Norma - Pruebas automáticas funcionales aplicaciones Java

Oficina de Calidad

Versión: 1.0.1

Control de versiones

Versión	Fecha	Autor	Descripción / Comentarios
V00_v01	05-07-2018	Oficina de Calidad	Versión Inicial
V00_v02	26-10-2018	Oficina de Calidad	A Revisar
V00_v03	26-11-2018	Oficina de Calidad	Publicación
V00_v04	18/03/2020	Oficina de Calidad	Referencia a la normativa Maven por el proyecto de unificación metodologia
V00_v05	04/09/2020	Oficina de Calidad	Se cambian las referencias a los documentos con la palabra “NORMATIVA” por “NORMA” ya que han sido cambiados de nombre para cumplir lo indicado en MIR-NORMA_COD_DOCUMENTACION.docx
0.6.0	15/02/2021	Oficina de Calidad	Se actualizan las referencias del apartado “DOCUMENTOS RELACIONADOS”
1.0.0	26/02/2021	Oficina de Calidad	Unificación documento
1.0.1	27/07/2021	Oficina de Calidad	Estandarización del criterio de cobertura de pruebas

Índice

ÍNDICE DE ILUSTRACIONES

1. INTRODUCCIÓN

1.1. OBJETO

La normativa detallada en el presente documento es de obligado cumplimiento para todas aquellas personas, equipos y empresas responsables de la definición, implementación, ejecución y evaluación de resultados de los Planes de Prueba requeridos dentro del marco de aseguramiento de la calidad de los sistemas de información desarrollados para la SGSICS

El objetivo del documento es definir unas directrices a seguir por los distintos equipos (desarrolladores, pruebas, jefes de proyecto) para el mejor funcionamiento del proceso de desarrollo del software y pruebas automatizadas.

En este documento se definirá una normativa (buenas prácticas) en el desarrollo de las pruebas funcionales automáticas para las aplicaciones java y su ejecución.

1.2. ALCANCE

En este documento se define la estructura de las pruebas funcionales usando la herramienta SELENIUM, en especial “Webdriver”, sus elementos, su mantenimiento y su evolución.

1.3. GLOSARIO

Término	Descripción
SGSICS	Subdirección General de Sistemas de Información y Comunicaciones para la Seguridad
MIR	Ministerio del Interior
SUT	System Under Test (Sistema bajo prueba)

1.4. DOCUMENTOS RELACIONADOS

/Ubicación/Documento	Descripción
PublicacionIntranet/22-PRUEBAS/MIR-INT-CALIDAD-PLAN_PRUEBAS.docx	Plan de pruebas, definición de la metodología a seguir para la definición de los Planes de Pruebas de los Sistemas de Información desarrollados para la SGSICS en el marco de la Gestión de la Calidad del software.
PublicacionIntranet/22-PRUEBAS/MIR-GUIA-Testlink-ELABORACION PLAN DE PRUEBAS.docx	Manual de usuario de TestLink. Descripción del uso y manejo de la herramienta TestLink.
PublicacionIntranet /plantillas/doc/01.ASI/MIR-PROY-ASI-ESPECIFICACIONES.docx	Entregable de Especificaciones

PublicacionIntranet/plantillas/doc/04.IAS/MIR-PROY-IAS-INFORME_PRUEBAS.docx	Entregable Informe de Pruebas
PublicacionIntranet/03.CSI/MIR-INT-NORMA-GRADLE.docx	Normativa para creación de proyecto Gradle
PublicacionIntranet/03.CSI/MIR-INT-NORMA-MAVEN.docx	Normativa para creación de proyecto Maven
PublicacionIntranet/22-PRUEBAS/MIR-NORMA-Testlink.docx	Normativa de creación de proyectos Testlink

2. DESCRIPCIÓN GENERAL DE LA PLATAFORMA DE PRUEBAS DEL MIR

En el siguiente diagrama se muestra la arquitectura de la plataforma de Integración Continua sobre la que se añadido los componentes necesarios para la realización de pruebas funcionales.

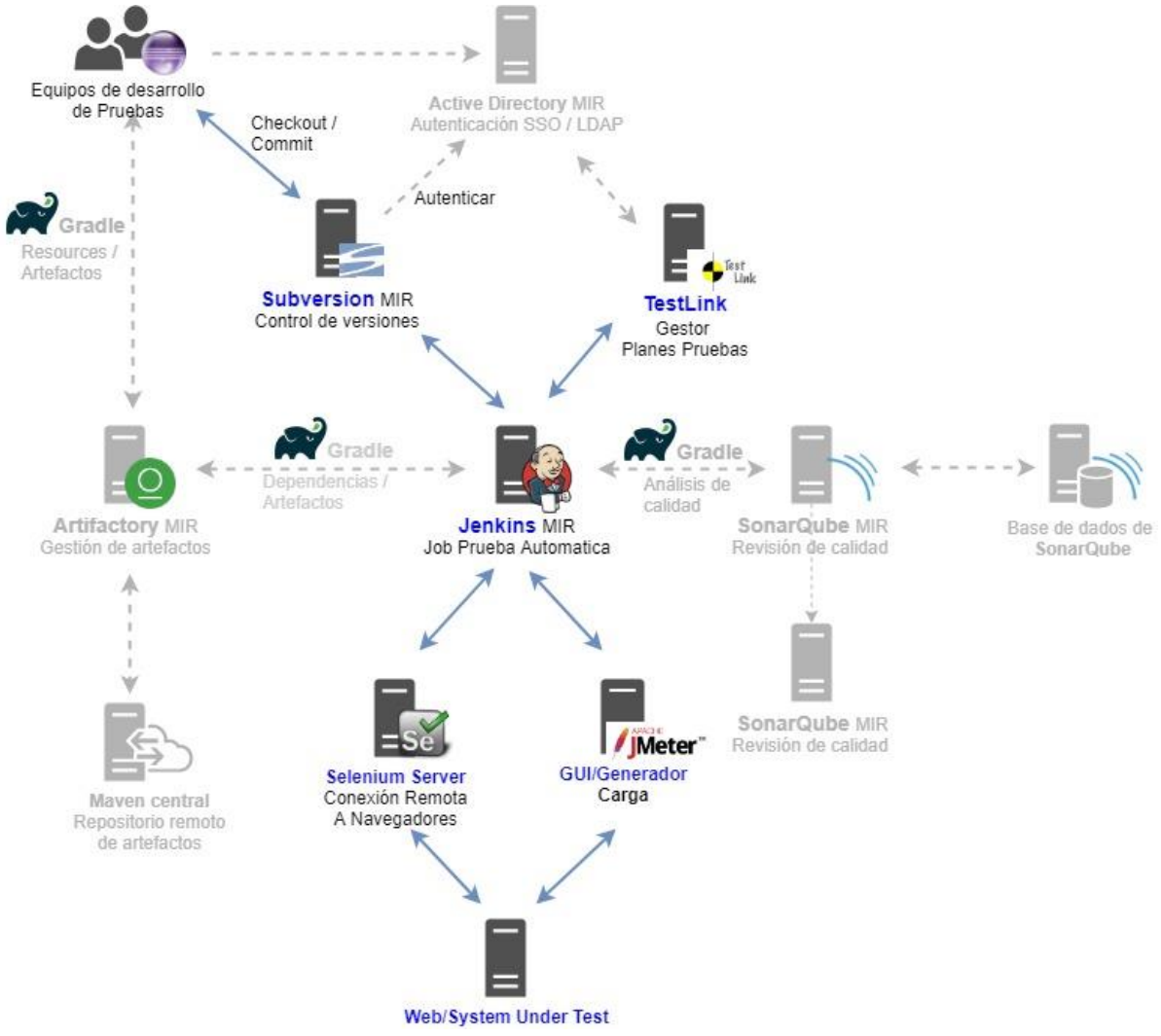


Ilustración 1. Arquitectura entorno de Pruebas Funcionales

Actualmente el sistema de pruebas funcionales consta de los siguientes elementos:

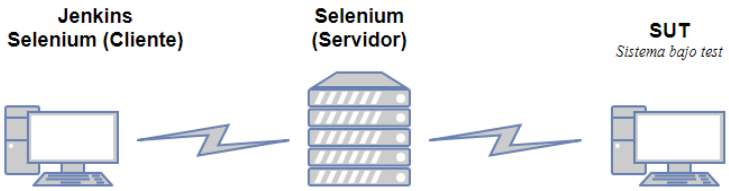


Ilustración 2. Estructura Selenium

3. SELENIUM

Selenium es un paquete de pruebas de software automatizado gratuito (código abierto) para aplicaciones basadas en la web en diferentes navegadores y plataformas.

Selenium no es solo una herramienta, sino un conjunto de software, cada uno de los cuales atiende las diferentes necesidades de prueba de una organización.

Tiene tres componentes:

- Selenium Integrated Development Environment (IDE)
- WebDriver + Selenium Remote Control (RC) = Selenium 3
- Selenium Grid

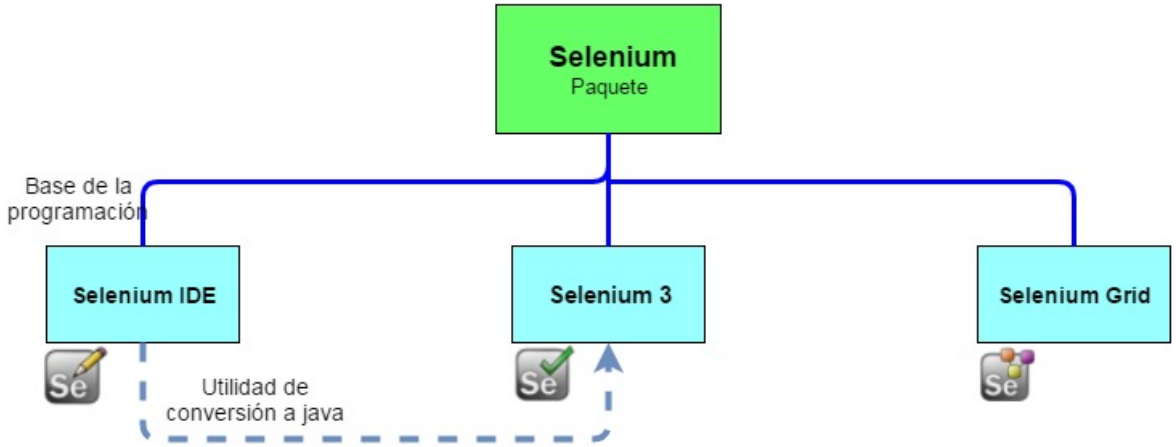


Ilustración 3. Selenium Componentes

Selenium IDE es un entorno de desarrollo integrado para pruebas con Selenium. Está implementado como una extensión de Firefox, provee una herramienta de grabar/reproducir para crear pruebas sin usar un lenguaje de scripting para pruebas. Incluye también un lenguaje específico de dominio para pruebas (**Selanese**).

WebDriver acepta comandos (enviados en Selenese o vía el API de cliente) y los envía a un navegador (HTTP). Esto se implementa a través de un controlador del navegador específico para cada navegador que envía los comandos y trae los resultados de regreso. Además, hace posible

escribir pruebas automatizadas para aplicaciones web, en cualquier lenguaje de programación lo que permite una mejor integración de Selenium a entornos de prueba existentes. Para realizar la codificación de las pruebas más fácil, Selenium actualmente provee controladores de dispositivos para PHP, Python, Ruby, .NET, Perl y Java. Las pruebas pueden ejecutarse entonces usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.

Selenium Grid es un servidor que permite usar instancias de navegador ejecutándose en máquinas remotas. Permite ejecutar pruebas en paralelo en múltiples máquinas y manejar diferentes versiones y configuraciones de manera centralizada.

4. PROCESO DE AUTOMATIZACIÓN

Para el proceso de automatización de las pruebas funcionales, usaremos Selenium, la parte correspondiente al WebDriver, que permite programar pruebas automáticas usando un lenguaje de alto nivel, como se ha comentado anteriormente y en otros documentos, se usará Java, que hará uso del API que nos proporciona.

Selenium ofrece una API completa (actualmente, sigue evolucionando, sacando constantemente versiones con mejoras y correcciones).

También la herramienta soporta diferentes navegadores en la que puede interactuar su API. En nuestro caso solo se usarán los navegadores Chrome y FireFox.

La plataforma de pruebas automáticas del MIR facilitar la ejecución de las pruebas funcionales en la cual se puede realizar:

- La ejecución de las pruebas de forma desatendida, se encarga la herramienta Jenkins, la cual puede lanzar las pruebas a cualquier hora, en días no laborables y por la noche para al día siguiente tener los resultados de la prueba, por ejemplo.
- Ejecutar las pruebas un número mayor de pruebas, si se ha automatizado un Caso de Uso o un Caso de Prueba, éste se puede ejecutar varias veces con distintos datos de prueba.
- Ejecuciones de las pruebas en paralelo, se puede lanzar pruebas en concurrencia con distintos navegadores y S.O. En nuestro caso, se puede realizar pruebas en los navegadores Chrome y Firefox en Linux (Red Hat).
- Pruebas de regresión y smoke, una vez que se tienen automatizados los casos de uso o casos de prueba, un subconjunto de estas, pueden ser reutilizadas para incluirlas en las pruebas de regresión y smoke.

5. PROCEDIMIENTO DE PRUEBAS FUNCIONALES AUTOMÁTICAS

5.1. ANÁLISIS DE PRUEBAS

Esta actividad consiste en realizar un análisis de la fase de pruebas funcionales automáticas de una aplicación con el fin de conocer el nivel de certificación de la aplicación, así como el nivel de profundidad de la fase de pruebas para las distintas funcionalidades de la aplicación.

Hay que recordar que en todo desarrollo software, debe tener su batería de pruebas, en nuestro caso pruebas funcionales automáticas y son obligatorias, en la cual tendrá más o menos número de pruebas y con distintos niveles de profundidad o detalles que serán valoradas por el responsable funcional.

5.1.1. Casos de uso

Un **caso de uso** es una descripción de los pasos o las actividades que deberán realizarse para llevar a cabo algunas transacciones o proceso, además hay que tener en cuenta lo que el sistema no debe hacer. En el MIR se usa el concepto de “Casos de Uso”.

Los casos de uso deben ser priorizados y dicha priorización debe realizarla preferiblemente el jefe de proyecto MIR o responsable funcional, junto con los usuarios.

Una correcta priorización de los casos de uso es clave para las pruebas, ya que de esta priorización se obtendrá la profundidad de las pruebas funcionales automáticas a realizar.

A continuación, se detallan algunos **criterios** que pueden servir como guía para la priorización de los casos de uso:

- Funcionalidad crítica/importante para el negocio
- Volumen de usuarios al que da/dará servicio
- Proporciona información a otras aplicaciones con funcionalidades importantes o críticas
- Valoración del impacto de fallo/no disponibilidad de la aplicación en el negocio
- Posibilidad de alternativas en periodos de fallo
- Impacto en la imagen del organismo en caso de fallo
- Frecuencia de uso de la funcionalidad
- Complejidad de la funcionalidad
- Solución técnica que lo soporta
- Frecuencia de cambios que soporte
- Volumen de información que maneja
- Número de incidencias/fallos detectados con anterioridad
- Tiempos y plazos que se dispone realizar el desarrollo

Los casos de uso que tengan prioridad Alta se definirán como funcionalidades críticas de la aplicación, y su criticidad se verá reflejada en los planes de prueba funcionales con una mayor cobertura de casos de prueba que los casos de uso de prioridad inferior (Media y Baja). Y serán candidatas a ser automatizadas.

5.2. ESPECIFICACIÓN DE PRUEBAS

El plan de pruebas funcional se define como un conjunto de casos de prueba que permiten verificar que el sistema cumple las necesidades establecidas por el usuario en los requisitos. También se debe tener en cuenta la realización de una valoración si el caso de prueba es candidata para su automatización.

5.2.1. Cobertura de pruebas

En función de la criticidad de un caso de uso, la cobertura de casos de prueba que debe crearse será distinta. Se tendrán las siguientes dos situaciones:

- **Casos de uso de prioridad Alta.** Será necesario identificar y documentar tantos casos de prueba como sean necesarios para que toda la funcionalidad del caso de uso quede cubierta. Para cubrir toda la casuística de un caso de uso será necesario crear un caso de prueba por cada uno de los caminos principales, caminos secundarios, caminos de error funcional y caminos de no éxito existentes en la funcionalidad. Todos estos casos de prueba deberán ser automatizados, dependiendo siempre de la decisión del jefe de proyecto MIR o responsable funcional.
- **Casos de uso de prioridad Media y/o Baja.** Solo será necesario identificar y documentar dos casos de prueba, uno que cubra el camino principal del requisito funcional y otro un camino de error funcional (Consideraciones con los casos de prueba). Un porcentaje mínimo de casos de uso con esta prioridad serán candidatos a ser automatizados, dependerá de la decisión del jefe de proyecto MIR o responsable funcional.

En general, es que cada caso de uso debe realizar al menos 2 flujos o caminos, un camino principal y otro no exitoso (camino de error funcional), que corresponderán a casos de prueba y que se debe reflejar en Testlink y en su caso de uso automatizado.

Camino Principal. Describe los pasos normales, entendiendo como normal la funcionalidad esperada, que se realizan durante la interacción con el sistema, cuya ejecución culmina con la obtención de los datos esperados. Se podría definir como el escenario exitoso desde el principio al fin.

Caminos Secundarios o Alternativos. Son los que nos permiten indicar qué es lo que hace el sistema en los casos menos frecuentes e inesperados. Describen un conjunto de pasos secuenciales originados a partir de la evaluación de una condición o regla, cuyo origen es un paso del camino principal.

Caminos de error funcional. Son los caminos que despliegan los mensajes de error o precaución, generalmente cuando se usan datos o acciones inválidas.

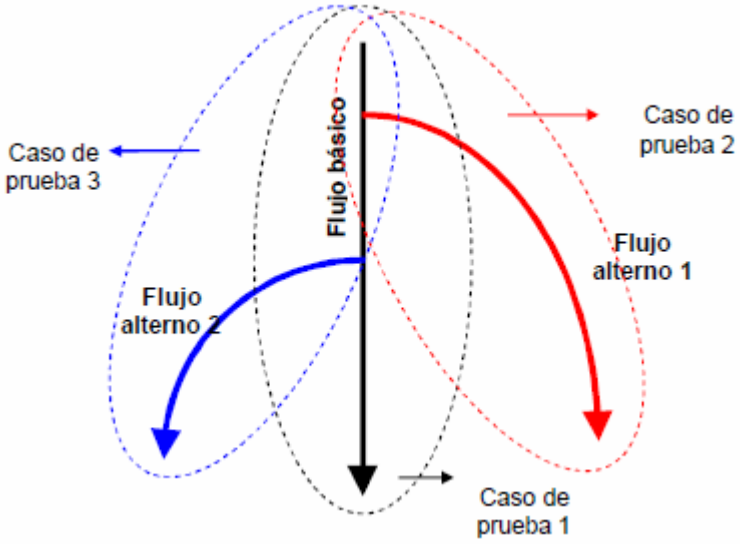


Ilustración 4. Flujos de pruebas

5.2.2. Consideraciones con los casos de prueba

A continuación, se indican una serie de consideraciones a tener en cuenta en el diseño de los casos de prueba:

- **Dos enfoques de pruebas**. Los casos de prueba deben tener dos enfoques, por un lado, debe verificar que el software hace lo que se espera de él, y segundo y menos obvio, verificar que el software no hace lo que no se espera de él.
- **Máxima cobertura funcional**. Un caso de prueba debe diseñarse para que cubra el máximo número de posibilidades de entrada diferentes que den como resultado un comportamiento común y único a nivel funcional con el fin de reducir el total de casos. Como posibilidades de entrada se pueden definir conjuntos de datos válidos o no válidos.
- **Independencia funcional**. Como se ha comentado anteriormente los casos de uso contienen casos de pruebas (caminos). Los casos de prueba automáticos deben diseñarse lo más independientes posibles desde el punto de vista funcional, es decir, un caso de prueba automático no debe depender del resultado de otros casos de prueba. O sea, no pueden existir casos de prueba que sean ejecutados previamente como generadores de datos de otros casos de prueba. Para evitar esto, el juego de datos tiene que ser lo suficientemente amplio como para probar de forma independiente los diferentes casos de pruebas.

- **Parametrización de datos de entrada**. La parametrización se define como la creación de variables en los casos de prueba que alojan el juego de datos, de manera que permite ejecutar el caso de prueba con valores de datos diferentes. Se deben parametrizar todos los datos de entrada a través de un fichero, txt o xml, que el código de prueba lo use o los datos estén dentro del código y este bien identificado. Se debe proporcionar al menos 2 juegos de datos para la realización de las pruebas automáticas o en su defecto 1 juego de datos y además, en código o en un documento se debe indicar cómo generar más juegos de datos, así como son añadidos al caso de prueba automático., si en el caso que se decida documentarlo debe estar disponible en la estructura de subversión correspondiente a la documentación.
- **Parametrización de resultados esperados**. Los resultados de los casos de prueba se envían a la herramienta Testlink, estos resultados deben tener solo dos valores, correcto (OK) o incorrecto (NooK). Y deben ser coherentes con el contenido del caso de prueba que está definido en la herramienta.

5.2.3. Código del caso de prueba

Los casos de prueba deberán ser escritos con el detalle suficiente para que cualquier desarrollador en un futuro pueda realizar modificaciones y mantenimiento del código para ello debe usar los PageObjects.

Además, tenemos que tener a nivel del código una serie de datos que nos describa el caso de prueba, a través de comentarios dentro del código, como identificar el caso de prueba, que hace cada función, que paramentos de entrada y salida necesita, condiciones iniciales y finales, etc.

Un caso de prueba identificado debe tener al menos la siguiente información que permita identificar de forma clara el caso de prueba y su trazabilidad con el caso de prueba correspondiente al Testlink:

- Identificación del caso de prueba. Corresponde a la identificación que se ha asignado en la herramienta TestLink, según la normativa de Testlink.

  Ejemplo: SIAVT-45:GF04:CU002:CP001 Envío de documento permitido al portafirmas
- Descripción. Una breve descripción sobre el aspecto de la funcionalidad que se está probando.
- Objetivo. Propósito del caso de prueba.
- Condiciones de ejecución. Condiciones que se deben cumplir antes de iniciar el caso de prueba.

Un caso de prueba debe estar documentado su código además de lo anterior debe tener la siguiente información:

- Datos de entrada. Como se obtiene los datos de entrada, al ser pruebas automáticas no pueden depender de la acción del usuario. Por ejemplo: nombre del fichero, tipo, estructura, lista de variables, etc.
- Datos de las funciones o métodos: En cada función o métodos debe existir una descripción breve del objetivo de la función o método, listas de parámetros entrada y salida.

Para ello se recomienda usar los “Code templates” del editor que se esté usando el desarrollador.

Ejemplo: Eclipse IDE

Para activar los Code Template, se accede a través del menú de “Project” → “Properties” → “Code Templates”

Viene desactivado por defecto, y marcamos “Enable Project ……” y “Automatically add…”

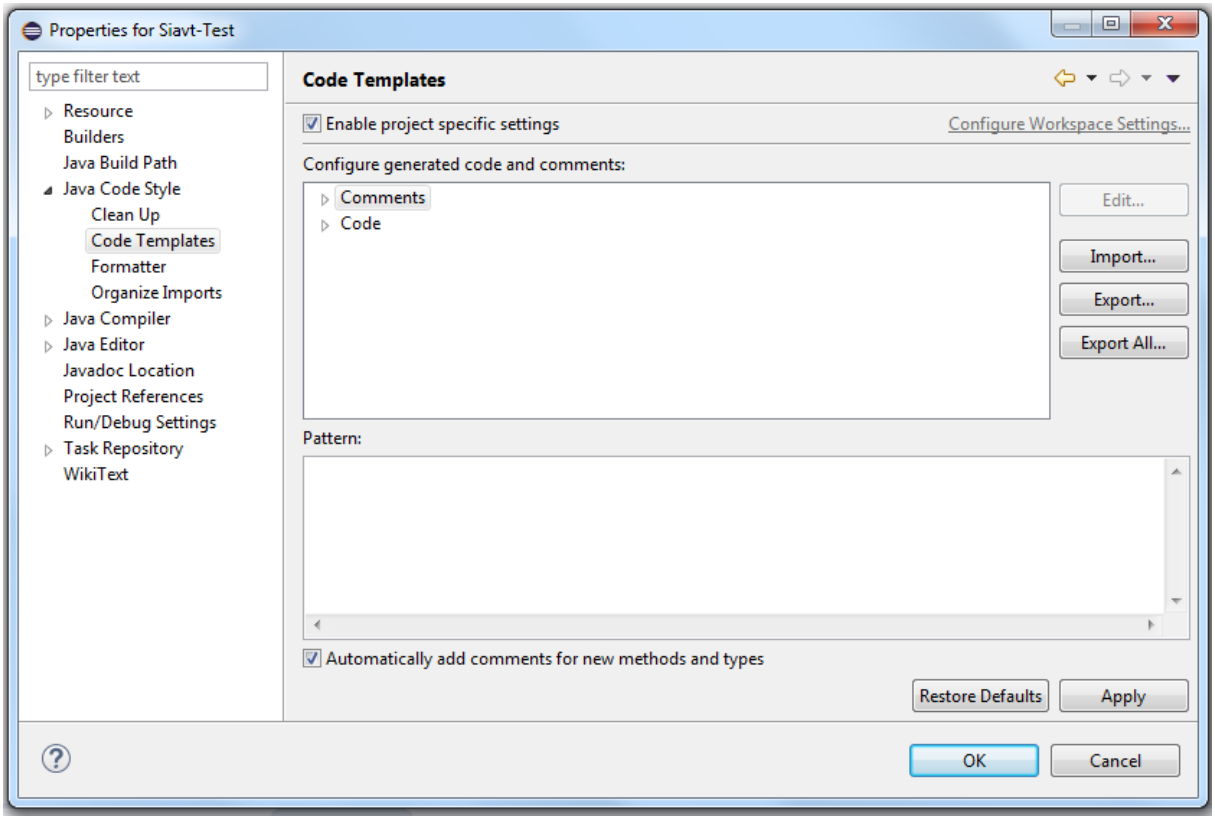


Ilustración 5. Code Templates - Eclipse

Para la ejecución de las pruebas automáticas tiene que tener dos entregables:

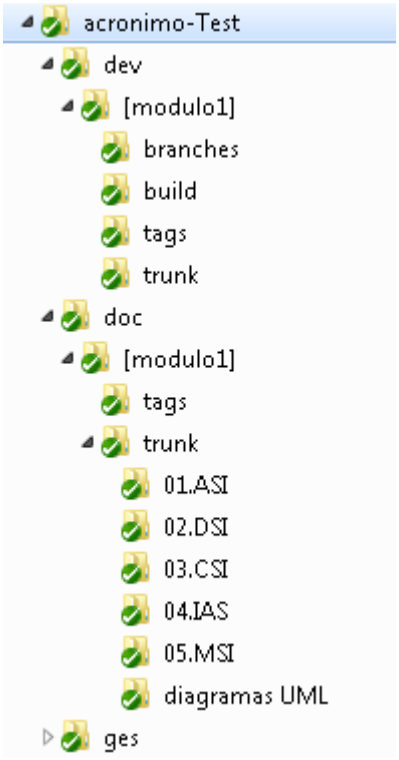
- Plan de prueba, se habrá dado de alta en la herramienta de Testlink.
- Código fuente (Java), que estará en subversión.

5.2.4. Construcción del plan de pruebas automáticas

Después de explicar en los apartados anteriores la línea que debe seguir, en esta fase, el desarrollador de las pruebas automáticas debe comenzar a desarrollar los casos de prueba, teniendo en cuenta lo siguiente:

En la herramienta de control de versiones, en este caso subversión se usará la siguiente estructura en la cual se almacenará el código de la prueba. Si no existe la estructura se tendrá que solicitar a Calidad de Software (sgsics.calidadsw@interior.es).

La estructura será la siguiente:



El código de la prueba debe ir en:

<acrónimo>-Test/dev/[modulo1]/trunk/src/java

Para la documentación del código, si es necesaria se deberá ir en:

<acrónimo>-Test/doc/[modulo1]/trunk

Acrónimo corresponde al valor del “acrónimo” que existe en el fichero build.gradle de la aplicación a probar.

Por ejemplo, el acrónimo de la aplicación de SIAVT en el build.gradle.

```
plugins {
  id 'project-report'
  id 'org.sonarqube' version '2.5'
  id "jacoco"
}

defaultTasks 'clean', 'build'

//Información global de proyecto
ext {
  projectInfo = [
    description : """Sistema integral de ayudas a victimas del terrorismo""",
    organization : 'Ministerio del Interior',
    developmentVendor : 'RICOH',
    acronimo : 'siavt',
    group: "es.mir.${rootProject.name}",
    name : "${rootProject.name}",
    version : '1.32.8',
    encoding : 'UTF-8',
    sourceJdkVersion : '1.7',
    targetJdkVersion : '1.7'
  ]
}

apply from: "gradle-env-conf/environmentBuildConfig.gradle"
```

Ilustración 6. Build.gradle de SIAVT

Cuando se realice una entrega de las pruebas automáticas, esta se realizará a través de la generación de una etiqueta <tag>. El nombre de la etiqueta corresponde a la versión del código, explicado más abajo.

La versión del código debe seguir lo que se indica en el apartado de “Versionado Semántico” de la normativa Gradle / Maven y será igual que para el código de la aplicación.

Teniendo en cuenta, que si la aplicación a probar corresponde siavt-3.1.0.

La versión del código de prueba correspondería a “funcionales-siavt-3.yy.zz”

- Nombre del proyecto. Corresponde al <rootProject.name>, en el ejemplo “siavt”.
- Dada una versión MAJOR.MINOR.PATCH
  - MAJOR. Debe coincidir con la versión del software a probar, en el ejemplo “3”.
  - MINOR. “yy”, como se indica en la normativa Gradle / Maven (Versionado Semántico).
  - PATCH. “zz”, como se indica en la normativa Gradle / Maven (Versionado Semántico).
- Prefijos: Tipos de prueba, nivel global “funcionales”, “regresión”, “smoke”.

Para la construcción se usará el patrón de diseño, PageObject, y se usará la herramienta de automatización de la construcción, Gradle.

5.2.5. Identificación clases, métodos y funciones

A la hora de identificar los nombres de los objetos que componen el código, nos basaremos en el patrón de PageObject.

Identificador de la clase: Existe dos posibilidades, según el contexto:

- Pageobject: Nombre de la página o pantalla.
- Test: Nombre del caso de uso o algún que otro nombre que nos identifique el caso de uso.

Identificador de funciones o métodos: Nombre de la acción que va a realizar dentro de la página o pantalla.

Ejemplo de nombre de clase y métodos de la aplicación web SIAVT.

5.3. EJECUCIÓN DE PRUEBAS AUTOMÁTICAS

En esta fase se realiza la ejecución de las pruebas automáticas, aunque no excluye que se realicen pruebas manuales y pruebas exploratorias, pero no es el documento adecuado.

5.3.1. Baterías de pruebas

5.3.1.1 Pruebas funcionales

Corresponde a todas los casos de pruebas que han sido seleccionados para la realización de las pruebas automáticas, estos casos de pruebas deben haber sido dados de alta en Testlink y estar configurados de acuerdo al documento Automatización Pruebas Funcionales-Manual TestLink.

6. PAGEOBJECTS

Para seguir con el tema de las buenas prácticas, en esta entrada vamos a hablar de las buenas prácticas en cuanto al diseño de Test Automáticos con Selenium Webdriver

Para la realización de las pruebas funcionales con selenium, nos basaremos en uso de patrón de diseño, PageObject.

El uso de PageObject tiene varias ventajas:

- Mayor legibilidad, se representan solo lo qué queremos probar los test, ya que se separa lo que se **quiere** probar de **como** probarlo, y la lectura del código mejora.
- Mantenimiento del código de las pruebas, se mejor y se realiza más fácilmente, ya que si se produce cambios en las pantallas de la aplicación no tenemos que tocar las distintas pruebas, pueden ser muy numerosas en una pantalla determinada, sino solo hay que actualizar PageObject de la pantalla correspondiente.
- Evita la aparición de código duplicado, al tener las funciones o métodos que más se usan en un PageObject identificado.
- Actualización del Framework más fácil, si hay que actualizar el framework, en nuestro caso Selenium, por una versión más actual, solo tenemos que modificar la capa de PageObject, pero no es necesario actualizar ninguna prueba.
- Cambio o sustituir el Framework es más rápido, si en el futuro se toma la decisión de cambiar o sustituir el framework (Selenium) usado actualmente por otro (ejemplo: Serenity), este cambio tendría menos impacto y se podría realizar en un menor tiempo.

Es crear una capa de abstracción en los scripts de las pruebas en la que ocultamos la implementación interna de las pruebas. Y nos centramos en lo que realmente se tiene que probar.

6.1. USO DE PAGEOBJECTS

En general, un PageObject representa cada una de las pantallas que compone la aplicación que se desea probar. Las pantallas contienen una serie de objetos (elementos) que encapsulan las características y funcionalidades representadas en la página. Por lo que nos permite tener el código para interactuar con elementos de una página en cada uno de los PageObjects.

En otros casos, se encontrará que dentro de una página tenemos elementos que reutilizamos en otras partes, en este caso podemos construir un PageObject de ese elemento para poder reutilizarlo.

En este último caso, como ejemplo sería el menú de navegación el cual aparecería en muchas páginas de la aplicación, en su consiguiente duplicación en varios PageObjects. El objeto recibirá nuestro controlador del navegador y escondería su uso del código de la prueba. Evidentemente esto nos lleva a que en una página puede tener varios PageObject ya que contenga elementos significativos en el cual su uso es necesario separarlos.

Cuando creamos un PageObject, lo que se realiza es la creación de una capa de abstracción de lo que realmente es test, con las acciones que hay que realizar con los elementos que contiene la página. Por lo que si hay alguna modificación de la página de la aplicación solo afectaría al PageObject en cuestión y no a las pruebas en sí.

Un test nunca debe manipular directamente componentes de la página, si no que se delega manejo a través del ***PageObject*** correspondiente que como se ha indicado anteriormente representa la página. Lo que realmente se convierte es que el ***PageObject*** en una API con la que podemos encontrar y manipular los datos de la página.

Ilustración 7. PageObjects

6.2. EJEMPLO: APLICACIÓN WEB SIAVT

- PageObject:
  - LoginPage
    - Login
    - Logout
  - HomePage
    - BuscarAlertas
    - BarraNavegación
    - VerDetalle
  - ApremioPage
    - BuscarExpediente
- Test:
  - ConsultaExpApremio
    - LoginPage.login
    - HomePage.BarraNavegación.Apremio.Expediente

- ApremioPage.BuscarExpediente
- LoginPage.logout

7. HERRAMIENTAS

7.1.1. Selenium Client & WebDriver

selenium-java-3.14.0

7.1.2. Servidor Selenium

Selenium Server: v3.14.0

Geckodriver: v0.21.0-linux64

chromedriver: v2.41-linux64

7.1.3. Navegadores

Corresponden a los navegadores instalados en el servidor de Selenium.

firefox: 60.1.0-4.el7_5

google-chrome-stable: 68.0.3440.106-1