte del estudio son favorables, teniendo en cuenta que se trata de una versión inicial y minimalista de la herramienta. Unos resultados prometedores de cara a una inversión en el tema y una evolución de los métodos utilizados en ella.

5.2 Resultados sobre las versiones a testear del producto

Como hemos introducido, la extensión se incorporó al proceso de testeo tan pronto estuvo lista para probar su efectividad. La fecha de liberación de la versión de ATUN que contiene la extensión puede considerarse el 25 de julio de 2016, si bien llevaba unas semanas probándose de manera extraoficial. Se muestran aquí algunos resultados del testeo, tanto manual

como automatizado, sobre las versiones del producto que se desarrollaron entre esa fecha y la de finalización del presente trabajo, que se completa en septiembre de 2016. Antes de mostrar los resultados, nos gustaría aclarar que durante el mes de agosto se redujo la plantilla del equipo de testeo, debido a las vacaciones.

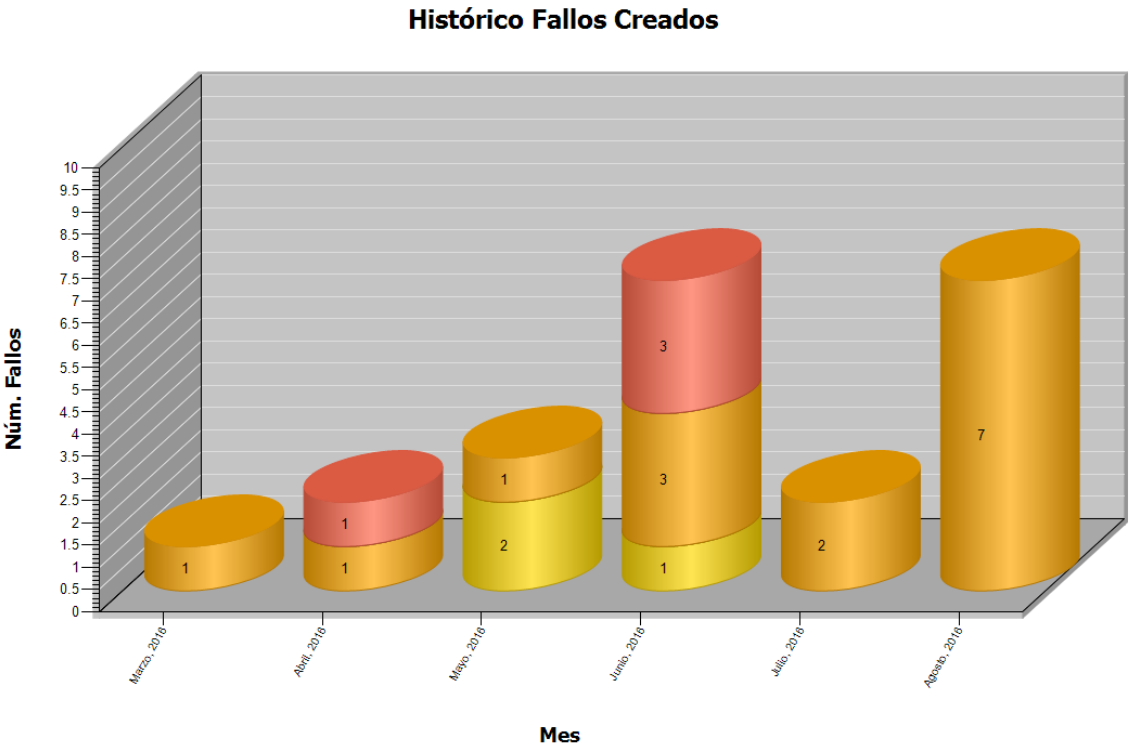

Ilustración 20: Fallos detectados en testeo automatizado. Febrero-Agosto 2016

.

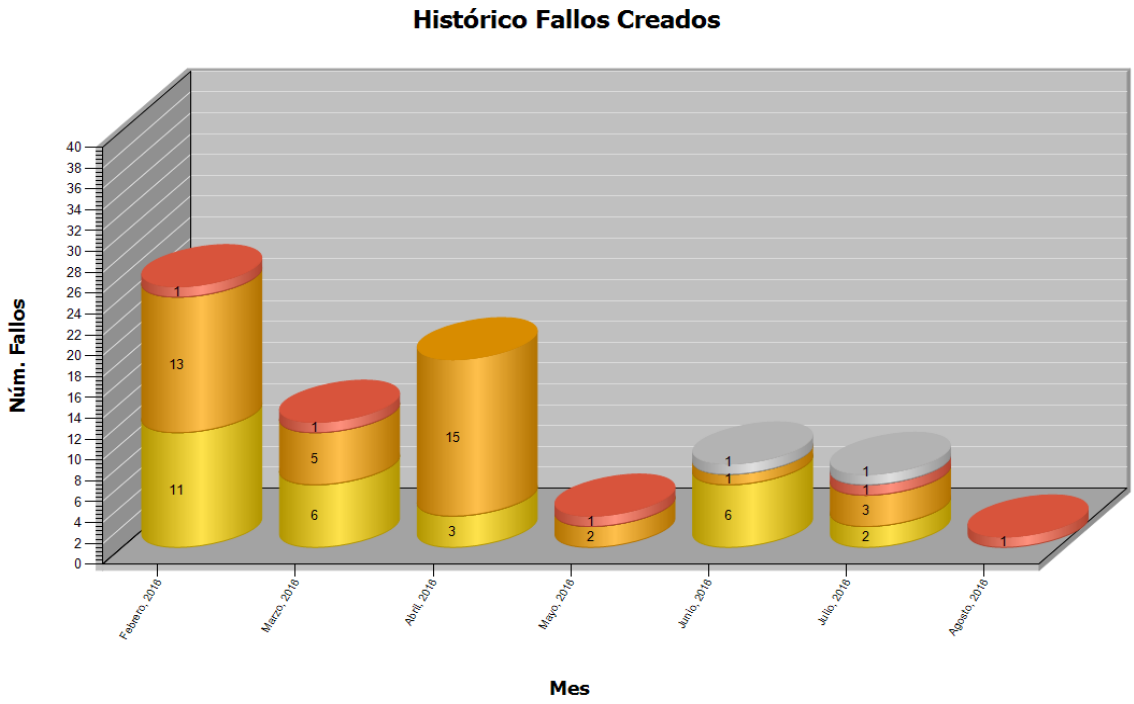

Ilustración 21: Fallos detectados en testeo manual. Marzo-Agosto 2016

Como se puede comprobar, agosto fue, junto con el de junio, el mes en el que más fallos se detectaron en testeo automatizado, a pesar de que únicamente se contaba con uno de los miembros del equipo de testeo automatizado. En el otro extremo, fue el mes en el que menos fallos se detectaron en testeo manual. En estos resultados tan dispares indudablemente influyeron factores externos, tales como el hecho ya comentado de que se trata de un mes de vacaciones para muchos trabajadores. Sin embargo, tampoco podemos negar que los resultados no podrían ser más alentadores. De confirmarse esta tendencia, se mostraría que el testeo automatizado puede sustituir en parte al manual.

Al margen de estos datos y análisis, nos gustaría anotar algunos puntos que el equipo de testeo comentó sobre la herramienta, de manera subjetiva o en forma de objetivos no contemplados en la idea original de la herramienta como beneficiosos, pero que al equipo le parecieron interesante comentar ya que les facilitaba sumamente la tarea, para completar la visión del cambio que ha supuesto la herramienta en la tarea de revisión de resultados.

- Puesto que a los fallos ya categorizados (ya sea como fallo de aplicación o falso fallo) no se les asigna ningún valor de prioridad, **separa los fallos ya revisados de los que no lo están** de una manera tan sencilla como usar el filtro de prioridadFA o FE, tal y como muestra la siguiente imagen, en la cual ya hay 8 resultados que no es necesario revisar pues ya tienen fallo asociado. En las versiones anteriores, era necesario comprobar que los campos Motivo fallo y código defecto estuvieran vacíos.

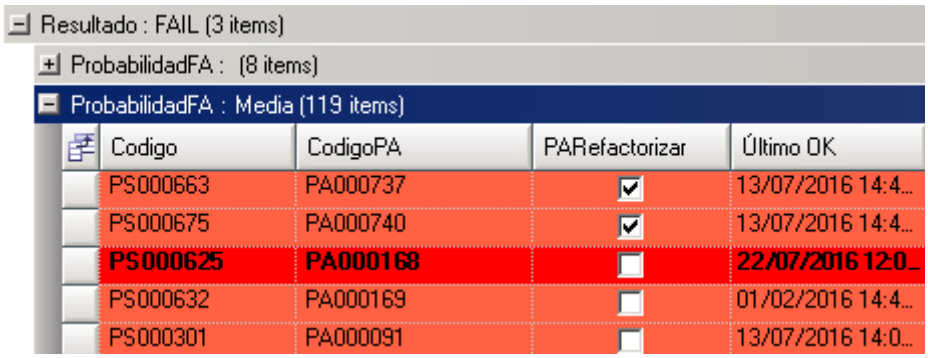

Ilustración 22: Captura de ATUN agrupando los fallos ya clasificados

- **Permite visualizar el estado de toda una familia de suites únicamente revisando la raíz de todas ellas.** Esto es debido a que, al revisar todos los hijos, no es necesario comprobar si una prueba que estaba en la suite padre se había vuelto a ejecutar y ya funcionaba. Esta tarea anteriormente dependía de que los testers clasificaran correctamente la prueba como resuelta (asignándoles código defecto o bien un motivo de fallo de falso fail, en su caso), y si esto no se hacía, provocaba retrabajo. De hecho, una de las mejoras más inmediatas propuestas para la herramienta es que, en lugar de que se aplique la heurística y se le añada probabilidad de fallo de entorno, directamente se clasifique la prueba como falso fail, si no tiene asignado código defecto, por supuesto. La razón por la que esto no se ha hecho aún es que no se deseaba que la aplicación modificara la base de datos hasta que se hubiera probado su estabilidad.

6. Conclusiones y trabajo futuro

Tras analizar los resultados y estudiarlos detenidamente, en este capítulo haremos un repaso de todo lo expuesto anteriormente a lo largo del presente documento y formularemos las conclusiones pertinentes, con el fin de extraer una visión completa y global de todo el trabajo realizado. También comprobaremos si se han cumplidos los objetivos marcados al inicio del documento. Finalmente daremos un enfoque hacia el posible trabajo futuro que se podría realizar teniendo en cuenta las conclusiones que saquemos.

Como vimos en el capítulo 2. las herramientas comerciales de más éxito en el mercado no ofrecían ninguna utilidad similar a la planteada por este trabajo, ni siquiera parecían orientados a resultar efectivos frente a la tarea de revisión masiva de resultados de pruebas automatizadas. La herramienta más avanzada parecía el lanzador ATUN desarrollado en la propia empresa, que, aunque no ofrecía la utilidad que estábamos planteando, sí que permitía resumir la información de los resultados y clasificarlos rápidamente. Es por ello que la herramienta a desarrollar se planteó como una extensión de este mismo lanzador.

En el capítulo 3. se mostraron y estudiaron los resultados que había ido devolviendo el proceso de testeo automatizado en el periodo inmediatamente anterior al inicio del desarrollo de la herramienta. Estos resultados mostraban una muy baja tasa de detección de fallos de aplicación, especialmente en comparación con el trabajo realizado en testeo manual, provocado, entre otras causas, por el variable pero siempre importante número de falsos fallos que arrojaban las sucesivas ejecuciones de las baterías de pruebas, lo cual reforzaba la idea de que era una tarea con un gran potencial de mejora si se lograba elaborar una herramienta efectiva.

En el capítulo 4. se abordó el diseño de la herramienta, mostrando tres alternativas básicas de desarrollo para la detección de los fallos de aplicación: desarrollo orientado a aplicación de heurísticas, desarrollo orientado al uso de técnicas de reconocimiento de formas, y un desarrollo mixto combinando estas dos técnicas. La conclusión que se extrajo fue que, aparentemente, la mejor opción consistía en utilizar el diseño basado en heurísticas para la primera versión de la herramienta, para, una vez probada su eficacia, optar por introducir el reconocimiento de formas paulatinamente (es decir, mediante el diseño mixto) para que, una vez creados los datos con los que entrenar el sistema y que se demuestre su fiabilidad, pasar a un diseño en el que el reconocimiento de formas tome un papel dominante sobre las heurísticas, pues este sistema es mucho más

adaptativo que el otro. También se mostró el diseño del algoritmo y la implementación final de la herramienta, además de los cambios que supone en el lanzador ATUN a nivel de interfaz gráfica, que se limitaron a la adición de dos columnas con las probabilidades de que un fallo determinado se deba a un fallo en la aplicación o a un fallo en el entorno.

En el quinto capítulo se muestran las pruebas realizadas de la herramienta. Los datos arrojados por los estudios realizados sobre las baterías ya estudiadas antes de la existencia de la extensión son muy positivos, y demuestra que la herramienta, a pesar de haber utilizado el diseño más minimalista, es muy efectiva, lo que lleva implícito el gran potencial que tienen estas herramientas para agilizar la tarea que no ocupa. Además, la introducción del mismo en el proceso ha tenido una excelente acogida por parte del equipo de testeo, que incluso encontró más ventajas en la herramienta de las originalmente buscadas.

En resumen, este trabajo ha permitido confirmar la hipótesis de que había un gran potencial en la elaboración de herramientas de apoyo a la revisión de suites, a pesar de que era un tema muy poco tratado y poco cubierto por las herramientas de testing comerciales actuales. Sin embargo, y con el fin de evaluar el trabajo realizado de la forma más imparcial posible, repasaremos los objetivos marcados uno a uno y comprobaremos si se han alcanzado:

- La herramienta debía permitir **priorizar la revisión de los fallos de aplicación con respecto a los falsos fallos.** Puesto que, tal y como muestran los datos del capítulo 5, la aplicación resalta correctamente los fallos de aplicación, creando un subconjunto de fallos con el valor probabilidadFA positiva en el cual es más probable encontrar fallos de aplicación que en el conjunto global, lo que reduce el tiempo de encontrarlos en comparación con no usar la herramienta.
- **Identificar y categorizar los falsos fallos.** Aunque este objetivo puede parecer que se ha dejado un tanto de lado a lo largo del documento, no dudamos de que la herramienta puede permitir agilizar esta tarea también, haciendo un estudio de los resultados del valor probabilidadFE y los mensajes asociados. Para cubrir completamente este objetivo creemos que será necesario dar un paso más y añadir al algoritmo utilizado técnicas de reconocimiento de formas, y puesto que esto no era viable en el contexto realizado ni alcanzable en una primera versión del proyecto, consideramos que, aunque no se haya completado, se ha avanzado lo suficiente en este apartado.
- **Dar una base para la gestión automática de los falsos fallos.** Como hemos visto, los fallos que la herramienta indicaba como candidatos a falsos fallos eran en más del 90% de los casos falsos

fallos. Si se consigue aumentar esa cifra al 100%, al menos para determinados casos o a partir de determinados valores de probabilidad, se podrá empezar a programar técnicas de resolución automáticas, como repetición de la prueba y borrado del resultado incorrecto. Esto además, permitiría aumentar la calidad de los datos obtenidos. Así pues, consideramos que la versión actual está muy cerca de permitir esta gestión automática y que, por tanto, hemos cubierto satisfactoriamente este objetivo.

En conjunto hemos cubierto dos de los tres objetivos y el que no hemos completado está debidamente justificado, por tanto, podemos considerar el trabajo como exitoso. Sin duda ayudará a hacer la tarea de revisión de suites más liviana y quizá incluso llegue al punto de ser completamente automática, para así garantizar la efectividad de todo el conjunto del proceso de testeo automatizado y aseguramiento de la calidad.

Este proyecto, como hemos reiterado a lo largo del documento, pretendía comprobar si era posible agilizar la tarea de revisión de suites mediante herramientas de apoyo. Una vez demostrado esto, queda para trabajo futuro desarrollar y potenciar estas herramientas. Algunas referencias acerca de los posibles frentes por donde atacar esta tarea son:

- **Pasar del diseño por heurísticas al mixto o por reconocimiento de formas.** Como ya se comentó en el capítulo 4, la opción de diseño escogida era la ideal para una primera versión, por su simplicidad, pero, una vez comprobada la efectividad de la herramienta, la evolución hacia el reconocimiento se convierte en un paso necesario para garantizar la mejora de la herramienta. Nuestra recomendación es introducir el reconocimiento de formas de forma gradual para ir generando una base de datos de entrenamiento suficiente, tal y como se comentó en el capítulo 4.
- **Introducir la posibilidad de ejecutar código automáticamente cuando los datos obtenidos superen cierto umbral.** Cuando la probabilidadFA o probabilidadFE tengan un valor determinado, puede ser interesante permitir que se ejecuten ciertas instrucciones. Por ejemplo, con una probabilidadFE suficiente podemos querer ejecutar de nuevo la prueba y descartar ese resultado, que no nos interesa pues es un fallo de entorno. Otro ejemplo sería cuando tenemos una probabilidadFA suficiente, el sistema podría emitir algún tipo de alerta para avisar al equipo de testeo de que se ha detectado un potencial defecto en la aplicación. Para introducir esta mejora sería necesario realizar la mejora expuesta en el punto siguiente:
- **Hacer que el algoritmo de cálculo de probabilidades se ejecute al finalizar cada prueba y guardarlo en base de datos.** Hasta ahora hemos mostrado la parte servidor del lanzador ATUN, pero

también hay una parte cliente que se encarga de ejecutar las pruebas en las máquinas virtuales, como se comentó en el capítulo 2. Esta parte cliente debe realizar el cálculo del algoritmo para poder tomar decisiones en consecuencia, y para poder hacer los datos persistentes guardándolos en la base de datos. No es viable hacer los datos persistentes hasta que se realice esto, pues si el cálculo se realiza como hasta ahora deberíamos hacer primero una consulta sobre los datos de cada prueba para comprobar si el dato está calculado y, si no lo está, calcularlo y guardarlo, lo cual no es eficiente.

Como comentario final, nos gustaría expresar la satisfacción que nos produce el haber podido trabajar en un área que podíamos considerar prácticamente inexplorada, a juzgar por las escasas referencias encontradas al respecto, y haber abierto un mundo de posibilidades en la tarea de revisión de suites. No dudamos que será uno de los pilares para convertir la ejecución de pruebas de regresión automatizadas en un proceso muy rentable para las empresas y hacer real la revolución en el mundo del testeo y aseguramiento de la calidad del software que promete ser.

7. Referencias

[1] HP, Capgemini, Sogeti. World Quality report 2015-16

[2] McCallum, Andrew Kachites. "MALLET: A Machine Learning for Language Toolkit." http://mallet.cs.umass.edu. 2002.

[3] Wikipedia. Principio de máximo de entropía. https://es.wikipedia.org/wiki/Principio_de_m%C3%A1ximo_de_entrop%C3%Ada. 2016

[4] IBM. “Rational Functional Tester Demo”. http://download.boulder.ibm.com/ibmdl/pub/demos/on_demand/Streamed/IBM_Demo_Rational_Functional_Tester8.1-Oct09/RFT8.1_Demo.html#IBM_Recorded_Demonstration. 2009

[5] IBM. “Rational Quality Manager Demo”. http://download.boulder.ibm.com/ibmdl/pub/demos/on_demand/Download/IBM_Demo_Rational_Quality_Manager_Overview-1-Oct08.swf?S=DL

[6] Microsoft. “How to: View Test Plan Results in Microsoft Test Manager” https://msdn.microsoft.com/en-us/library/hh553099(v=vs.110).aspx 2016

[7] Jorge Hernan Abad Londoño. “Tipos de pruebas de Software” http://ing-sw.blogspot.com.es/2005_04_01_archive.html 2005