Especificación e implementación de casos de prueba

Javier Gutiérrez
javierj@us.es

1. Introducción 2
2. Especificación de los casos de prueba 3
   2.1. Planteamiento del problema 3
   2.2. Agedis 3
   2.3. UML 2.0 Testing Profile 4
   2.4. Test Specification Language 5
   2.5. TTCN-3 6
3. Implementación de los casos de prueba de la aplicación de escritorio 7
   3.1. Planteamiento 7
   3.2. Implementación de la prueba con Abbot 8
   3.3. Implementación de la prueba con JFCUnit 10
   3.4. Implementación de la prueba con Marathon 12
   3.5. Conclusiones 13
4. Implementación de los casos de prueba de la aplicación web 14
   4.1. Planteamiento 14
   4.2. Implementación de la prueba con JWebUnit 15
   4.3. Implementación de la prueba con Canoo WebTest 17
   4.4. Implementación de la prueba con Avignon 18
   4.5. Conclusiones 19

1. Introducción

En la sección 2 se describen algunas de las técnicas de especificación de pruebas más difundidas y se valora su utilidad en nuestro proceso de generación de pruebas. En la sección 3 se prueba un caso de uso de la aplicación de bloc de notas y se estudian distintas herramientas e implementaciones de dicho caso de uso. En la sección 4 se realiza lo mismo pero para la aplicación web de gestión de enlaces.

2. Especificación de los casos de prueba

2.1. Planteamiento del problema

Es necesario contar con una herramienta que permita la representación de los casos de prueba, es decir, la representación de las interacciones entre actores y el sistema, los valores de prueba implicados y los resultados esperados. Para ser útil, esta herramienta debe tener las siguientes características:

- Debe ser independiente de la plataforma y tecnología del sistema a prueba.
- Debe abstraernos de la implementación del sistema a prueba.
- Debe permitir interactuar a los actores externos mediante ratón y teclado a través de varias pantallas distintas.
- Debe ser manipulable programáticamente.
- Debe permitir expresar verificaciones o aserciones.
- Debe ser de libre acceso.
- Debe ser modular y permitir construir un caso de prueba a partir de casos de prueba y módulos ya existentes.
- Debe permitir generar código de prueba automáticamente.

En los siguientes puntos describimos brevemente algunas de las herramientas y propuestas estudiadas.

2.2. Agedis

Agedis ha sido un proyecto financiado por la comunidad europea con la participación de empresas privadas. Su objetivo ha sido elaborar un conjunto de documentos y herramientas para permitir las pruebas basadas en modelos. Uno de los resultados de Agedis ha sido una especificación para definir casos de prueba.
Sin embargo, esta especificación no cumple las características citadas arriba y no es de aplicación en este trabajo. A continuación se describe esta especificación y se expone los motivos por los que no nos es útil.

Los conjuntos de prueba de Agedir (AGEDIS Test Suite) están compuestos de descripciones abstractas (Abstract Test Suite) y trazas (Test Suite Traces). El primer elemento describe las clases, objetos y escenarios lógicos (casos de prueba), así como el comportamiento esperado (como parte de los escenarios lógicos). El segundo elemento describe una ejecución del primero. Un conjunto de pruebas contiene una única descripción abstracta y tantas trazas como se desee.

A continuación se muestra un ejemplo tomado de la documentación de Agedis. En la descripción abstracta se observa que se está probando una clase llamada Test con un método control() y otro observe() que devuelve un valor booleano. Se definen 2 casos de prueba iguales.

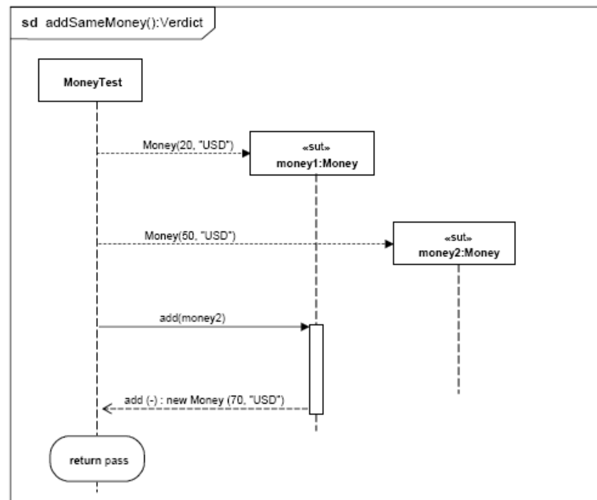
```
<testSuite xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../schema/testSuite.xsd">
  <abstractTestSuite generator="manual">
    <model>
      <class name="Test">
        <members>
          <member signature="control()"/>
```

```
          <member signature="observe():bool"/>
        </members>
      </class>
      <object name="test" class="test"/>
    </model>

    <testCase id="TC1">
      <step id="S1">
        <interaction signature="control()" object="test" type="call_return"/>
      </step>
      <step id="S2">
        <interaction signature="observe():bool" object="test" type="call_return">
          <value>true</value>
        </interaction>
      </step>
    </testCase>

    <testCase id="TC2">
      <step id="S1">
        <interaction signature="control()" object="test" type="call_return"/>
      </step>
      <step id="S2">
        <interaction signature="observe():bool" object="test" type="call_return">
          <value>true</value>
        </interaction>
      </step>
    </testCase>

  </abstractTestSuite>
</testSuite>
```

Aunque la especificación de pruebas de Agedis es independiente de la plataforma de implementación del sistema y fácilmente manipulable mediante herramientas software, no permite expresar interacciones entre actores externos y el sistema (por ejemplo, “pulso un botón”) ni nos abstrae de la implementación del sistema a prueba.

2.3. UML 2.0 Testing Profile

El objetivo de este perfil es capturar toda la información necesaria para pruebas de caja negra. Este perfil está divido en cuatro paquetes principales: comportamiento de la prueba (test behaviour), el cual expresa las observaciones y actividades a realizar durante la prueba, arquitectura de la prueba (test architecture) que contiene todos los elementos involucrados en la prueba y sus relaciones, datos de prueba (test data) que contiene las estructuras y significado de los valores utilizados en la prueba y tiempo de prueba (test time) que no es relevante en este caso. Este perfil puede utilizarse a cualquier nivel en el que se empleen pruebas de caja negra, desde pruebas unitarias hasta pruebas de aceptación. Sin embargo nos parece muy contradictorio hablar de pruebas de caja negra y pruebas unitarias y de integración. De hecho, el ejemplo incluido en la especificación oficial se centra principalmente en pruebas unitarias A continuación se muestra un ejemplo de diagrama de comportamiento de una prueba tomada de la documentación oficial y que sigue la forma de trabajar de herramientas tipo xUnit.



En este perfil no se define el modelo del sistema a prueba, sino que se importa (se supone ya hecho) y se enlaza con el modelo de prueba a través de sus interfaces externas. En todo momento el sistema a prueba se trata como una caja negra y no se relevan detalles de su implementación.

En una primera valoración, este perfil presenta varios problemas. Es difícil expresar la interacción de un usuario humano con una interfaz gráfica mediante diagramas de UML. Existe poca documentación que describa como utilizarlo, y menos aún a nivel de pruebas del sistema expuesto en este trabajo. Es difícil expresar comprobaciones concretas y comprobaciones que involucren patrones y expresiones regulares. La mejor opción es utilizar un lenguaje complementario como OCL. No vemos claro que, a partir de este perfil, se pueda obtener directamente código de prueba (al igual que no se puede obtener directamente el código de un sistema a partir de sus modelos en UML). Por esto necesitamos herramientas y lenguajes adicionales.

Por estos motivos, este perfil no es la solución definitiva. Sin embargo consideramos que este perfil es interesante y creemos que hay que adoptarlo siempre que sea posible.

2.4. Test Specification Language

Test Specification Language (no confundir con Task Specification Language), es un lenguaje ideado como soporte del método de categoría partición. Este lenguaje permite indicar las categorías, particiones y restricciones entre ellas. Mediante una herramienta es posible calcular todas las posibles combinaciones de valores que satisfacen las restricciones y, en algunas circunstancias, generar pruebas ejecutables.

Este lenguaje se utiliza en algunas herramientas propietarias de reconocido prestigio como WinRunner. También hemos encontrado herramientas libres que trabajan con este lenguaje, aunque poco elaboradas.

No vemos factible expresar las interacciones entre el usuario y el sistema a través de una interfaz con este lenguaje. Además, la documentación existente es escasa y no hemos encontrado ninguna especificación formal de este lenguaje. En los documentos que hemos consultado, no hemos encontrado nada que nos indique que TSL permite indicar verificaciones o aserciones sobre el sistema a prueba.

Algunas extensiones, como la herramienta WinRunner mencionada y otras herramientas de soporte como EMOS (de libre descarga), sí solucionan todas estas carencias, pero puesto que no es de libre acceso no la tendremos en cuenta.

2.5. TTCN-3

TTCN-3 es un lenguaje orientado a la especificación e implementación de casos de prueba desarrollado por el European Telecommunications Standards Institute (ETSI). TTCN-3 se puede dividir en dos grandes bloques, en lenguaje propiamente dicho, y la especificación de las pruebas. Actualmente existen dos alternativas para la especificación: tabular y diagramas gráficos, aunque TTCN-3 permite añadir cualquier otra especificación como diagramas UML o XML.

TTCN-3 es un lenguaje independiente del sistema a prueba. Las áreas típicas de aplicación son pruebas de protocolos, pruebas de componentes de bajo nivel (sistemas comunicaciones, sistemas de automóviles, sistemas aeroespaciales, etc.) y prueba de plataformas CORBA.

Hemos encontrado algunos trabajos que mencionan la aplicación de TTCN-3 en sistemas web y de información. Sin embargo no hemos visto aún de una manera clara como utilizar este lenguaje para especificar la interacción entre un actor humano y una interfaz gráfica. Tampoco hemos investigado la existencia de herramientas de libre acceso basadas en este lenguaje. A medida que profundicemos en el estudio de las posibilidades de pruebas con GUI y descubramos si se adapta o no, buscaremos herramientas de soporte.

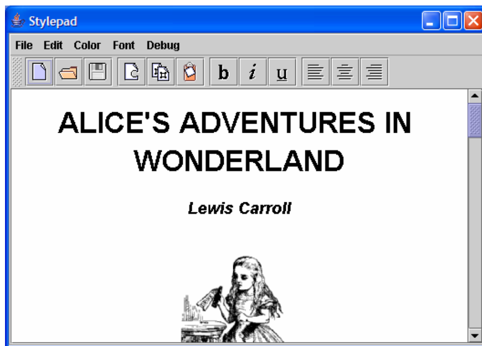
TTCN-3 no es nuestra solución definitiva porque no vemos factible escribir directamente las pruebas en este lenguaje. Necesitaríamos, primero, una especificación de las pruebas que pudiéramos traducir a TTCN-3. Actualmente existen varios trabajos que proponen trabajar conjuntamente con UML 2.0 TP para el modelado de pruebas y con TTCN-3 para su implementación. Si encontramos la manera de utilizar TTCN-3 para probar sistemas de información esta puede ser una buena solución debido a que ambas son tecnologías estándares (consideramos UML como estándar) y ampliamente difundidas.

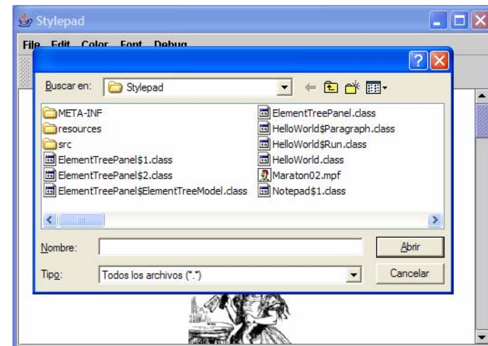
3. Implementación de los casos de prueba de la aplicación de escritorio

3.1. Planteamiento

Utilizamos una aplicación de bloc de notas con interfaz gráfica escrita en Java. Esta aplicación se distribuye como ejemplo en el kit de desarrollo estándar de Java.

Sin embargo, esta aplicación nos ha dado muchos problemas a la hora de utilizar las herramientas de prueba. Por eso hemos utilizado otra de las aplicaciones de ejemplo del mismo kit (Stylepad), la cual implementa un sencillo procesador de textos. La principal diferencia entre ambas aplicaciones es que el bloc de notas trabaja con texto sin formato y el Stylepad permite texto con formato. Sin embargo el caso de uso utilizado para la generación de pruebas es el mismo en ambas. A continuación se muestran dos capturas de la aplicación Stylepad.





A continuación se incluye el caso de prueba. Sólo se ha incluido un valor de prueba. Este valor se ha tenido que cambiar ya que Stylepad trabaja con documentos binarios y no con los documentos de texto plano que utiliza el bloq de notas.

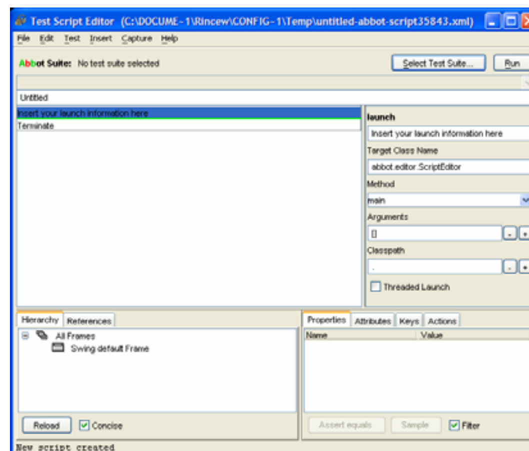
Nombre	Cargar un documento con éxito.
Objetivo	Este caso de prueba verifica el comportamiento del caso de uso, en su escenario principal.
Acciones	01 -> 02(nombreFichero) -> 03(contenidoFichero) -> 04(error) 01 La prueba selecciona la opción de cargar un documento 02 El sistema pregunta el archivo a abrir. 03 La prueba selecciona el documento de prueba. 04 El sistema muestra el documento.
Valores de prueba	*nombreFichero* = “prueba.stylepad” *contenidoFichero* = “Archivo de prueba” *error* = Sin Errores
Resultados observables	01 -> 02 [V02]-> 03 -> 04 [V04] V02: El sistema muestra el diálogo para seleccionar un archivo. V04: El sistema muestra el contenido del documento abierto.

Notas	El fichero de prueba estará en la misma carpeta que la aplicación. El texto del fichero de prueba no tendrá ningún formato.

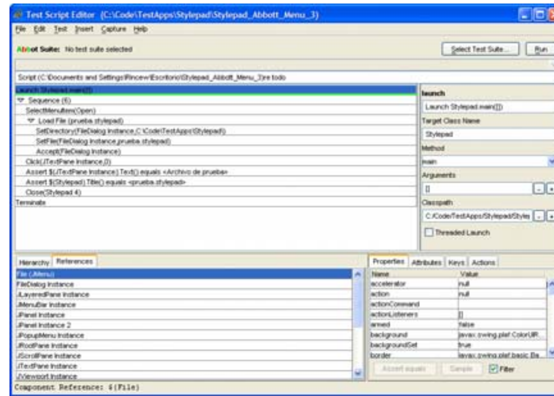
Para implementar la prueba del punto 3.1 se han buscado herramientas de libre descarga que permitieran indicar una secuencia de operaciones sobre una interfaz gráfica escrita en Java. Además, se han seleccionado aquellas herramientas con una documentación lo suficientemente completa para poder utilizarlas. Las herramientas seleccionadas son: Abbot, JFCUnit y Marathon. A continuación se describe la implementación de la prueba con cada una de ellas.

3.2. Implementación de la prueba con Abbot

Abbot es una herramienta que sirve tanto para probar componentes de manera aislada como para grabar y reproducir una secuencia de acciones. La herramienta se distribuye junto con un editor (llamado Costello) muy completo que facilita la tarea de grabar secuencias, construir casos de prueba, y reproducirlo. Fue posible ejecutar la herramienta sin tener que realizar ninguna instalación ni configuración. A continuación se muestra una captura del editor.



Mediante este editor, configuramos la aplicación a ejecutar y seleccionamos la opción de capturar secuencia. El editor ejecutó la aplicación y capturó todas las pulsaciones de ratón que realizamos, con lo que ya teníamos grabada secuencia de acciones de prueba. Esta secuencia se muestra en la siguiente captura. El editor, además, registró todos los componentes implicados en la secuencia, como se observa en la parte inferior de la siguiente captura.



Para la primera verificación (comprobar que el cuadro de diálogo aparece), no fue necesario añadir nada. Si el cuadro de diálogo no apareciera, Abbot indicaría un error ya que el resultado de la aplicación no coincidiría con las acciones y componentes que tiene grabados.

Para la segunda verificación (comprobar que la aplicación muestra el contenido del archivo), buscamos el componente encargado de mostrar el texto, buscamos la propiedad que contenía el texto, y mediante el botón "Assert equals" añadimos una comprobación. Además, nos dimos cuenta que, cuando se carga un archivo, el título de la ventana de la aplicación cambia con el nombre del archivo, por lo que, de la misma manera, añadimos una verificación adicional. Ambas verificaciones se pueden ver en la captura anterior.

Abbot almacena los casos de prueba en XML. El código XML generado para la prueba descrita se muestra a continuación.

```
<?xml version="1.0" encoding="UTF-8"?>
<AWTTestScript desc="Script (C:\Documents and Settings\Rincew\Escritorio\Stylepad_Abbott_Menu_3)re todo">
  <component class="javax.swing.JMenu" id="File" index="0" parent="JMenuBar Instance" text="File" window="Stylepad 6" />
  <component class="java.awt.FileDialog" id="FileDialog Instance" parent="Stylepad 4" title="" />
  <component class="javax.swing.JLayeredPane" id="JLayeredPane Instance" index="1" parent="JRootPane Instance"
window="Stylepad 2" />
  <component class="javax.swing.JMenuBar" id="JMenuBar Instance" index="0" parent="Stylepad Instance" window="Stylepad
5" />
  <component class="javax.swing.JPanel" id="JPanel Instance" index="1" parent="JLayeredPane Instance" window="Stylepad 3"
/>
  <component class="javax.swing.JPanel" id="JPanel Instance 2" index="1" parent="Stylepad Instance" window="Stylepad 4" />
  <component class="javax.swing.JPopupMenu" id="JPopupMenu Instance" index="0" invoker="File" />
  <component class="javax.swing.JRootPane" id="JRootPane Instance" index="0" parent="Stylepad" />
  <component class="javax.swing.JScrollPane" id="JScrollPane Instance" index="1" parent="JPanel Instance 2"
window="Stylepad 4" />
  <component class="javax.swing.JTextPane" id="JTextPane Instance" index="0" parent="JViewport Instance" window="Stylepad
4" />
  <component class="javax.swing.JViewport" id="JViewport Instance" index="0" parent="JScrollPane Instance"
window="Stylepad 4" />
  <component class="javax.swing.JMenuItem" icon="open.gif" id="Open" index="1" parent="JPopupMenu Instance" text="Open"
window="Stylepad 7" />
  <component class="javax.swing.JFrame" id="Stylepad" root="true" title="Stylepad" />
  <component class="javax.swing.JFrame" id="Stylepad 2" root="true" title="Stylepad" />
  <component class="javax.swing.JFrame" id="Stylepad 3" root="true" title="Stylepad" />
  <component class="javax.swing.JFrame" id="Stylepad 4" root="true" title="Stylepad" />
  <component class="javax.swing.JFrame" id="Stylepad 5" root="true" title="Stylepad" />
  <component class="javax.swing.JFrame" id="Stylepad 6" root="true" title="Stylepad" />
  <component class="javax.swing.JFrame" id="Stylepad 7" root="true" title="Stylepad" />
  <component class="Stylepad" id="Stylepad Instance" index="0" parent="JPanel Instance" window="Stylepad 4" />
  <launch args="[]" class="Stylepad" classpath="C:/Code/TestApps/Stylepad/Stylepad.jar" method="main" />
  <sequence>
    <action args="Open" method="actionSelectMenuItem" />
    <sequence desc="Load File (prueba.stylepad)">
      <action args="FileDialog Instance,C:\Code\TestApps\Stylepad\" class="java.awt.FileDialog" method="actionSetDirectory" />
      <action args="FileDialog Instance,prueba.stylepad" class="java.awt.FileDialog" method="actionSetFile" />
      <action args="FileDialog Instance" class="java.awt.FileDialog" method="actionAccept" />
    </sequence>
```

```
    <action args="JTextPane Instance,0" class="javax.swing.text.JTextComponent" method="actionClick" />
    <assert component="JTextPane Instance" method="getText" value="Archivo de prueba" />
    <assert component="Stylepad" method="getTitle" value="prueba.stylepad" />
    <action args="Stylepad 4" class="java.awt.Window" method="actionClose" />
  </sequence>
  <terminate />
</AWTTestScript>
```

Abbot es una herramienta muy interesante, especialmente por su editor Costello. Sin embargo presenta varios problemas. Durante nuestro trabajo con el editor, este siempre se colgaba antes o después, lo cual resultó muy molesto. La descripción de errores es muy poco clara, ya que se limita simplemente a mostrar el texto de la excepción. Como se puede ver en las capturas, es necesario tener un conocimiento muy detallado de los componentes concretos de la interfaz para poder implementar una prueba. Además, el código XML que genera Abbot es complejo y difícil de generar de manera automática a partir de la información que tenemos para generar pruebas tempranas.

3.3. Implementación de la prueba con JFCUnit

Esta herramienta, al igual que la anterior, permite realizar pruebas unitarias de interfaces gráficas de usuario y pruebas del sistema. A diferencia de las otras dos herramientas empleadas, JFCUnit no cuenta con ningún editor que facilite la tarea de grabar y reproducir pruebas. Para grabar una prueba, es necesario ejecutar un programa en Java como el que se muestra a continuación.

```
import junit.extensions.xml.XMLTestSuite;
import junit.extensions.xml.XMLUtil;
import junit.framework.Test;
import junit.textui.TestRunner;

import org.w3c.dom.*;
import org.w3c.dom.events.*;

public class TestXMLRecording extends XMLTestSuite {

    public static final String DOCUMENT_FACTORY = "javax.xml.parsers.DocumentBuilderFactory";

    public TestXMLRecording() {
        super("xmlrecordingtemplate.xml",
             XMLUtil.readFileFromClassContext(
               TestXMLRecording.class,
               "xmlrecordingtemplate.xml"));
        junit.extensions.jfcunit.WindowMonitor.start();
        Stylepad.main(new String[] {});
        try {
          Thread.currentThread().sleep(3000);
        } catch (InterruptedException ex) {
          ; // Ignore
        }
    }

    public static Test suite() {
        return new TestXMLRecording();
    }

    public static void main(final String[] args) {
```

```
        if (System.getProperty(DOCUMENT_FACTORY) == null) {
            System.setProperty(DOCUMENT_FACTORY,
                "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
        }
        TestRunner.run((Test) TestXMLRecording.suite());
    }
}
```

Para reproducir una prueba es necesario ejecutar otro programa en Java. Dicho programa se muestra a continuación.

```
public class TestXMLReplay extends XMLTestSuite {

    public static final String DOCUMENT_FACTORY = "javax.xml.parsers.DocumentBuilderFactory";

    public TestXMLReplay() {
        super("c:/saved_stylepad.xml",  openFile("c:/saved_stylepad.xml"));
        junit.extensions.jfcunit.WindowMonitor.start();
        XMLRecorder.setReplay(true);
        Stylepad.main(new String[] {});
        try {
            Thread.currentThread().sleep(3000);
        } catch (Exception e) {
          // Ignore
        }
    }

    private static InputStream openFile(final String fileName) {
      try {
        return new FileInputStream(fileName);
      } catch (IOException ioe) {
        return null;
      }
    }

    public static Test suite() {
        return new TestXMLReplay();
    }

    public static void main(final String[] args) {
        if (System.getProperty(DOCUMENT_FACTORY) == null) {
            System.setProperty(DOCUMENT_FACTORY,
                "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
        }
        TestRunner.run((Test) TestXMLReplay.suite());
    }
}
```

El resultado de grabar la prueba se muestra a continuación.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--  An example test suite template             -->
<!--  The first step in creating the test case is  -->
<!--  to document what is going to be tested.      -->
<!--  To do this, create the test suites and       -->
<!--  test cases which are to be recorded.         -->
<!--  2. Setup the JFCXMLTestCase extension,       -->
<!--  to startup your application.                 -->
<!--  3. Run the JFCXMLTestCase defined above.     -->
<!--  4. While the test case is running each       -->
<!--  "record" element will add to the XML.        -->
<!--  5. The save element will write the XML to    -->
<!--  the file name given.                         -->
```

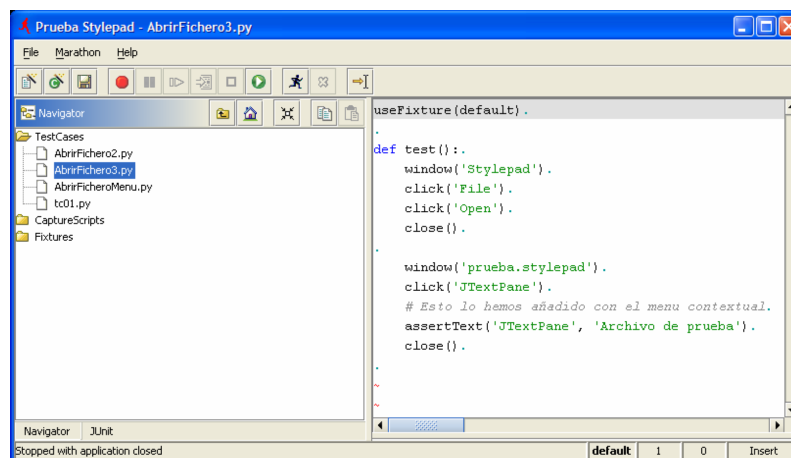
```
<suite name="SwingSet">
   <!--  definition of a local suite  -->
   <suite name="Recording test suite">
      <test debug="true" name="Recording test" robot="true">
            <!-- started recording Tue Dec 27 13:31:37 CET 2005 -->
            <find finder="FrameFinder" id="JFrame1" index="0" operation="equals" title="Notepad"/>
            <find class="Notepad$1" container="JFrame1" finder="AbstractButtonFinder" id="Component2" index="4" label=""
operation="equals"/>
            <click position="custom" reference="19,16" refid="Component2" type="MouseEventData"/>
            <record encoding="UTF-8" file="saved.xml"/>
      </test>
   </suite>

</suite>
```

Esta herramienta presenta varios problemas. Uno de los más importantes es que la herramienta no ha capturado correctamente la interacción con el cuadro de selección de fichero. Cuando se reproduce la prueba anterior, se muestra el cuadro de selección de archivo y ahí se detiene. Además, la documentación no detalla la signatura y sintaxis del lenguaje de etiquetas utilizado para guardar una prueba, por lo que es difícil de entender la manera de trabajar de esta herramienta. Además, este lenguaje también desvela información de la implementación, por lo que no se puede utilizar en la implementación de pruebas tempranas

Como curiosidad citamos que esta ha sido la única herramienta capaz de grabar una prueba sobre la aplicación de bloc de notas.

3.4. Implementación de la prueba con Marathon

A diferencia de las dos herramientas anteriores, Marathon está centrada en pruebas del sistema y de aceptación, en lugar de pruebas unitarias de interfaces gráficas. Esta herramienta también incluye un entorno gráfico que permite grabar y reproducir pruebas. Una captura de este entorno se muestra a continuación.



Sin embargo este entorno no incluye tantas opciones como Costello.

Como se puede apreciar en la captura, esta herramienta no utiliza un lenguaje de etiquetas para almacenar los casos de uso, sino lenguaje Python. El resultado de la grabación de la prueba se muestra a continuación.

```
useFixture(default)
```

```
def test():
	window('Stylepad')
	click('File')
	click('Open')
	close()

	window('prueba.stylepad')
	click('JTextPane')
	# Esto lo hemos añadido con el menú contextual
	assertText('JTextPane', 'Archivo de prueba')

	close()
```

Además, la herramienta de captura incluye un menú contextual sobre la aplicación a prueba que permite añadir comprobaciones al mismo tiempo que se graba.

Sin embargo esta herramienta también presenta problemas. El más importante es que, al igual que en la herramienta anterior, no se ha capturado la interacción con el diálogo para abrir un archivo. Tampoco hemos encontrado en la documentación la forma de añadir esta parte, ya que, como se observa, Marathon utiliza como identificador el nombre o título de la ventana y el diálogo para seleccionar un fichero no tiene título.

3.5. Conclusiones

En primer lugar vamos a evaluar las herramientas en función de su lenguaje. Lo que más se valora es que este lenguaje oculte los detalles de la implementación, para que pueda ser utilizado con independencia de la tecnología del sistema (lenguaje, y aplicación de escritorio o web). En segundo lugar evaluamos las herramientas en función de su madurez.

No es difícil manipular programáticamente un lenguaje de script, como muestra Maratón. De hecho, esta solución es la que mejor ha ocultado los detalles de la implementación. Sin embargo una de las principales pegas es la flexibilidad. Trabajando con XML es sencillo convertirlo en cualquier lenguaje de script o en un lenguaje de XML concreto para una herramienta. Esta flexibilidad se pierde al utilizar un lenguaje. Sin embargo, es más sencillo escribir en lenguaje de script que en XML.

En cuanto a la madurez de las herramientas, hemos constatado que ninguna de las herramientas es la definitiva.

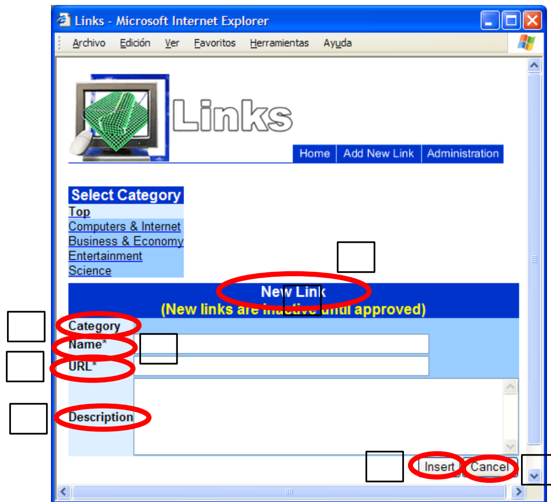
4. Implementación de los casos de prueba de la aplicación web

A continuación se describe con mayor detalle la implementación de las pruebas sobre la aplicación web y se evalúan distintas herramientas utilizadas.

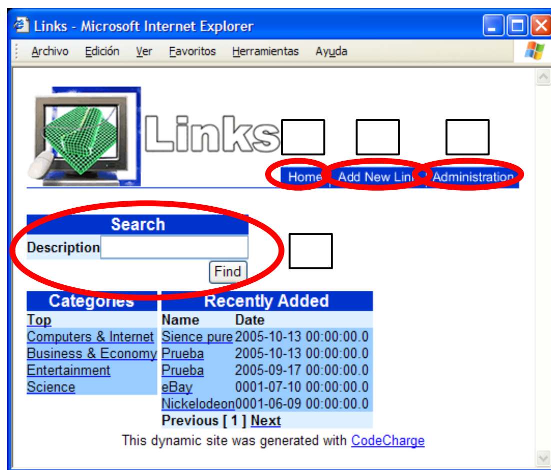
4.1. Planteamiento

La aplicación a prueba es una aplicación web que permite gestionar un catálogo de enlaces on-line. A continuación se muestra el caso de prueba seleccionado.

Nombre	Añadir un nuevo enlace con éxito.
Objetivo	Este caso de prueba verifica el comportamiento del caso de uso en su escenario principal.
Acciones	00 -> 01(categoriaEnlace) -> 02(datosEnlace) -> 03 00 La prueba carga la página principal y selecciona la opción de añadir nuevo enlace. 02 El sistema selecciona la categoría “top” y muestra el formulario. 03 La prueba introduce los datos del nuevo enlace. 04 El sistema almacena el nuevo enlace.
Valores de prueba	*categoriaEnlace* = “top” *datosEnlace.name* = “Prueba” *datosEnlace.URL* = “http://www.prueba.com” *datosEnlace.description* = “Un link de prueba”
Resultados observables	00 -> 01 [V01] -> 02 -> 03 -> F [V02] V01 Tabla X V02 Tabla Y
Notas	El enlace no aparece en la sección de últimos enlaces añadido porque debe ser validado por el administrador para poder mostrarse



Cualidad	Expresión regular
01	“New Link”
02	“Category”
03	“Name*”
04	“URL*”
05	“Description”
06	“Insert”*
07	“Cancel”*



Cualidad	Expresión regular
01	“Home” *
02	“Add New Link” *
03	“Administration” *
04	“Search .* Description .* Find” *

Para implementar esta prueba también se han buscado herramientas de libre distribución con la suficiente documentación para poder trabajar con ellas. Por comodidad todas las herramientas seleccionadas están implementadas en la plataforma Java. Las herramientas evaluadas son : JWebUnit, Canoo WebTest y Avignon.

No vamos a incluir en las pruebas el acceso a la base de datos para comprobar que el nuevo enlace se registró correctamente.

4.2. Implementación de la prueba con JWebUnit

Esta herramienta es una fachada de otra herramienta similar, llamada HttpUnit. JWebUnit funciona de manera similar a un navegador, ofreciendo un conjunto de métodos en Java para simular las acciones de un navegador (cargar enlace, rellenar formulario, hacer click) y para verificar los elementos recibidos (el título de la página, las imágenes, las tablas, etc.). El código de la prueba se muestra a continuación.

```
import net.sourceforge.jwebunit.WebTestCase;

public class PruebaWebLink
```

```
    extends net.sourceforge.jwebunit.WebTestCase
{
    String url;

    public PruebaWebLink () {
        this.url = "http://localhost:8080/links_jsp";
    }

    public void setUp() {
      getTestContext().setBaseUrl(this.url);
    }

    /** La prueba propiamente dicha **/
    public void testInsertarEnlaceCorrecto() {
        // 1. Accedemos a la páhina principal
        beginAt("/Default.jsp");

        // 2. Pulsamos en el enlace de añadir nuevo enlace
        clickLinkWithImage("images/add.gif");

        // 3. Comprobamos que hemos llegado a la página
        // Formulario
        assertFormPresent ("Link");
        // Cualidad 01
        assertTextPresent("New Link");
        // Cualidad 02
        assertTextPresent("Category");
        // Cualidad 03
        assertTextPresent("Name*");
        // Cualidad 04
        assertTextPresent("URL*");
        // Cualidad 05
        assertTextPresent("Description");
        // No se ha podido verificar la presencia de los botones del formulari
        // porque el únito método dispible comprueba un atributo que no está
        // presente en los botoens del formulario
        // Cualidad 06
        // assertButtonPresent("Insert");

        // 4. Rellenamos el formulario con los valores de prueba
        setFormElement("name", "Prueba");
        setFormElement("link_url", "http://www.prueba.com");
        setFormElement("description", "Un link de prueba");

        // 5. Pulsamos aceptar
        submit();

        // 6. Comprobamos que hemos vuelto a la página principal
        // Comprobamos que hay un formulario
        assertFormPresent();
        // Enlaces
        // Los enlaces son imágenes, no textos
        // Cualidad 01
        assertLinkPresentWithImage("images/home.gif");
        // Cualidad 02
        assertLinkPresentWithImage("images/add.gif");
        // Cualidad 03
        assertLinkPresentWithImage("images/admin.gif");
        // Cualidad 04
        assertTextPresent("Search");
        assertTextPresent("Description");

        // Tenemos aquí el mismo problema que antes con los botones
     }
}
```

Esta herramienta ofrece un conjunto muy completo de métodos para interactuar con las páginas HTML. Para la prueba hemos intentado elegir aquellos métodos que necesitan menos detalles concretos de la implementación

Esta herramienta no soporta expresiones regulares, por lo que no las hemos incluido en las pruebas. Sin embargo, extender JWebUnit para verificar expresiones regulares e suna tarea trivial y no realizada por falta de tiempo.

4.3. Implementación de la prueba con Canoo WebTest

Esta herramienta está construida sobre Ant y HttpUnit (como JWebTest). Su principal aportación, es permitir implementar pruebas sin programas en lenguaje Java. Para ello, WebTest define su propio lenguaje XML.

El código de la prueba añadir un nuevo enlace con éxito se muestra a continuación. Esta herramienta sí ofrece soporte para expresiones regulares, por lo que las hemos utilizado siempre que nos ha sido posible.

```
<project name="WebLinks" basedir="." default="main">

 <property name="webtest.home" location="C:/Code/TestApps/Weblinks" />
 <description>  </description>
   <import file="${webtest.home}/lib/taskdef.xml"/>

 <target name="main">
   <webtest name="myTest">
     <config      host="localhost"
       port="8080"
       protocol="http"
       basepath="links_jsp" />

     <steps>

       <!-- Accedemos a la pagina principal -->
       <invoke     description="Acceso a la pagina principal"
       url="Default.jsp" />

       <!-- Pulsamos enlace del formulario de alta -->
       <clickLink    description="Acceso a la pagina de nuevo enlace"
       href="LinkNew.jsp" />

       <!-- Asertos -->
       <verifyText description="01 Titulo del formulario"
       text="New Link"  />
       <verifyText description="02 Campo categoria"
       text="Category"  />
       <verifyText description="03 Campo nombre"
       text="Name*"  />
       <verifyText description="03 Campo nombre"
       text="Name*"  />
       <verifyText description="04 Campo URL"
       text="URL*"  />
       <verifyText description="05 Campo descripcion"
       text="Description"  />
       <verifyText description="06 Boton insertar"
       text="Name*"  />
```

```
        <!-- Rellenamos el formulario -->
        <setInputField description="set user name"
                    name="name"
                    value="Prueba" />
        <setInputField description="set password"
                    name="link_url"
                    value="http://www.prueba.com" />
        <setInputField description="set password"
                    name="description"
                    value="Un link de prueba" />

        <!-- Pulsamos el boton -->
        <clickButton  label="Insert"
                  description="Pulsar en el boton de insertar" />
        <!-- Asertos -->
        <verifyText description="04 Comprueba el formulario de busquega"
        text="Search(.*)Description(.*)"
        regex="true" />

     </steps>
   </webtest>
  </target>
</project>
```

En este caso concreto, la utilización de XML presenta varios problemas. En primer lugar, algunos caracteres en español provocan errores. Dado que esta herramienta no permite incluir la cabecera de un documento XML con la codificación empleada, ignoramos como solucionar este error. Otro error importante es que

Esta herramienta tiene el mismo fallo que la herramienta Abbot. Cuando sucede un error no programático (por ejemplo el servidor web no está disponible), muestra todo el mensaje de la excepción construida por la maquina virtual Java, lo cual incluye mucha información inútil.

Por un lado, esta herramienta incorpora opciones que no tiene la herramienta anterior, como la posibilidad de guardar resultados en archivos y utilizarlos en las verificaciones, emplear expresiones XPath, o verificar archivos PDF. Además, aunque en esta prueba no lo hemos hecho, los valores de prueba se pueden guardar en archivos externos, lo que facilita la reutilización de una prueba con distintos valores de prueba.

Por otro lado, su lenguaje de etiquetas también tiene carencias. No existen etiquetas específicas para comprobar enlaces, imágenes o botones del formulario. Esta carencia podría resolverse utilizando la etiqueta *verifyText* para buscar las etiquetas HTML. Sin embargo, el uso de esta etiqueta requiere utilizar el código HTML y revela detalles de la implementación, además no hemos sido capaces de conseguir que la herramienta aceptara etiquetas como parte del texto de un atributo para que compruebe la existencia de los botones en el formulario para añadir un nuevo enlace y las imágenes de los enlaces en la página principal.

4.4. Implementación de la prueba con Avignon

Esta herramienta es muy similar a la herramienta anterior. Aunque Avignon se define como un lenguaje de prueba independiente, con las herramientas con las que se distribuye sólo es posible utilizarlo para probar aplicaciones web. Avignon también trabaja con su propio lenguaje basado en XML y también utiliza la herramienta

HttpUnit, ancestro común de las tres herramientas empleadas en esta sección. A diferencia de WebTest, Avignon incorpora un juego de etiquetas para acceso a bases de datos. Sin embargo por los motivos expuestos en el documento de casos prácticos, no utilizaremos dichas etiquetas en la prueba.

La principal diferencia que encontramos, aparte de la escasez de documentación y etiquetas en su lenguaje, es que trabaja con un explorador web (en nuestro caso el Internet Explorer) en lugar de acceder directamente al servidor web, como JWebUnit y Canoo WebTest. No vemos que esta diferencia tenga ninguna consecuencia.

Finalmente no vamos a utilizar estas herramientas: la documentación es pobre, no se describen todas las etiquetas ni su modo de uso. Los ejemplos también son muy pobres. No hemos descubierto como realizar algunas acciones, por ejemplo, para pulsar en un enlace es necesario indicar el texto del enlace. Sin embargo, como ya se ha visto, el enlace que la prueba debe seleccionar con contiene texto, sino una imagen. Además no hemos encontrado etiquetas para realizar verificaciones, sino que las verificaciones se realizan mediante expresiones XPath, lo cual añade una complejidad innecesaria e injustificada.

4.5. Conclusiones

Trabajar con una interfaz de usuario expresada en HTML ha sido mucho más sencillo que trabajar con una interfaz gráfica en Java (Swing). Las herramientas presentan mayor madurez y opciones y, salvo en el caso de Avignon, no se han encontrado ningún problema.

Ni con JWebTest ni con Canoo WebTest se han podido verificar la presencia de los botones del formulario. Esto sugiere la necesidad de extender las opciones de verificación de estas herramientas. Sin embargo, como ya hemos mencionado, estas herramientas son lo suficientemente completas y maduras para poder utilizarse con garantías.

Si se comparan el código de las pruebas implementadas en esta sección con el código de las pruebas implementadas en la sección anterior se observa que, pese a provenir de casos de uso especificados de la misma manera y expresar ambas interacciones entre un usuario humano y el sistema, ambos presentan un gran número de diferencias. Se hace patente la necesidad de obtener un lenguaje único para expresar pruebas sobre aplicaciones de escritorio y aplicaciones web.