



**Treball final de grau**

**GRAU D'ENGINYERIA INFORMÀTICA**

**Facultat de Matemàtiques  
Universitat de Barcelona**

---

**INFRAESTRUCTURA PARA LA  
AUTOMATIZACIÓN DE PROCESOS  
DE SOFTWARE  
ENFOQUE SOBRE TESTING Y CALIDAD**

---

Autor: Cristhian Carmona T

Director: Lluís Garrido

Realitzat a: Departament de Matemàtica Aplicada i Anàlisi

Barcelona: 28 de enero de 2016



### ***Abstract***

*The current project is aimed to build an infrastructure to generate automatized software processes focused on web application testing.*

*An infrastructure has been designed to be easily adapted to settings for Agile developing. It will be subject to an exhaustive set of automatized testing before the release of an integration or deployment.*

*In order to achieve the maximum efficiency, the root issue has been studied, adopting a new working procedure that includes every stage of the software's life cycle. Furthermore, to obtain a significant reduction in the cost of buying software licences, OpenSource tools have been chosen.*

*Finally, this infrastructure will have the necessary mechanisms to increase the quality and to reduce the delivery time of a software product to the client.*

### **Resumen**

El presente proyecto pretende crear una infraestructura para la automatización de procesos de software con enfoque sobre el testing de aplicaciones web.

Se ha diseñado la infraestructura para que se adapte fácilmente a entornos de desarrollo ágil y para que someta la aplicación a constantes baterías de pruebas automáticas antes de que se realice una integración o despliegue.

Para conseguir al máximo su eficiencia se ha atacado el problema de raíz, adoptando una nueva metodología de trabajo que engloba todas las etapas del ciclo de vida del software, además para conseguir una reducción considerable de costes que conlleva la compra de licencias de software, se ha elegido herramientas OpenSource.

Al final, la infraestructura proveerá de los mecanismos necesarios para aumentar la calidad y reducir los tiempos de entrega de un producto de software al cliente.

## Agradecimientos

Quiero agradecer de manera infinita el apoyo incondicional de *mis padres y de mi hermana*, por saber aguantarme y pese a ello nunca dejar de darme ánimos, estoy seguro que de no ser por ellos muy posiblemente no habría redactando este informe.

A mis primos y demás familiares que tampoco han dejado nunca de animarme.

Al resto de mis amigos, especialmente a mi amigo *Augusto Príncipe* que pese a la distancia y sus obligaciones ha contribuido sustancialmente a mejorar la calidad de este documento.

Al resto de compañeros y colegas de la facultad por esas inolvidables experiencias.

Al conjunto entero de profesores por haberme enriquecido con sus conocimientos, sin duda la culminación de este proyecto es fruto de ello.

Finalmente a mi tutor, el *Dr. Lluís Garrido* por su profesionalidad y por confiar en mí para este reto.

## Índice

<b>1</b>	<b>INTRODUCCIÓN</b>	<b>7</b>
1.1	ANTECEDENTES	8
1.2	MOTIVACIÓN	9
1.3	DEFINICIÓN DE OBJETIVOS	10
1.4	ESTRUCTURA DEL DOCUMENTO	11
<b>2</b>	<b>DESCRIPCIÓN DEL PROBLEMA</b>	<b>12</b>
<b>3</b>	<b>DESCRIPCIÓN DE LA SOLUCIÓN</b>	<b>13</b>
3.1	QUÉ ES LA AUTOMATIZACIÓN DE TESTS DE SOFTWARE	13
3.2	POR QUÉ AUTOMATIZAR	13
3.3	AUTOMATIZACIÓN PARA APLICACIONES WEB	14
3.4	VENTAJAS Y DESVENTAJAS DE AUTOMATIZAR	15
3.5	VENTAJAS DEL TESTING AUTOMÁTICO RESPECTO AL TESTING MANUAL	16
3.6	SELECCIÓN DEL ENFOQUE PARA LA AUTOMATIZACIÓN	17
3.6.1	GRABACIÓN DE PRUEBAS	17
3.6.2	DESARROLLO GUIADO POR COMPORTAMIENTO	17
<b>4</b>	<b>DESARROLLO DEL PROYECTO</b>	<b>18</b>
4.1	PROCESO DE AUTOMATIZACIÓN	18
4.1.1	PRUEBA DE ANÁLISIS DE VIABILIDAD DE LA AUTOMATIZACIÓN	20
4.1.2	SELECCIÓN DE HERRAMIENTAS ADECUADAS	23
4.1.3	EVALUACIÓN DE FRAMEWORKS ADECUADOS	29
4.1.4	ELABORACIÓN DE PRUEBA DE CONCEPTO	38
4.1.5	DESARROLLO DE UN FRAMEWORK PARA LA AUTOMATIZACIÓN	38
4.1.6	DESARROLLAR CÓDIGO PARA TESTING, EJECUTARLO Y ANALIZARLO	41
4.2	AUTOMATIZANDO PRUEBAS CON INTEGRACIÓN CONTINUA	52
<b>5</b>	<b>PRUEBAS Y RESULTADOS</b>	<b>55</b>
5.1	RESULTADOS DE TESTS MANUALES	55
5.2	RESULTADOS DE TESTS AUTOMÁTICOS	55
5.3	ANÁLISIS DE RESULTADOS TESTS MANUALES VERSUS TESTS AUTOMÁTICOS	56
<b>6</b>	<b>CONCLUSIONES</b>	<b>57</b>
<b>7</b>	<b>CONTRIBUCIONES</b>	<b>58</b>
<b>8</b>	<b>ANEXOS</b>	<b>59</b>
8.1	CAPTURAS DE PANTALLA	59
8.2	VERSIONES DE LAS HERRAMIENTAS AQUÍ UTILIZADAS	60



# 1 Introducción

Actualmente donde el factor tiempo es equivalente a dinero, donde la sociedad demanda productos y soluciones continuamente, donde para su fabricación intervienen equipos de varios países, ha requerido que la industria adopte soluciones cada vez más ágiles e innovadoras.

Muchas soluciones que se han aportado a lo largo del desarrollo tecnológico vienen de la confluencia de varias metodologías y tecnologías haciendo que no solo evolucione mas rápido sino que sigan unos rigurosos estándares de calidad. Los productos basados en software no son la excepción a estos cambios.

Desde los inicios del desarrollo del software varias han sido las metodologías que se han adoptado, cada una con su propio enfoque, donde la pionera en tomar fuerza allá por los 70's fue la de desarrollo en cascada, convirtiéndose en la clásica por excelencia.

Años más tarde aparecieron otras con un enfoque incremental, donde cada entrega debe ser construida en un corto plazo de tiempo obligando que en todas las etapas se optimice al máximo el tiempo.

A día de hoy el objetivo de los equipos de desarrollo ágil [1] es generar bloques de software de forma iterativa cada dos o tres semanas. En cada iteración se llevan a cabo principalmente las siguientes etapas: planificación, codificación, testing y entrega de un bloque de software totalmente funcional. Los nuevos bloques de software deben integrarse con los anteriores sin que los anteriores se vean afectados por los nuevos.

Si nos centramos en la etapa de testing de una iteración significaría testear todas las funcionalidades que se hayan implementado además de testear las anteriores para comprobar que la nueva no ha afectado al resto, lo cual conlleva un esfuerzo considerable.

El presente proyecto pretende aportar una solución tecnológica a este tipo de problemas mediante la instalación y configuración de diversas herramientas que permiten al equipo de test la capacidad de desarrollar pruebas automatizadas sobre aplicaciones web, además de brindar medios para automatizar procesos que actúan sobre todo el ciclo de vida del software.

En definitiva, se busca reducir tiempo y recursos a procesos repetitivos evitando al máximo la intervención humana, efectuando tests más efectivos para aumentar la calidad de un producto, y reducir los tiempos de entrega.

## 1.1 Antecedentes

Resulta complicado afirmar que la calidad del software es clave para alcanzar el éxito en un proyecto, pero lo que sí está claro que lo es para el fracaso, más cuando en la actualidad productos y servicios basados en software están formados por entornos de trabajo complejos con múltiples equipos que son muy susceptibles a errores.

Para lograrlo se hace indispensable plantearse la calidad desde un punto de vista global, que abarque el ciclo de vida completo que permita desarrollar y poner en producción un producto de calidad. De esta forma, los errores y costes serán menores y, la agilidad y eficacia crecerán.

Siendo la etapa de testing una de las que mayor valor aporta a un producto resulta curioso que entre la comunidad de informáticos se subestime dicha etapa. Con este proyecto se pretende mayor notoriedad a la etapa de testing aportando valor mediante la automatización de test.



## 1.2 Motivación

- **Inversión a medio-largo plazo**  
La consideramos una inversión a medio-largo plazo puesto que al principio hay un tiempo de aprendizaje, mientras que los resultados se verán a medida que se incremente el número de entregas del producto.
- **Reducir tiempo y recursos a procesos repetitivos**  
Conforme avancen las entregas el número de test se irá incrementando, cuando los tests automáticos entren en juego las pruebas manuales se reducirán considerablemente sin la necesidad de gastar tiempo y recursos innecesarios.
- **Mejorar la organización y comunicación del equipo**  
El establecimiento de una metodología de trabajo conlleva a que todos los miembros del equipo sigan una misma disciplina y que todos hablen el mismo lenguaje, creando un ambiente enriquecedor.
- **Aumentar la confianza del equipo**  
La ejecución de pruebas y demás procesos de forma automática, con poca intervención humana, posibilita que la cantidad de errores de forma manual se reduzcan para poner énfasis en el buen desempeño de los equipos de desarrollo y testing.
- **Aportar una solución global al ciclo de vida del software**  
No sólo se aporta una solución a problemas en la etapa de testing sino que mediante la implementación de 'Integración Continua' todo el ciclo de vida del software se verá mejorado con la automatización de compilaciones y despliegues.

### 1.3 Definición de objetivos

El objetivo principal del presente proyecto es implantar una infraestructura para automatizar procesos de software, dando más protagonismo a la etapa de testing.

- Adoptar una metodología de desarrollo para optimizar tiempos
- Implementar un framework propio para test automatizados
- Reducir los costes de recursos y software destinados al testing
- Disponer de mecanismos para automatizar otros procesos
- Reducir el tiempo de pruebas automatizadas ahorran tiempo

## 1.4 Estructura del documento

Con el objetivo de que sea más fácil la comprensión de todo el documento, éste se ha dividido principalmente en cuatro grandes etapas. A continuación una breve descripción de cada una de ellas:

1. **Descripción del problema:** se explicará brevemente quién tiene actualmente este tipo de problemas, equipos de desarrollo, que pueden formar de una empresa o trabajar de forma independiente
2. **Descripción de la solución:** se propondrá una solución a los problemas que antes hemos mencionado, destacando sus principales beneficios luego de la implementación.
3. **Desarrollo del proyecto:** esta etapa es el núcleo del proyecto, pues aquí se seleccionan todas las herramientas, se crea el entorno de desarrollo para desarrollar nuestro propio framework, es decir, todo el proceso de automatización de pruebas se reparte en varias fases.

Además de toda la implementación de la infraestructura, para probarla crearemos una batería de pruebas automáticas mediante un ciclo de vida de desarrollo ágil.

4. **Escalado a Integración Continua:** con toda la plataforma funcionando toca escalar al sistema de Integración Continua para disponer de mecanismos que automaticen ciertas tareas que antes se realizaban de forma manual

## 2 Descripción del problema

El proyecto que aquí se plantea pretende dar solución principalmente a dos tipos de problemas que actualmente se presentan en equipos de desarrollo de aplicaciones, aunque también se puede extrapolar a otro tipo de productos que compartan la misma filosofía:

- Equipos de desarrollo en constante crecimiento, que sólo disponen de un equipo de pruebas manuales y, que quieran potenciar la calidad de sus productos reduciendo costes de pruebas manuales y tiempo de entrega final.
- Incluso equipos de desarrollo que partan desde cero el desarrollo de una aplicación web y quieran realizar un despliegue en un corto plazo.

En el primer caso se presentan algunos problemas:

1. Equipo de pruebas insuficiente para la gran carga de trabajo
2. Fatiga laboral en los testers al realizar la misma tarea repetidas veces
3. Entrega de productos con alto porcentaje de errores
4. Testers no cualificados para pruebas de bajo nivel
5. Intervención manual en todas las etapas del ciclo de vida

En el segundo caso:

1. No personal cualificado para realizar pruebas a aplicación con lógica compleja
2. Plazo muy corto de entrega
3. Futuras aplicaciones de similar complejidad

En ambos casos, la solución pasa por implantar una infraestructura capaz de realizar pruebas automáticas, reduciendo al mínimo la intervención de personas en estas tareas.

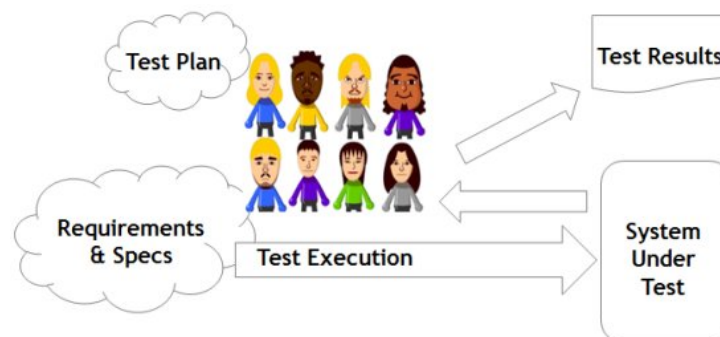


Imagen 1. Situación de testing manual [2]

## 3 Descripción de la solución

### 3.1 Qué es la automatización de tests de software

La automatización de pruebas consiste en el uso de software especial para controlar la ejecución de pruebas y la comparación de resultados obtenidos y resultados esperados. La automatización permite incluir pruebas repetitivas dentro de un proceso formal de pruebas ya existente, o bien adicionar pruebas cuya ejecución manual resultaría difícil.

### 3.2 Por qué automatizar

La principal razón es el tiempo y con las pruebas automatizadas se puede reducir el tiempo de las pruebas. Además al automatizar las actividades comunes que no requieren de inteligencia humana, los testers reales pueden dedicar mayor tiempo a pruebas más críticas y caminos más elaborados dejando los caminos singulares a las pruebas automatizadas.

Muchos fallos de software pueden costar millones e incluso billones de euros y en algunos casos hasta puede el cese de su actividad.

*Si un software no funciona, no importa cómo de rápido o barato lo vendamos.*

La automatización de pruebas de software proveen tres beneficios clave:

1. Cobertura acumulada para detectar errores y reducir los costos de fallos
2. Repetitividad para ahorrar tiempo y reducir el coste para comercializarlo
3. Aprovechamiento para mejorar la productividad

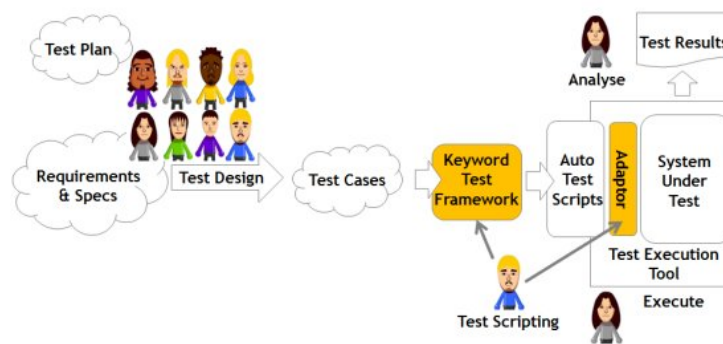


Imagen 2. Situación con Tests Automatizados

### 3.3 Automatización para aplicaciones web

Hoy en día la mayoría de aplicaciones son aplicaciones Web. Los días donde las personas usaban herramientas de escritorio para desempeñar sus tareas están desapareciendo.

En el gráfico de abajo podemos observar la oportunidad de negocio entre tecnologías que existe hoy en día. Las aplicaciones web presentan la particularidad de disponer de dos factores fundamentales, su agilidad para el despliegue y una gran capacidad para saber adaptarse a los gustos de los usuarios.

Mientras que las aplicaciones de escritorio que hasta hace poco tiempo gozaban de casi todo el mercado, no tienen la capacidad de hacer un despliegue ágil en comparación a otras tecnologías, este factor con el nivel de competitividad donde predomina el que primero coloque su producto resulta una seria desventaja frente a la competencia.

Otra gran oportunidad de negocio son las aplicaciones que aún se desarrollan con tecnología web 2.0 y aplicaciones destinadas a dispositivos móviles. Estas últimas en pleno auge.

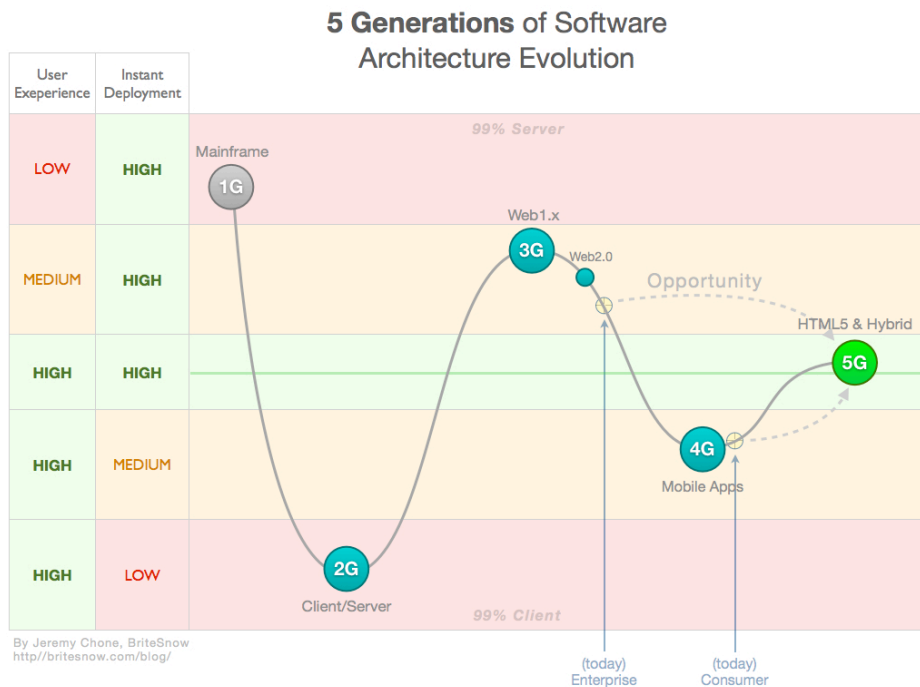


Imagen 3. Estadística de desarrollo de aplicaciones [2]

### 3.4 Ventajas y desventajas de automatizar

#### Ventajas:

- Útil cuando se necesita repetir una prueba
- Útil en pruebas de regresión
- Es usado para ejecutar 'la compatibilidad de pruebas' de una aplicación
- Pruebas rápidas en códigos diferentes
- Pueden correr en diferentes máquinas de forma simultánea
- A la larga los costes se pueden reducir
- No requiere mano de obra
- Confiable, reusable y rápido
- Pueden ejecutar casos de prueba que son posible ejecutarlos de forma manual

#### Desventajas:

- Dificultad de mantener el código desarrollado
- Resulta cara como inversión inicial
- Algunas herramientas pueden no garantizar la buena calidad
- No es posible automatizar todo
- Los pobres casos de prueba pueden no encontrar defectos
- Las herramientas no pueden adivinar o calcular donde están los defectos. Solo se ejecuta los que está codificado
- Los testers necesitan mucho tiempo para diseñar casos de prueba

### 3.5 Ventajas del testing automático respecto al testing manual

Se considera prueba manual a todo proceso mediante el cual los desarrolladores de software ejecutan pruebas manualmente, comparando las expectativas y los resultados actuales con el fin de encontrar defectos de software. [4]

Realizar pruebas manuales es una habilidad. La automatización de pruebas es otra habilidad en un contexto diferente. Algunas ventajas de éstas sobre las pruebas manuales son:

- Existen ciertas tareas difíciles de realizar manualmente, por ejemplo, pruebas de regresión de bajo nivel de una interfaz, tarea que es muy difícil y donde es fácil cometer un error. Mientras que si configuramos correctamente un conjunto de pruebas automáticas para que se ejecuten será mucho más fácil encontrar errores.
- Las pruebas manuales pueden ser repetitivas y aburridas, nadie quiere rellenar el mismo tipo de documentos una y otra vez. Las pruebas automáticas son ideales para este tipo de tareas.
- Las pruebas manuales no se puede reutilizar, mientras que las automáticas si se añade algo a la aplicación se pueden volver a reutilizar las veces que sean necesarias de forma instantánea.



## 3.6 Selección del enfoque para la automatización

### 3.6.1 Grabación de pruebas

El enfoque de la captura o grabación de pruebas significa que las pruebas son ejecutadas manualmente, mientras las entradas y salidas son capturadas en segundo plano. Esto lo hace Selenium IDE mediante un plugin propio.

Las razones por las que **se descarta** este enfoque para la automatización de pruebas es porque es una herramienta con la que no se puede reutilizar nada, su ejecución es lenta, todas las pruebas han de ser locales, nada en remoto y sólo se puede lanzar con Firefox.

### 3.6.2 Desarrollo guiado por comportamiento

**(Behaviour Development Driven, en inglés)** en adelante BDD [4]

Es un conjunto de prácticas que buscan mejorar la comunicación de los requerimientos y expresar mediante ejemplos el diseño de un software.

Da soporte directo a los valores ágiles de *colaboración con el cliente* y *respuesta al cambio*, ya que ayuda a crear un lenguaje común entre el equipo que desarrolla y las personas que definen los requerimientos del negocio, permitiendo al equipo enfocarse en el desarrollo de software que verdaderamente cumpla las necesidades del cliente, y quedando las especificaciones en formato ejecutable, lo que resulta en una matriz de pruebas automatizadas que ayudan a detectar regresiones y fallas

Al final, permite al equipo escribir código limpio y adaptable a cambios en el tiempo.

## 4 Desarrollo del proyecto

### 4.1 Proceso de automatización

Para llevar a cabo la implementación de toda la infraestructura para la automatización de pruebas seguiremos las siguientes fases, correspondiendo cada una a una actividad particular que a su vez generan resultados para la siguiente. [6]

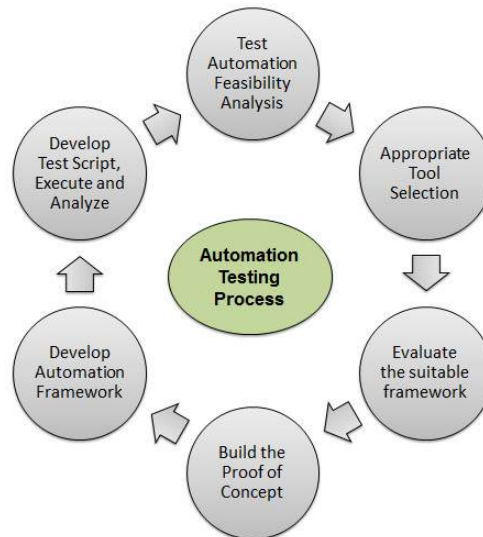


Imagen 4. Proceso de la Automatización de tests [7]

#### 1. PRUEBA DE ANÁLISIS DE VIABILIDAD DE LA AUTOMATIZACIÓN

El primer paso es verificar si la aplicación puede ser automatizada o no. No todas las aplicaciones pueden ser automatizadas debido a sus limitaciones.

#### 2. SELECCIÓN DE HERRAMIENTAS APROPIADAS

El paso más importante es la selección de las herramientas. Depende de la tecnología que se ha desarrollado la aplicación, características y sus usos.

#### 3. EVALUAR FRAMEWORK ADECUADO

De las herramientas seleccionadas la siguiente etapa es seleccionar un framework orientado al testing de aplicaciones web. Pueden haber algunos que no se ajusten a nuestras exigencias y lo mejor será descartarlos.

La creación del entorno de desarrollo para el equipo de testing queda implícita en esta etapa.

#### **4. CONSTRUIR LA PRUEBA DE CONCEPTO**

(Proof Of Concept POC, en inglés) En esta etapa haremos pruebas del entorno de desarrollo, es decir de las herramientas y los framework que lo integran, directamente sobre la aplicación web con el propósito de verificar que el concepto o teoría es susceptible de ser explotada de manera útil.

#### **5. DESARROLLAR UN FRAMEWORK PARA AUTOMATIZACIÓN**

Después de construir la POC el desarrollo del framework es llevado a cabo usando todas las herramientas que hemos elegido. Es diseño de éste debe realizarse con especial cuidado, probando su correcta integración entre ellas.

Para la creación del framework se creará un conjunto de clases Python y se explicará paso a paso el método de desarrollo de cómo implementar los scripts que la contienen.

#### **6. DESARROLLAR CÓDIGO PARA TESTING, EJECUTARLOS Y ANALIZARLO**

Una vez el desarrollo de los scripts se haya terminado, serán ejecutados, los resultados serán analizados y los defectos escritos en ficheros log, dado el caso.

Esta etapa se ha desarrollado completamente sobre el ciclo de vida de software de desarrollo ágil. Consta de varias fases, desde la toma de requerimientos hasta el análisis de resultados. [4]

### 4.1.1 Prueba de análisis de viabilidad de la automatización

Antes de empezar a automatizar es conveniente conocer algunos factores que pueden acarrear algunos problemas y otros en los que directamente se descarta este tipo de tareas.

#### 4.1.1.1 Factores que afectan al testing de aplicaciones web

- **Numerosos usos de aplicaciones con caminos de entrada y salida son posibles.** Debido al diseño y a la propia naturaleza de la aplicación web es posible que diferentes usuarios usen rutas en la aplicación.

Por ejemplo, en una aplicación de banca online un usuario puede consultar su saldo disponible en varias páginas. Todas estas permutaciones y combinaciones necesitan ser probadas a fondo.

- **Personas de diferentes orígenes y habilidades técnicas pueden usar la aplicación** No todos los usuarios tienen el mismo nivel de conocimientos por lo que muchos pueden encontrar la aplicación difícil de usar.

Por ejemplo, aplicaciones orientadas al Business Intelligence capaces de generar informes complejos pueden funcionar para ciertos usuarios y para otros no.

- **Aplicaciones de Intranet versus aplicaciones basadas en internet.** Las aplicaciones basadas en intranet generalmente atienden a un grupo de usuarios controlados. Los desarrolladores y arquitectos por lo general asumen que las personas que acceden a esta cumplen con las especificaciones técnicas, software o hardware en sus máquinas.

Mientras que en las aplicaciones basadas en internet los usuarios muchas veces necesitan autenticarse y cumplir con medidas de seguridad más restrictivas.

- **Los usuarios finales pueden usar diferentes tipos de navegadores.** Normalmente los usuarios de aplicaciones basadas en internet pueden acceder a la aplicación desde diferentes navegadores.

Si testeamos la aplicación en un solo navegador no podemos asegurar que funcione para el resto de navegadores, porque no sólo

muestran la página sino que además tiene varios niveles de soporte del lado del cliente.

- **Velocidad de la red.** Una baja velocidad de la red puede causar que varios componentes tarden en descargarse pudiendo ocasionar cualquier tipo de error. Las aplicaciones basadas en internet necesitan considerar seriamente este tipo de factor para procesos de testing.
- **Aspectos de seguridad.** Si la aplicación captura cierto tipo de información personal o sensible, puede ser crucial para testear la fortaleza de la aplicación. Se necesita tener un cuidado especial para que la información no se vea comprometida.

#### 4.1.1.2 Cuando no automatizar

En la automatización de pruebas se requiere que el comportamiento esperado de la aplicación se conozca. Esta premisa se considera como la piedra angular en la automatización.

Algunos puntos a considerar cuando se pretende automatizar:

- **Diseño inestable.** Hay ciertas aplicaciones que son inherentemente inestables por su diseño. Por ejemplo, un sistema que mapea el tiempo u otra que muestra datos en tiempo real no mostrará resultados lo suficientemente predecibles para la automatización.
- **Testers inexperimentados.** Si las personas que automatizan las pruebas no están lo suficientemente familiarizadas con la aplicación como para saber el comportamiento, automatizar estas pruebas puede generar dudas sobre su valor. Un test automático es igual de bueno que la persona que lo creó.
- **Testers temporales.** En algunos casos el equipo de pruebas puede verse apoyado por personal ajeno de otras áreas y que no se hayan involucrado el tiempo suficiente como para conocer bien el área o simplemente que no estuvieran desde el inicio es razón de peso a valorar.
- **Insuficiente tiempo y recursos.** Si no dispones del tiempo y recursos para ejecutar pruebas de forma manual, no esperes que las herramientas te ayuden. La inversión inicial para planificar,

entrenar e implementarlos tomará más tiempo a corto plazo de que lo que cualquier herramienta te pueda ayudar.

Cabe remarcar que la automatización es una solución estratégica, no un parche a corto plazo.

## 4.1.2 Selección de herramientas adecuadas

### 4.1.2.1 Principales lenguajes para desarrollar aplicaciones web

Conocer los principales lenguajes de programación ayudará a determinar el grado de dificultad que puede acarrear las pruebas automatizadas. Algunos de los más populares son, JavaScript, Java, Python, CSS, PHP, Ruby, C++, etc. [8]

Veamos algunas características más relevantes de los tres lenguajes más usados:

- **JavaScript** es uno de los más populares y dinámicos lenguajes de programación usados para crear y desarrollar aplicaciones Web. Este lenguaje es capaz de controlar el comportamiento del navegador, editar contenido de documentos que han sido mostrados, usado principalmente del lado del cliente y permite la comunicación asíncrona.

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Actualmente es ampliamente utilizado para enviar y recibir información desde el servidor junto con la ayuda de otras tecnologías de AJAX.

- **AJAX** es una técnica de desarrollo web para crear aplicaciones interactivas. Como característica principal, permite realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.
- **Java** es otro lenguaje ampliamente usado para el desarrollo Web. Inicialmente fue desarrollado para la televisión interactiva, hoy en día, de hecho es el lenguaje de mayor demanda como plataforma estándar por empresas y desarrolladores de juegos y móviles de todo el mundo.
- **Python** es un lenguaje altamente usado. Un desarrollador puede escribir código y ejecutarlo sin necesidad de un compilador. Muchas aplicaciones móviles desarrolladas en Python son Rdio, Instagram, and Pinterest. Y con ejemplos de aplicaciones web con soporte de Python tenemos a Django, Google, Nasa, y Yahoo.

Como lenguajes de desarrollo web más básicos tenemos a HTML y CSS. El primero como estándar que permite estructurar y formatear contenido de una web, mientras que CSS permite al desarrollador definir la apariencia y formato de múltiples páginas web a la vez.

La aplicación de lenguajes como JavaScript o cualquier otro con posibilidad de alterar el comportamiento de un navegador debe ser considerado seriamente, muchos de estos comportamientos pueden generar interacciones o animaciones que acarrearán tiempos de espera que, en caso de no ser considerados previamente generarán una reacción en cadena de retrasos en la entrega de un paquete de software.

Factores externos como la velocidad de la red o el rendimiento del hardware del equipo también pueden generar tiempos de espera, no sólo el tipo de navegador.

#### 4.1.2.1.1 Herramientas de soporte

Algunas de las herramientas que se han utilizado de forma constante a lo largo del desarrollo del proyecto:

- **Firefox**, es un navegador web libre y de código abierto desarrollado para la gran mayoría de sistemas operativos modernos. Recientes estudios lo consideran el navegador más personalizable y seguro del momento.  
Para los desarrolladores web posee un importante repertorio de herramientas incorporadas.
- **Firebug**, es un plugin gratuito creado y diseñado especialmente para desarrolladores y programadores web. Permite realizar gran número de acciones sobre el código fuente de la web de manera fácil y rápida. Su uso principalmente será para localizar elementos web en la aplicación web.
- **Terminal**, es un programa informático donde el usuario interactúa con el sistema operativo mediante una ventana que espera órdenes escritas por el usuario. 'A grosso modo' nos permite tener un mayor control del equipo. Las más potentes vienen integradas en sistemas operativos Linux y Mac OSX.
- **SourceTree**, es uno de los mejores clientes GUI para manejar repositorios Git. Básicamente convierte operaciones complejas en algo sencillo sin perder ni un ápice de potencia y flexibilidad. Alternativamente se pueden usar otros clientes si se trabaja en Windows o Linux.
- **Virtualenv**, es una herramienta para crear entornos virtuales aislados de Python localmente para el usuario. Para entendernos, si necesitamos instalar diferentes versiones de un mismo paquete y no



queremos dañar la instalación original sólo necesitamos crear un entorno independiente donde haremos las nuevas instalaciones.

En caso de que algo vaya mal lo único que tenemos que hacer es eliminar el entorno y listo, seguiremos con nuestro entorno original intacto.

- **PIP**, es una herramienta de Python que nos permite instalar paquetes externos desde su repositorio llamado PyPi, incluso los que no están en este. Solo es necesario ejecutar un comando en la terminal para instalar cualquier paquete que necesitemos.

#### 4.1.2.1.2 Herramientas de programación

**Python**, para el desarrollo del código de test se elige este lenguaje por cuatro razones principales: [9]

- i) Fácil de aprender para los programadores
- ii) Lenguaje interpretado. Su sintaxis favorece un código legible
- iii) Multiparadigma. Soporta programación orientada a objetos, imperativa y funcional
- iv) Sólo necesita de un compilador para su ejecución

Como alternativa, Java también presenta una gran cantidad de recursos dedicados a test automáticos. Si se desea se puede optar por esta opción.

#### 4.1.2.1.3 Herramientas de edición

- **Robot Framework IDE**, es un entorno de desarrollo para casos de pruebas del Robot Framework. Su interfaz de usuario presenta una apariencia amigable y sencilla especialmente diseñada para el diseño de casos de pruebas.
- **PyCharm**, es un IDE especialmente diseñado para la programación en Python, consta de varios plugins, es multiplataforma (corre sobre sistemas como Windows, Mac OSX, y Linux). Tiene dos versiones disponibles, la gratuita y la profesional que es de pago.

#### 4.1.2.1.4 Sistemas de control de versiones de ficheros

Dada la gran cantidad de ficheros y directorios que pueden ser generados y accesibles por todo el equipo de testing, necesitamos un sistema capaz de gestionarlos de manera ágil y lo mas seguro posible. Algunos mecanismo que nos aportan estos sistemas:

- Almacenar versiones distintas de un fichero, para que cualquiera de ellos puedan recuperarse en cualquier momento
- Definir diferentes ramas de desarrollo, para que los programadores puedan generar versiones nuevas independientes de cada rama
- Automatizar la ‘fusión’ (merge, en inglés) de ficheros de versiones o ramas diferentes, de este modo, la integración del código de cada programador sobre el mismo fichero será, en la medida de lo posible, automática
- Facilita la Integración Continua

**GitHub**, Distribuida, permite trabajar offline, basada en modelo fácil de colaboración, posibilidad de crear y enlazar tantos repositorios como queramos, fácil instalación en servidor, por el contrario es un sistema algo más complejo y difícil de aprender.

El IDE SourceTree ayuda a que la gestión de repositorios GitHub sea más sencilla.

#### **4.1.2.1.5 Sistemas para la [11]ción Continua**

##### **4.1.2.1.5.1 Qué es la Integración Continua**

El proceso de integración continua tiene como objetivo principal comprobar que cada actualización del código fuente no genere problemas en una aplicación que se está desarrollando. La integración continua fue utilizada por IBM para el desarrollo del OS/360 en los años 60.

La integración continua no es una herramienta sino mas bien una práctica salida del eXtreme Programming (XP)

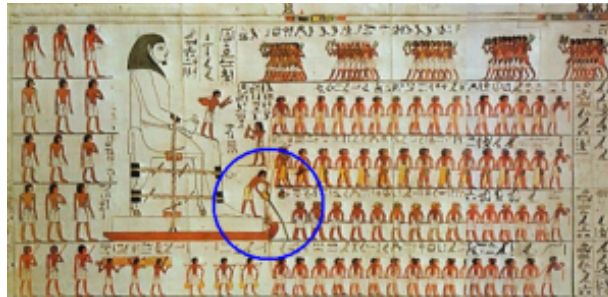


Imagen 5. Integración Continua en el antiguo Egipto

#### 4.1.2.1.5.2 Cómo funciona

Los miembros de un equipo de desarrollo integran el programa sobre el que trabajan con frecuencia. La integración continua consiste en activar en cada integración un proceso basado en una plataforma que verifica automáticamente el funcionamiento de la aplicación para que las anomalías sean detectadas lo más pronto posible. [10]

Lo más difícil para el desarrollador es conocer el impacto real de una actualización fundamental sobre todas las funcionalidades de la aplicación. La integración continua permite al desarrollador tener esta visión más global de la aplicación ya que las pruebas de la aplicación se hacen sobre un entorno similar de producción.

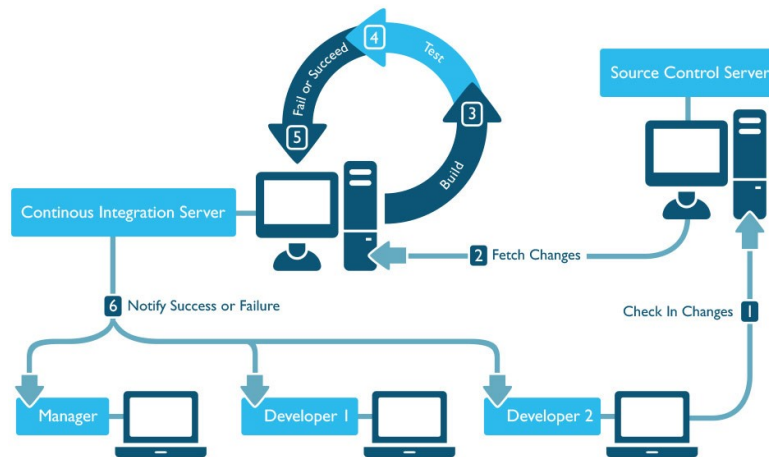


Imagen 6. Diagrama de Integración Continua

**Travis-CI**, es un servicio que ofrece IC disponible en dos versiones, cloud y server. La cloud requiere mucho menos trabajo que una instancia local, por ello resulta ideal su aplicación en esta infraestructura. [11]

Travis solo trabaja con repositorios Git, por lo que su integración con esta plataforma es bastante simple. En cuanto un usuario se registra ya es capaz de enlazar repositorios y poder configurar sus tareas.

Esta diferencia de otros sistemas IC no necesita saber qué ejecutar, es lo suficientemente inteligente como para averiguarlo por sí mismo, tan solo necesita que le especifiquemos las cosas que tiene que hacer en un solo fichero, el `.travis.yml`.

En comparación a otros sistemas que se dividen en varios ficheros de configuración repartidos por varios directorios, Travis-ci guarda este fichero en el mismo directorio del código fuente.

Travis, te permite conectar un repositorio y probar después de cada push que un desarrollador haga. Soporta múltiples lenguajes como PHP, Ruby, Node.js, entre otros. De este modo, podemos probar nuestras aplicaciones o librerías contra distintas configuraciones sin tener que tenerlas instaladas localmente. Tienen varias máquinas virtuales preparadas para cada combinación.

### 4.1.3 Evaluación de Frameworks adecuados

#### 4.1.3.1 Frameworks de testing

- **Selenium**, es un framework para prueba de aplicaciones Web que permite tests en muchos lenguajes de programación como Java, C#, Groovy, Perl, PHP, Python, Ruby. Disponible para varios sistemas operativos: Windows, Linux y Mac OS.
- **Selenium2Library**, es una librería de pruebas para aplicaciones web desarrollada por y para Robot Framework. Se la puede usar con la idea de disponer más funcionalidades ya implementadas. La idea de funcionamiento es la misma que su predecesora, Selenium.
- **Selenium WebDriver**, básicamente es el sucesor de Selenium. Aporta nuevas funcionalidades y mejoras. De cara al desarrollo de las librerías que implementaremos más adelante la API que aquí se usará será esta.
- **Robot Framework**, es un framework genérico de test automatizados para pruebas de aceptación y Acceptance Test Driven Development. De uso muy sencillo.

De serie viene con varias librerías formando un potente ecosistema dedicado a pruebas automatizadas. En sus últimas versiones ya se ha adaptado para pruebas basadas en metodología Behaviour Driven Development.

Todos los Frameworks aquí mencionados son Open Source.

#### 4.1.3.2 Funcionamiento del framework Selenium WebDriver

Para entender su funcionamiento usaremos dos vías, la fácil y la técnica [11]:

**Vía fácil**, para la explicación de esta vía nos serviremos de la analogía de la 'conducción de un taxi'. En esta actividad hay tres tipos de actores:

1. **Cliente**, le dice al taxista donde quiere ir y cómo llegar
2. **Taxista**, ejecuta la solicitud del cliente y envía la solicitud al coche

### 3. **Coche**, ejecuta la solicitud hecha por el taxista

El cliente llega al destino mediante los diálogos que suceden entre cliente – taxista y taxista – coche.

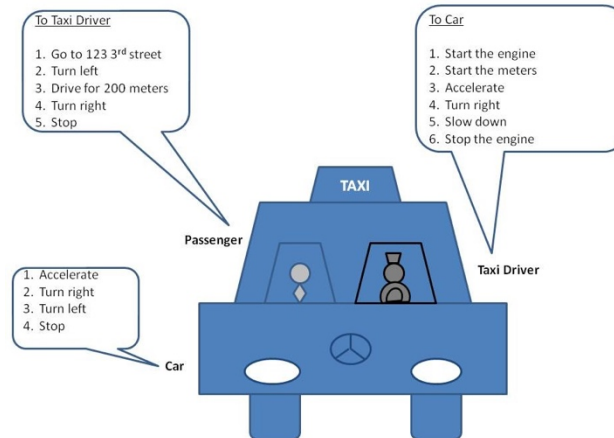


Imagen 7. Analogía del taxista

En la automatización de pruebas con Selenium Webdriver (y otras herramientas) también hay tres actores que intervienen:

1. **Ingeniero de pruebas que escribe código de automatización.**- el código de automatización envía las solicitudes al controlador del navegador
2. **El controlador del navegador** (driver, en inglés).- ejecuta las solicitudes del ingeniero de pruebas y envía sus propias solicitudes al navegador
3. **El navegador** (browser, en inglés).- ejecuta las solicitudes del controlador del navegador

Así, esta es la analogía:

1. El ingeniero de pruebas es como el cliente del taxi
2. El controlador del navegador es como el taxista
3. El navegador es como un taxi.

**La explicación técnica no usa analogías.** Cuando el código es ejecutado sigue los siguientes pasos:

1. Para cada instrucción Selenium, una petición es creada y la envía al controlador del navegador
2. El controlador del navegador usa u servidor HTTP para recibir las peticiones HTTP
3. El servidor HTTP determina los pasos necesarios para implementar una instrucción Selenium
4. La pasos implementados son ejecutados en el navegador
5. El estado de la ejecución es enviado de vuelta al servidor http
6. El servidor HTTP envía el estado de vuelta al código de automatización

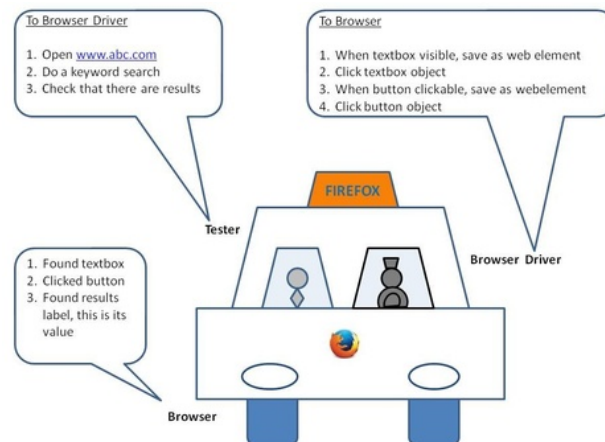


Imagen 8. Analogía del taxista en versión técnica

#### 4.1.3.3 Instalación y configuración del entorno local

Todo el desarrollo de esta infraestructura se ha hecho con *Mac OSX 'El Capitán'*. Ver anexo en caso de querer instalar sobre máquinas Linux.

Como prerequisite fundamental necesitamos tener instalado Python en nuestro sistema, pero si no lo tenemos instalado lo podemos hacer desde la Terminal ejecutando el comando:

- Instalación Python

```
$ brew install python --framework
```

Para la instalación (y desinstalación) de todos los paquetes se usará el Administrador de Paquetes de Python **pip**. Se ejecuta mediante la interfaz de línea de comandos. Veamos la sintaxis básica de pip:

```
$ pip help # muestra ayuda
$ pip list # lista los paquetes
instalados
$ pip install nombre-paquete
$ pip uninstall nombre-paquete
```

- Actualizamos e instalamos Virtualenv:

```
$ pip install --upgrade setuptools pip wheel
virtualenvwrapper
```

- Editamos el fichero `.bash_profile` y agregamos el siguiente código

```
#homebrew
export
PATH=/usr/local/bin:/usr/local/sbin:/Library/Frameworks/Python.framework/Versions/2.7/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin

# Python
export WORKON_HOME=/Users/cristhian/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python2.7
export PIP_VIRTUALENV_BASE=$WORKON_HOME
export PIP_RESPECT_VIRTUALENV=true
if [[ -r /usr/local/bin/virtualenvwrapper.sh ]]; then
    source /usr/local/bin/virtualenvwrapper.sh
    echo -----
    echo virtualenvwrapper started
    echo -----
else
    echo "WARNING: Can't find virtualenvwrapper.sh"
    echo "It's not in /usr/local/bin/"
```



```
fi

# Setting PATH for Python 2.7
# The original version is saved in .bash_profile.pysave
PATH="/Library/Frameworks/Python.framework/Versions/2.7/bin
:${PATH}"
export PATH
```

- Descargar e instalar Xcode 7.2 desde:  
<https://developer.apple.com/xcode/download/>
- Descargar e instalamos wxPython 2.9.5.0  
<http://sourceforge.net/projects/wxpython/files/wxPython/2.9.5.0/>

```
$ tar -xzf wxPython-src-2.9.5.0.tar.bz2
$ cd wxPython-src-2.9.5.0
$ PREFIX=/usr/local

$ ./configure --prefix=$PREFIX --enable-shared --enable-
monolithic --enable-unicode --enable-std_string --enable-
display --with-opengl --with-osx_cocoa --with-libjpeg --
with-libtiff --with-libpng --with-zlib --enable-dnd --
enable-clipboard --enable-webkit --enable-svg --with-expat
--with-macosx-version-min=10.11 --with-macosx-
sdk=/Applications/Xcode.app/Contents/Developer/Platforms/Ma
cOSX.platform/Developer/SDKs/MacOSX10.11.sdk --enable-
universal_binary=i386,x86_64 --disable-precomp-headers

$ make install
$ cd wxPython

$ python setup.py build_ext WXPORt=osx_cocoa
WX_CONFIG=$PREFIX/bin/wx-config UNICODE=1
INSTALL_MULTIVERSION=1 BUILD_GLCANVAS=1 BUILD_GIZMOS=1
BUILD_STC=1 --inplace

$ python setup.py install WXPORt=osx_cocoa
WX_CONFIG=$PREFIX/bin/wx-config UNICODE=1
INSTALL_MULTIVERSION=1 BUILD_GLCANVAS=1 BUILD_GIZMOS=1
BUILD_STC=1
```

- Creación del entorno aislado de Python

```
$ mkvirtualenv project
```

- Instalamos wxPython en nuestro entorno aislado [5]

```
$ cd /path/to/wxPython/build/above
$ python setup.py install WXPORt=osx_cocoa
WX_CONFIG=$PREFIX/bin/wx-config UNICODE=1
INSTALL_MULTIVERSION=1 BUILD_GLCANVAS=1 BUILD_GIZMOS=1
BUILD_STC=1
```

Instalamos otras librerías. Cabe recordar que han sufrido un par de modificaciones leves y se han subido a un repositorio particular de GitHub. Para su instalación se clonará el repositorio de ambas con los cambios ya hechos.

- Robot Framework

```
$ pip install --allow-external -e
git+https://github.com/CristhianCarmona/tools.git@robotfram
ework-2.9.2#egg=robotframework
```

- Selenium2Library

```
$ pip install --allow-external -e
git+https://github.com/CristhianCarmona/tools.git@robotfram
ework-selenium2library-1.7.4#egg=robotframework-
selenium2library
```

- Robot Framework IDE

```
$ pip install --allow-external -e
git+https://github.com/CristhianCarmona/tools.git@robotfram
ework_ride-1.5a2#egg=robotframework-ide
```

Si listamos todas los paquetes que tenemos instalados hasta el momento debería salir algo parecido a esto:

```
(project)crsthian-2:bin crsthian$ pip list
decorator (4.0.6)
docutils (0.12)
EasyProcess (0.2.1)
pip (7.1.2)
Pygments (2.0.2)
PyVirtualDisplay (0.1.5)
robotframework (2.9.2)
robotframework-ride (1.5a2)
robotframework-selenium2library (1.7.4)
selenium (2.48.0)
```

```
setuptools (18.2)
wheel (0.24.0)
wxPython (2.9.5.0)
wxPython-common (2.9.5.0)
```

- Configurando variables de entorno  
Copiar el fichero ejecutable *'ride'* en el siguiente directorio

```
/Users/cristhian/.virtualenvs/project/bin
```

Este fichero ejecutable modifica la variable de entorno PYTHONPATH que el RIDE consulta para saber dónde está instalado el Python, agrega la ubicación de las librerías implementadas por nosotros guardadas en TestLibs/lib, accede al directorio donde están implementados los tests del RIDE y finalmente lanza su interfaz de usuario.

- Desde la terminal lanzamos el fichero *ride*

```
$ ride
```

#### 4.1.3.4 Funcionamiento del Robot Framework IDE

Robot Framework IDE es un entorno de desarrollo integrado que implementa tests automatizados por Robot Framework.

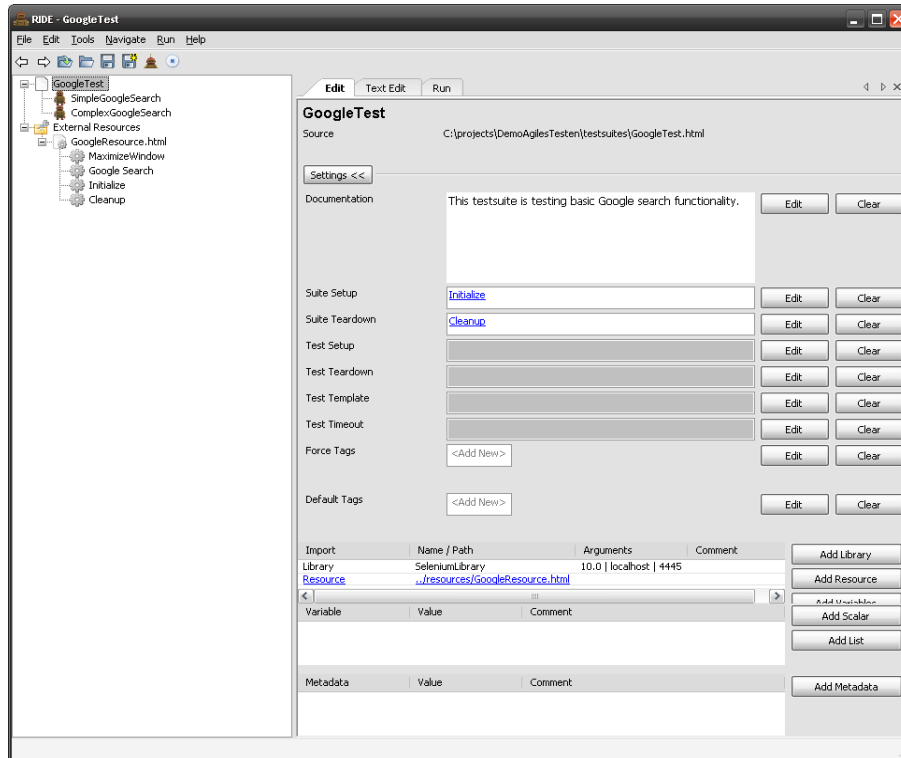


Imagen 9. GUI del Robot Framework IDE

En este mismo entorno podremos usar el framework Selenium2Library, que es una versión adaptada para el RIDE, para hacerlo simplemente importamos su librería.

Primero veamos su estructura de directorios y ficheros:

- Test Cases son creados en ficheros test case
- Un fichero Test Case crea automáticamente una Test Suite conteniendo los test cases en este fichero.
- Un directorio contiene ficheros test case que forman la capa de nivel más alta de la Test Suite
- Un directorio de Test Suite puede contener otro directorios Test Suite, e ir formando estructuras anidadas mas complejas.
- Los directorios Test Suite pueden tener un fichero especial de inicialización.

Además de estos, existen:

- Test Libraries que contienen las keywords de los niveles mas bajos.
- Los ficheros Resource con variables y keywords de usuario de los niveles mas altos.
- Los ficheros Variable para proveer formas más flexibles para crear variables que los ficheros Resource

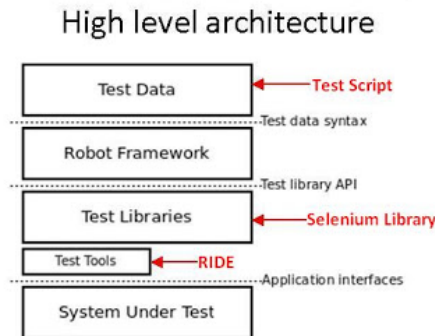


Imagen 10. Arquitectura básica de RIDE

En el siguiente Test Case básico podemos ver cuatro columnas, Test Case, Action, y dos de Argument. Cuando se ejecutar o lanza un Test Case el RIDE empieza a ejecutar cada una de las acciones de la columna Action que a su vez pueden tener argumentos o no, definidos en la columna Argument.

Test Case	Action	Argument	Argument
User can create an account and log in	Create Valid User	fred	P4ssw0rd
	Attempt to Login with Credentials	fred	P4ssw0rd
	Status Should Be	Logged In	
User cannot log in with bad password	Create Valid User	betty	P4ssw0rd
	Attempt to Login with Credentials	betty	wrong
	Status Should Be	Access Denied	

Si aplicamos metodología BDD a un Test Case tendría esta apariencia, el Test Case con todos los pasos (steps, en inglés), en este caso cada paso es una Keyword.

Test Case	Steps
User can change password	Given a user has a valid account
	when she changes her password
	then she can log in with the new password
	and she cannot use the old password anymore

Podemos importar las librerías que queramos, por ejemplo, para hacer operaciones propias del sistema operativo, o, librerías que desarrollemos nosotros.

Setting	Value
Library	OperatingSystem
Library	testlibs/LoginLibrary.py

#### 4.1.4 Elaboración de prueba de concepto

En esta etapa es conveniente realizar pruebas básicas que abarquen los pasos extremos de un escenario de una historia, con el propósito de verificar que las herramientas junto a los Frameworks que hemos instalado funcionan correctamente.

Para ello podremos implementar Test Cases haciendo uso del framework Selenium2Library, especialmente desarrollado para desarrollar acciones desde el RIDE

Su API resulta algo limitada y puede que para conseguir la ejecución de algunos pasos resulte algo lenta, aún así nos es suficiente para lograr implementar algunas interacciones y probar que todo el entorno hasta ahora instalado y configurado funciona correctamente.

#### 4.1.5 Desarrollo de un framework para la automatización

Para poder desarrollar nuestro propio framework hemos de saber cómo localizar los elementos que tiene un aplicación web, elegir la mejor estrategia para identificarlos, saber el funcionamiento del framework Selenium Webdriver y el Robot Framework para que finalmente podamos codificar todas las acciones que queramos.

##### 4.1.5.1 Localizar elementos con Firebug

Primero necesitamos localizar un elemento en el navegador, lo hacemos con la ayuda de Firebug. En este caso localizaremos el botón 'Iniciar sesión' (3)

1. Por defecto cada vez que se abre un nuevo navegador o pestaña Firefox desactiva el Firebug, lo activamos (1)
2. Activamos el botón del localizador (2) y clicamos sobre el elemento web que queremos identificar (3). Esto nos lleva al código HTML del elemento (4).

- Una vez localizado lo podemos identificar de varias formas, en este caso para identificar el botón 'Iniciar sesión' lo haremos mediante el nombre de la clase.

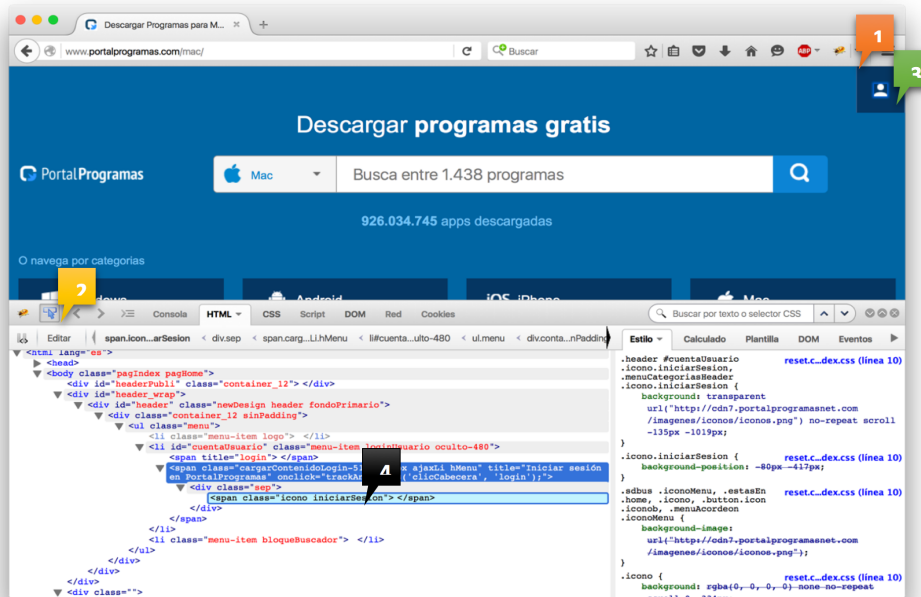


Imagen 11. Localizando un elemento en la aplicación

#### 4.1.5.2 Estrategias para localizar elementos en la aplicación web

El propósito básicamente los dividiremos en dos:

1. Identificar elementos para **simular** operaciones
2. Identificar elementos para **verificar** sus estados

Selenium Webdriver provee la clase By que soporta varias estrategias de localización, además encuentra métodos que toman un localizador u objeto solicitado como una instancia de la clase By como un argumento.

Existen varias formas de localizar un elemento HTML:

- Id.- lo más parecido a la solución ideal
- Link.- encuentra elementos por el texto mostrados en el link
- Class.- nombre de la clase de un elemento
- Tag names.- encuentra elementos basados en sus tags HTML.
- CSS Selector.- el navegador formatea al estilo CSS para buscar su elemento

- XPath.- es un lenguaje de petición para seleccionar nodos de un XML
- JQuery. Ejecuta scripts tipo JavaScript. Usa CSS Selectors
- Text.- con la ayuda de XPath se puede localizar un texto cualquiera

Una de las herramientas que nos pueden ayudar a esta tarea es Firebug en Firefox e Internet Explorer y para Chrome se hace click derecho sobre el propio elemento de la página luego Inspeccionar elemento.

Escrito en código Python:

```
driver.find_element_by_id(<elementID>)  
driver.find_element_by_link_text(<linkText>)
```

Diferencias de los dos principales métodos para encontrar elementos:

1. `find_element()`
  - 0 coincidencias: lanza excepción tipo `NoSuchElementException`
  - 1 coincidencia: devuelve una instancia tipo `WebElement`
  - 2 o mas coincidencias: devuelve sólo el primero que aparezca en la estructura de objetos (DOM) que genera el navegador cuando se carga un documento.
2. `find_elements()`
  - 0 coincidencias: devuelve una lista vacía
  - 1 coincidencia: devuelve la lista de una instancia tipo `WebElement`
  - 2 o mas coincidencias: devuelve una lista con todas las instancias que coincidan

#### 4.1.5.3 Codificación de librería usando Python y Selenium Webdriver

Se ha de crear una clase compuesta de varias funciones, que vistas desde el RIDE, las funciones son las denominadas keywords. Para poder implementar todas estas funciones que hacen instancias del framework Selenium Webdriver tenemos importar sus clases.

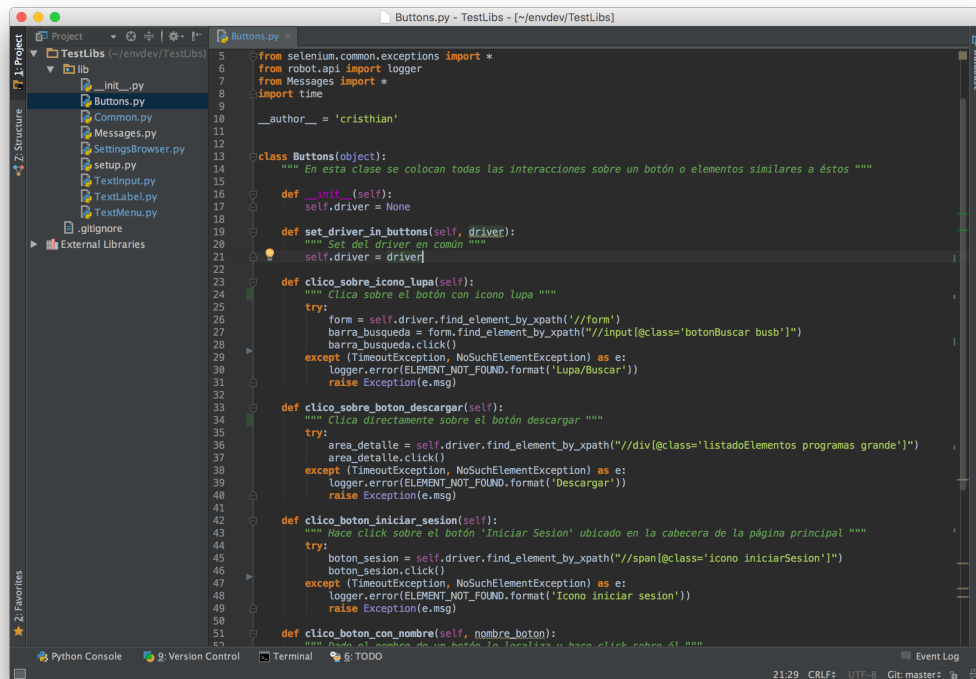
Por ejemplo, para implementar el click sobre el botón 'Inicio de sesión' se ha usado la clase **Button**. Analizando la función **click\_boton\_iniciar\_sesion** implementamos la acción click que hace un usuario sobre un elemento de la aplicación web.

Mediante el controlador del navegador 'driver' buscamos el elemento por el nombre de la clase.



Este 'driver' devuelve un objeto tipo WebElement sobre el que se pueden hacer varias acciones en función del tipo de elemento recuperado, entre ellas el click. Llamamos su método.

Para controlar los errores que puedan suceder durante la ejecución, capturamos las excepciones, escribimos un mensaje de error en la consola del RIDE y provocamos un error controlado.



```
Buttons.py - TestLibs - [~/envdev/TestLibs]
5 from selenium.common.exceptions import *
6 from robot.api import logger
7 from Messages import *
8 import time
9
10 __author__ = 'cristhian'
11
12 class Buttons(object):
13     """ En esta clase se colocan todas las interacciones sobre un botón o elementos similares a éstos """
14
15     def __init__(self):
16         self.driver = None
17
18     def set_driver_in_buttons(self, driver):
19         """ Set del driver en común """
20         self.driver = driver
21
22     def clico_sobre_icono_lupa(self):
23         """ Clica sobre el botón con icono lupa """
24         try:
25             form = self.driver.find_element_by_xpath("//form")
26             barra_búsqueda = form.find_element_by_xpath("//input[@class='botonBuscar busb']")
27             barra_búsqueda.click()
28         except (TimeoutException, NoSuchElementException) as e:
29             logger.error(ELEMENT_NOT_FOUND.format('Lupa/Buscar'))
30             raise Exception(e.msg)
31
32     def clico_sobre_boton_descargar(self):
33         """ Clica directamente sobre el botón descargar """
34         try:
35             area_detalle = self.driver.find_element_by_xpath("//div[@class='ListadoElementos programas grande']")
36             area_detalle.click()
37         except (TimeoutException, NoSuchElementException) as e:
38             logger.error(ELEMENT_NOT_FOUND.format('Descargar'))
39             raise Exception(e.msg)
40
41     def clico_boton_iniciar_sesion(self):
42         """ Hace click sobre el botón 'Iniciar Sesión' ubicado en la cabecera de la página principal """
43         try:
44             boton_sesion = self.driver.find_element_by_xpath("//span[@class='icono iniciarSesion']")
45             boton_sesion.click()
46         except (TimeoutException, NoSuchElementException) as e:
47             logger.error(ELEMENT_NOT_FOUND.format('Icono iniciar sesion'))
48             raise Exception(e.msg)
49
50     def clico_boton_con_nombre(self, nombre_boton):
51         """ Hace el nombre de un botón y se ejecuta y hace click sobre él """
52
```

Imagen 12. Implementación de acciones con Selenium Webdriver

Para poder usar todas estas clases debemos de importarlas desde el RIDE.

#### 4.1.6 Desarrollar código para testing, ejecutarlo y analizarlo

Llegados a esta etapa de la infraestructura, con las herramientas instaladas y el entorno de desarrollo configurado, lo mejor para probar su funcionamiento será someterla al ciclo de vida de software ágil, con iteraciones regulares.

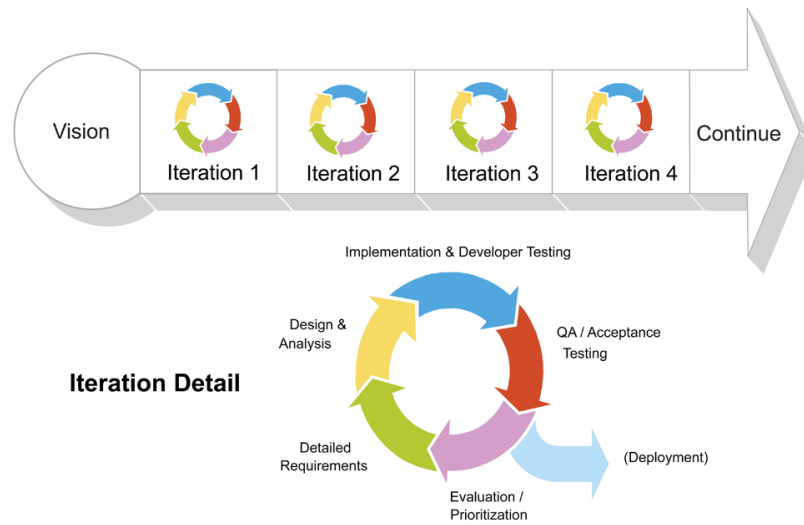


Imagen 13. Ciclo de vida de desarrollo iterativo del software

#### 4.1.6.1 Resumen de la aplicación web a testar

Para aplicar cada una de las etapas de este ciclo de desarrollo tomaremos como aplicación de pruebas la web <http://www.portalprogramas.com>.

La actividad de este portal es la de facilitar el acceso al software libre o de pruebas para que cualquier usuario pueda descargarlo. También ofrece ayuda a que nuevos desarrolladores puedan difundir sus aplicaciones mediante su plataforma.

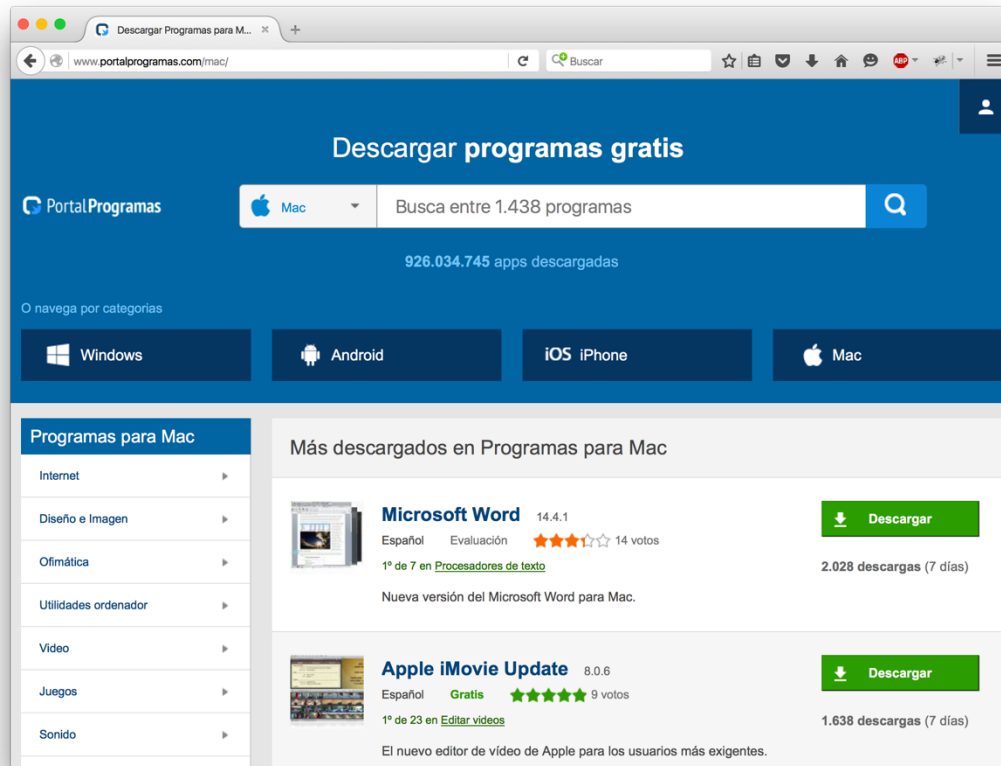


Imagen 14. Página de bienvenida de Portal Programas

#### 4.1.6.2 Probando infraestructura mediante desarrollo iterativo

##### 4.1.6.2.1 Etapa de requerimientos

En esta primera etapa comenzaremos por tomar requerimientos haciendo uso de Historias de Usuario porque son las que mejor se adaptan al enfoque hemos mencionado en capítulos anteriores, la BDD, que prueba el comportamiento de las historias más que los criterios de aceptación.

##### 4.1.6.2.1.1 Historias de usuario

Una historia de usuario es una representación de un requisito escrito en una o dos frases utilizando el lenguaje común del usuario. Cada historia debe ser limitada, pudiéndose escribir sobre una nota adhesiva pequeña.

Para redactarlas es recomendable seguir unos criterios:

- Independientes unas de otras
- Negociables. Deben ser valoradas por los clientes y por los usuarios mediante una discusión, lo cual permite esclarecer su alcance
- Estimables. Mediante discusión se estima el tiempo que tomará completarla
- Pequeñas. Las historias muy largas son difíciles de estimar e imponen restricciones sobre la planificación
- Verificables. Tratar que su verificación sea automática, de manera que pueda ser verificada en cada entrega del proyecto

Story Name  User Story  Spike  Foundation

As a

I want

so that

Acceptance Criteria:

COMMENTS:

Story Card Template ©2013 Database Consulting LLC www.database.com

Imagen 15. Historia de usuario en formato papel [11]

#### 4.1.6.2.1.2 Redactando requerimientos mediante historias de usuario

Se han creado unas cuantas historias de usuario que nos servirán para completar todo el proceso, desde los requerimientos hasta el análisis de los resultados de la ejecución

- Como usuario anónimo, quiero buscar un programa en el catálogo de todas las aplicaciones disponibles para poder consultar su detalle y descargarlo en mi equipo
- Como usuario anónimo, quiero registrarme en el sistema cuando necesite acceder a una ventaja o beneficio para poder obtener estas ventajas o beneficios
- Como usuario registrado quiero recibir notificaciones de una versión nueva de un software para poder bajarla desde el sistema a mi equipo
- Como usuario anónimo quiero cambiar de idioma cuando el idioma por defecto no sea mi preferido para poder leer mejor toda la información

- Como usuario anónimo quiero registrarme en el sistema como autor de un programa cuando tenga un programa por publicar para que cualquier persona pueda bajarlo a su equipo

#### 4.1.6.2.2 Etapa de análisis y diseño

En esta etapa analizaremos cada historia de usuario recogida de los requerimientos con el objetivo diseminarla en varios escenarios. Este desglose ayudará a evaluar todos los posibles flujos y casos que puede tener la aplicación para conseguir el objetivo principal de la historia.

Una vez se tengan los escenarios corresponde priorizar los mas sensibles. No necesariamente todos los escenarios de una historia se han probar.

El resultado del análisis es escrito en formato de las plantillas BDD. Al hacer uso de lenguaje natural tanto el equipo de las capas más altas del negocio como el equipo técnico no tendrán dificultades en entenderlo.

Otro aporte de este enfoque es que el equipo técnico sólo se dedicará a implementar el código de cada una de los escenarios que tiene la historia, dejando las tareas de análisis en un plano secundario.

##### 4.1.6.2.2.1 Adaptación al enfoque BDD

El BDD no tiene requerimientos formales para describir exactamente cómo estas historias de usuario deben ser escritas, pero sí insiste en que cada miembro del equipo declare las historias de usuario usando un formato simple y estandarizado.

Sin embargo, en 2007 Dan North [6] sugirió una plantilla para un formato textual, la cual ha encontrado un amplio seguimiento en diferentes herramientas de software basadas en TDD. [9]

```
Feature: texto descriptivo de lo que deseamos  
Scenario: alguna situación determinada del negocio  
Given alguna precondición del inicio del escenario  
And cualquier otra precondición  
When alguna acción detonante del actor  
And cualquier otra acción del actor  
Then algún resultado del detonante es obtenido  
And cualquier otro resultado es obtenido
```

#### 4.1.6.2.2 Cómo escribir historias de usuario en formato BDD

- Escribiremos tantas Feature como historias tengamos en los requerimientos. Básicamente se compone de un breve resumen de la historia de usuario
- Para cada Feature se crearán tantos Scenario como situaciones posibles queramos probar
- Cada Scenario constará de diferentes steps ‘Dado’, ‘Cuando’, ‘Entonces’ e ‘Y’ (Given, When, Then, And, en inglés) que serán todas las acciones que hace un usuario para terminar lo que se define en el Scenario
- Los Scenario se han de expresar idealmente en forma declarativa y no imperativa, es decir, en el lenguaje del negocio sin hacer referencia a las interfaces del usuario

#### Ejemplo de una historia de usuario en plantilla BDD

```
Feature: Quiero buscar un programa en el catálogo del sistema
  Scenario: Como usuario anónimo quiero buscar un programa que exista en el
  sistema
    Dado un usuario entra al sistema mediante la "${URL}"
    Cuando selecciono la plataforma con nombre "${nom_plataforma}"
    Y relleno el campo de búsqueda con el nombre del programa "${nom_programa}"
    Y clico en el botón de la lupa
    Entonces aparece el mensaje "Descargar" como resultado de la búsqueda
```

#### 4.1.6.2.3 Etapa de implementación y desarrollo de pruebas

En esta etapa ya disponemos de todas las historias desglosadas en uno o más escenarios y listas para empezar a implementar el código de los scripts.

##### 4.1.6.2.3.1 Implementando las historias de usuario en el RIDE

Como tenemos las historias escritas en formato BDD sólo tenemos que saber escribirlas en el RIDE. El orden es el mismo que se sigue en la etapa anterior de análisis cuando pasamos de la historia de usuario a BDD, pero esta vez la única diferencia es que tenemos que conocer la terminología básica del RIDE.

En la captura de debajo se puede observar el orden de escritura y, la terminología RIDE junto a su equivalente en BDD respectivamente.

1. Primero empezaremos creando las Test Suite
2. Crearemos las keywords que serán cada uno de los escenarios de la historia
3. Dentro de la Keyword que acabamos de crear, crearemos más keywords que serán cada uno de los pasos que hace un usuario para ese escenario
4. Finalmente para cada escenario crearemos un solo Test Case que contiene tantos juegos de datos como casos queramos probar.

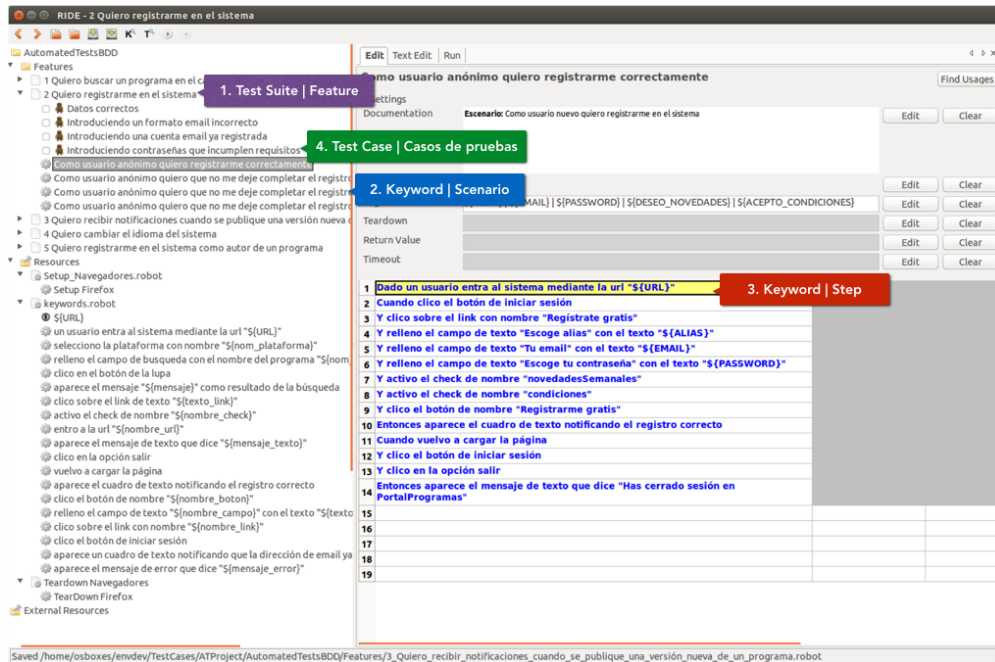


Imagen 16. Test Suites implementadas en RIDE

	ALIAS	EMAIL	PASSWORD	DESEO_NOVEDADES	ACEPTO_COND
1	usuario152345	j@server1.proseware.cor	12345	si	si
2	usuario212416	j@server1.proseware.cor	12345	si	si
3	usuaasdfa3153	j.s@proseware.com	12345	si	si
4	usuarlafsa43	js.@proseware.com	12345	si	si
5	usufasfaf2224	js@proseware.com	12345	si	si
6	usuarknk54635	js@proseware.com	12345	si	si
7	usuario654635	plainaddress	12345	si	si
8	usuaasdf54635	@%~%#s@#s.com	12345	si	si
9	usuarthhe4635	@example.com	12345	si	si
10	usuarlogsgt35	joe Smith <email@example.com>	12345	si	si
11	usuario65fg35	email.example.com	12345	si	si
12	usufwo654635	email@example@examp	12345	si	si
13	usuario654635	email@example.com	12345	si	si
14	usuario654635	email@example.com	12345	si	si
15	us3r340654635	email_email@example.ci	12345	si	si
16	usulouo654635	#i->.&@example.com	12345	si	si
17	usuarikmk635	email@example.com (joe Smith)	12345	si	si

Imagen 17. Test Case con juego de datos

#### 4.1.6.2.3.2 Librería con acciones implementadas

Con la capa más alta implementada empezamos a codificar los scripts de cada paso que tiene un escenario. Para ello seguiremos el mismo proceso explicado en el apartado DESARROLLAR UN FRAMEWORK PARA LA AUTOMATIZACIÓN.

En este proyecto se han desarrollado varias clases dentro del directorio lib capaces de ejecutar acciones de bajo nivel correspondientes a los pasos que se han de realizar en cada escenario de la historia.

- **SettingBrowser.py**  
Aplica valores a los parámetros que componen la configuración del navegador, el Setup y Teardown del navegador Firefox
- **TextInput.py**  
Acciones sobre los campos de texto del navegador
- **TestLabel.py**  
Realiza acciones sobre campos que tienen un id asociado y sobre campos tipo texto
- **TextMenu.py**  
Realiza acciones sobre agrupaciones de menú
- **Messages.py**  
Clase que agrupa en un solo lugar los mensajes que retorna en caso de error
- **Common.py**  
Ejecuta las acciones comunes
- **Buttons.py**  
Realiza acciones, sobretodo clics, en elementos tipo button



### 4.1.6.2.3.3 Instanciando librerías propias

Para poder hacer uso de esta clases la tenemos que importar desde el Setup del navegador.

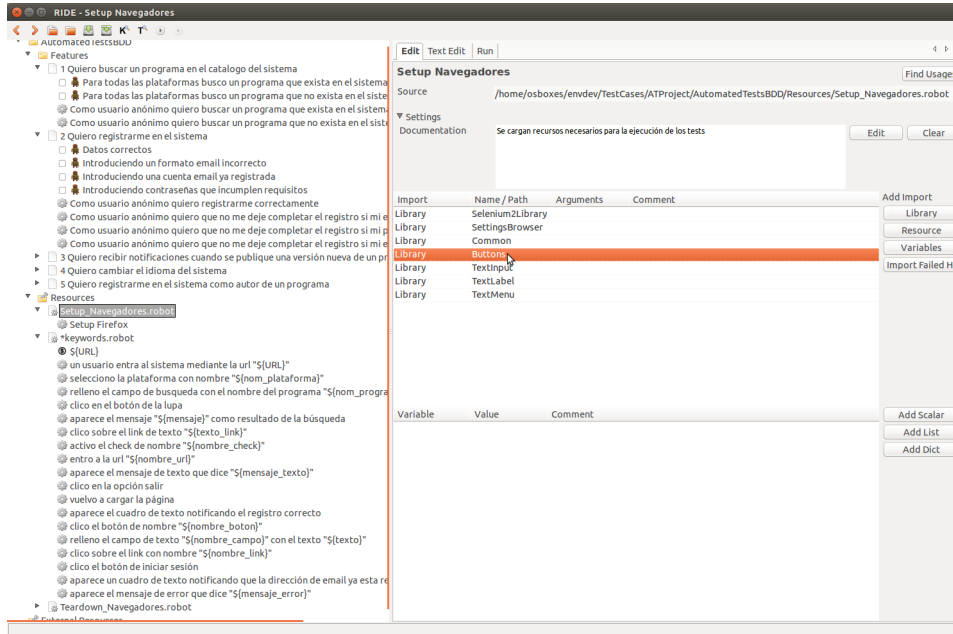


Imagen 18. Importación de librerías en RIDE

Una vez importada podemos llamar cualquiera de sus funciones, en este caso Keyword.

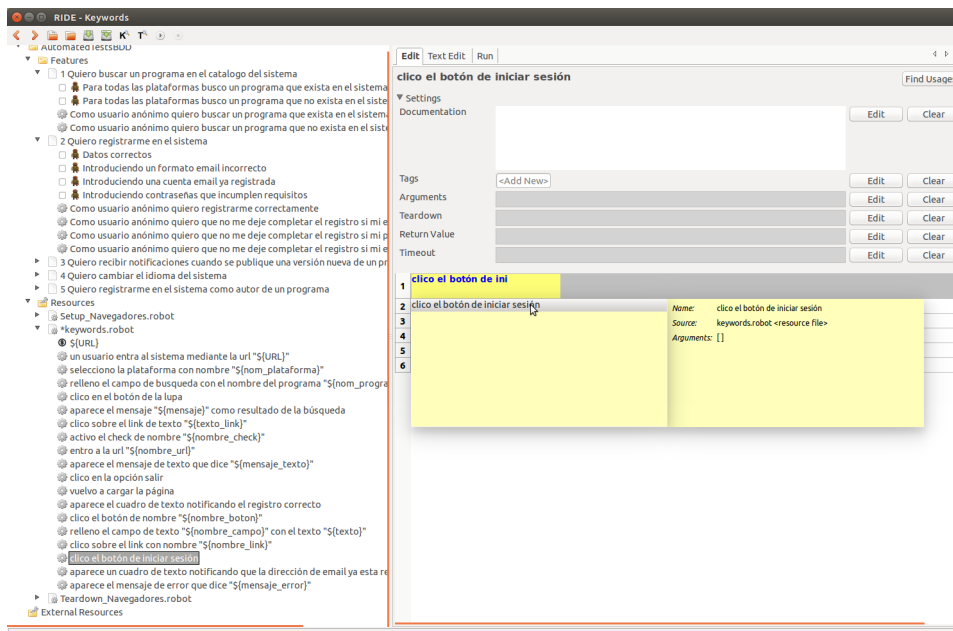


Imagen 19. Instanciando keywords de librerías propias

#### 4.1.6.2.3.4 Generación de conjunto de datos para Test Case

La recolección de datos para cada escenario de pruebas se ha de realizar manualmente conforme a los requerimientos de la aplicación.

#### 4.1.6.2.3.5 Ejecución de Test Case

Para ejecutar uno o más Test Case sólo tenemos que seleccionarlos y presionar el botón Start de la pestaña Run en el RIDE. A partir de aquí irán saliendo mensajes en su consola.

Cuando termine la ejecución aparecerá la barra de estado coloreada de verde o rojo que está justo debajo de la línea de argumentos (Arguments, en inglés)

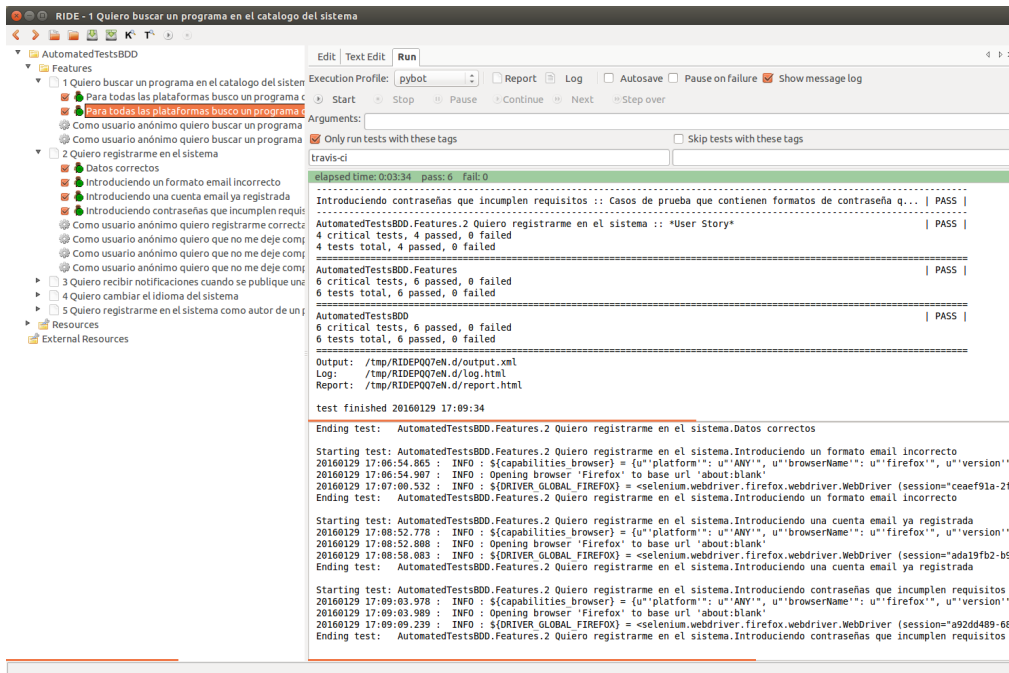


Imagen 20. Resultado de ejecución de los Test Cases con RIDE

Al finalizar la ejecución de todos los Test Case que se han lanzado se generan tres ficheros log con formato enriquecido HTML, tanto si todos pasaron 'PASS' como si alguno de ellos ha fallado 'FAIL'. En este último, el alto nivel de detalle generado permite seguir el rastro hasta la causa del fallo de manera sencilla.

Para abrir estos ficheros se ha de clicar sobre el botón Report o Log ubicado justo al principio de la pestaña Run. Se abren en el navegador configurado por defecto en el sistema operativo.

Otra de las características de estos ficheros es la inclusión de las estadísticas con el resultado de todos los Test Cases que se han lanzado.

En la imagen de abajo vemos que todos los Test Cases se han ejecutado correctamente, ninguno ha fallado.

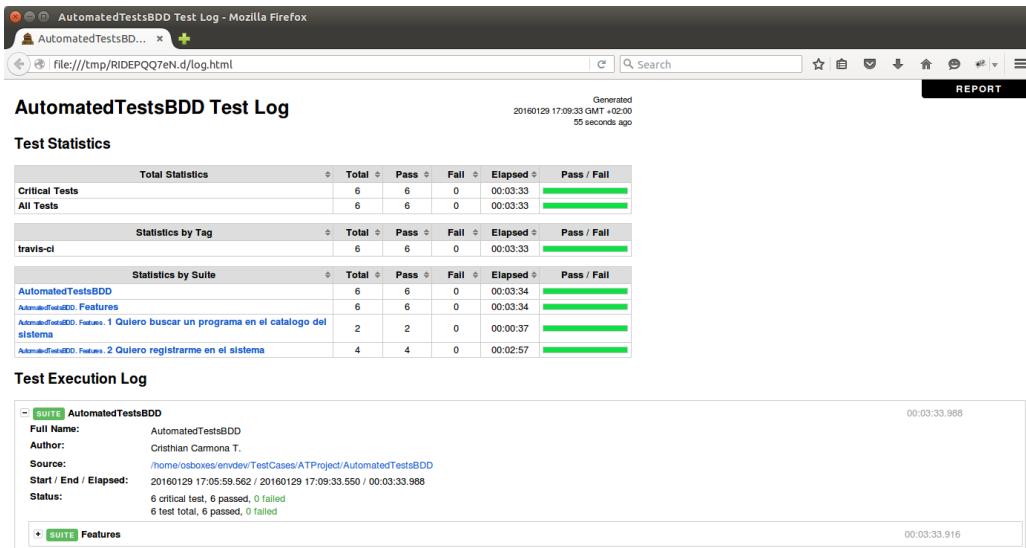


Imagen 21. Log de ejecución. Todos PASS

#### 4.1.6.2.4 Etapa de aceptación de pruebas

Su objetivo es que todas cumplan todos los requerimientos del cliente. Generalmente las personas que las realizan son usuarios finales o los clientes del sistema.

Los fichero logs generados a partir de la ejecución servirían, dado el caso, de evidencia cuando se detectan fallos en el comportamiento de la aplicación y se tienen que gestionar hacia el equipo de desarrollo para solventarlos.

Esta etapa no forma parte del análisis de este proyecto, pues correspondería hacerlas sobre unos requerimientos que partan de clientes reales.

#### 4.1.6.2.5 Etapa de evaluación y priorización

Ídem a la etapa anterior.

#### 4.1.6.2.6 Etapa de despliegue

La infraestructura aquí desarrollada tiene la capacidad de desplegar la aplicación de forma automática, mecanismo concedido por Travis-CI, pero para llevarlo a cabo es necesario disponer de acceso al código fuente de la aplicación.

En el hipotético caso que tuviéramos acceso al código fuente de la aplicación, lo que siguiente que haríamos es implementar las instrucciones y parámetros necesarios en el fichero `.travis.yml` para generar una versión entregable de la aplicación.

## 4.2 Automatizando pruebas con Integración Continua

Ya disponemos del entorno de desarrollo completamente operativo, las herramientas, el framework y ejecución de tests funcionan correctamente, nos corresponde escalar la infraestructura para que actúe de forma automática. Para ello, como hemos visto antes, con la ayuda de Travis-CI configuraremos todo el entorno de Integración Continua.

**Situación actual:** ahora mismo somos capaces de implementar tests automáticos, ya no desperdiciamos tiempo y recursos a tests manuales, la etapa de test en su conjunto se ha agilizado considerablemente aportando al final tener un producto de más calidad, pero hay un ligero problema. Aún se tiene la necesidad de que una persona lance los todos los tests.

La solución pasa por configurar el sistema de integración continua.

**Situación futura:** integrando Travis-CI a nuestra infraestructura terminaremos de cerrar el ciclo de vida para la generación de un producto de software. Para hacerlo la idea es replicar prácticamente el mismo entorno que hemos instalado de forma local, pero ahora en los sistemas remotos de Travis.

Para que los tests sean capaces de correr automáticamente, como se ha dicho antes hay que replicar el entorno local en sus máquinas virtuales.

Uno de los problemas que nos encontramos es que cuando lanzamos un test localmente se abre el navegador Firefox y empieza a ejecutar todo el test, hasta aquí todo correcto, el problema es que virtualmente este navegador requiere de la instalación de unas librerías especiales, de lo contrario no correrá.

Otra posible solución a este problema, es instalar un navegador virtual, el más popular es el PhantomJS. La gran desventaja de este navegador es que requiere mucho tiempo para instalarlo pudiendo llegar a ser un proceso tedioso.

Resuelto el problema del navegador, se siguen los mismos pasos que haría un tester de forma manual. Esta vez los replicamos para que se realicen de forma automática. La secuencia de los pasos es la siguiente:

- 1) Especificar el lenguaje de programación y su versión
- 2) Especificar variables de entorno, en caso de tenerlas
- 3) Especificar la versión del navegador
- 4) Instalar todas las dependencias (librerías u otros paquetes de software)
- 5) Instalar todos los Frameworks
- 6) Descargar la última versión del framework que hemos implementado desde nuestro repositorio
- 7) Descargar la última versión los casos de pruebas de nuestro repositorio
- 8) Actualizar nuevas variables de entorno
- 9) Finalmente, ejecutar el RIDE con el fichero Arguments.txt como parámetro de entrada. Este fichero contiene todos los tests que se han de ejecutar, en nuestro caso todos los test que lanzan tienen un tag llamado *travis-ci*

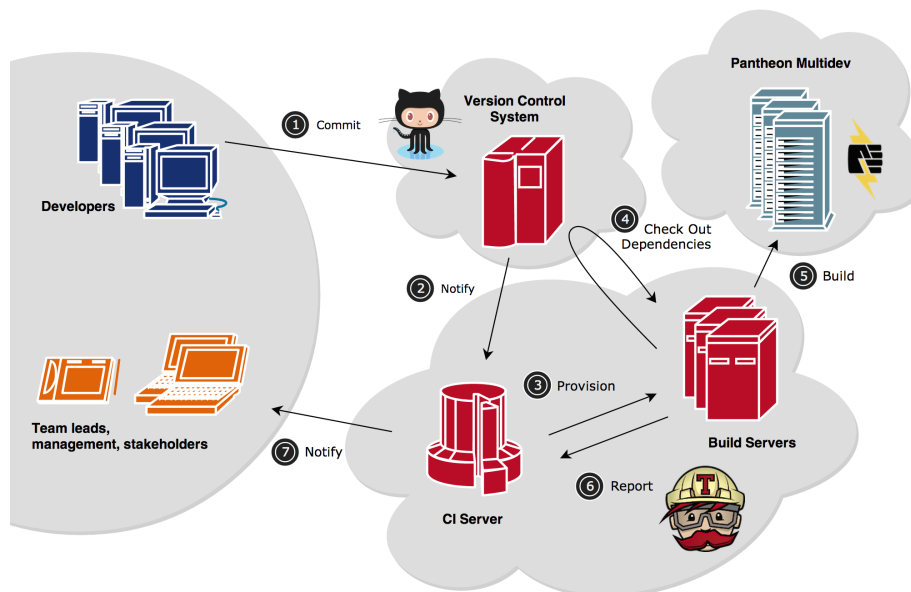


Imagen 22. Diagrama de funcionamiento de Travis-CI



## 5 Pruebas y resultados

Sobre la infraestructura actual se han efectuado pruebas manuales y automáticas, de las cuales se ha obtenido los siguientes resultados:

### 5.1 Resultados de tests manuales

Resultado de reproducir manualmente la historia número uno, que comprueba si existe o no un programa de software en el catálogo del sistema.

1 Iteración	
Manual	Segundos
Abrir el navegador	5
Escribir la url	11
Escribir el nombre de un programa	6
Resultado de la búsqueda	10
Total tiempo	32

Para lograr reproducirla se ha empleado una media de 32 segundos para una iteración, es decir, una sol búsqueda. Estos datos son suficientes para hacer la comparativa con las mismas pruebas hechas de forma automática.

Tiempo total empleado (en caso ideal) para la reproducción manual de las seis iteraciones.

Iteraciones manuales	6
Tiempo manual (s)	192

### 5.2 Resultados de tests automáticos

Resultados de reproducir automáticamente seis veces casos de pruebas diferente la historia número uno.

Automático			
Historia 1	Escenario1	Escenario2	Total
Iteraciones	3	3	6
Tiempo total (s)	17	17	34

Resultados de reproducir automáticamente la historia número dos con treinta iteraciones diferentes y con cuatro escenarios diferentes.

Automático					
Historia 2	Escenario1	Escenario2	Escenario3	Escenario4	Total
Iteraciones	1	23	1	5	30
Tiempo total (s)	10	96	8	22	136

### 5.3 Análisis de resultados tests manuales versus tests automáticos

Como observamos en el gráfico una simple prueba de comprobación de existencias en el catálogo del sistema de forma manual, cuesta más que reproducirla automáticamente. Mientras se tardan 192 segundos en reproducir manualmente 6 iteraciones, incluso en menos la infraestructura es capaz de reproducir automáticamente 36 iteraciones que implican más acciones.

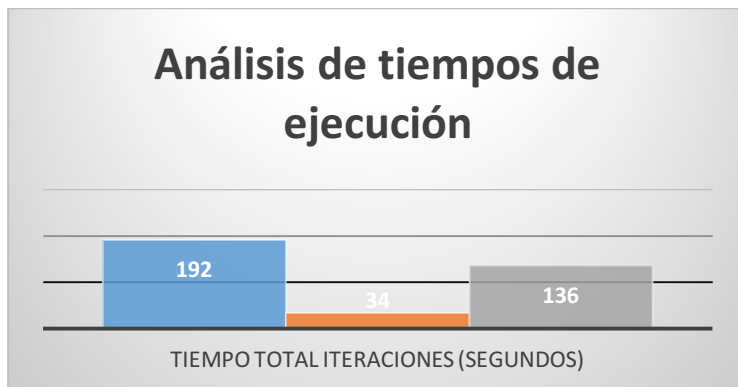


Imagen 23. Tiempos de ejecución de test manuales y automáticos

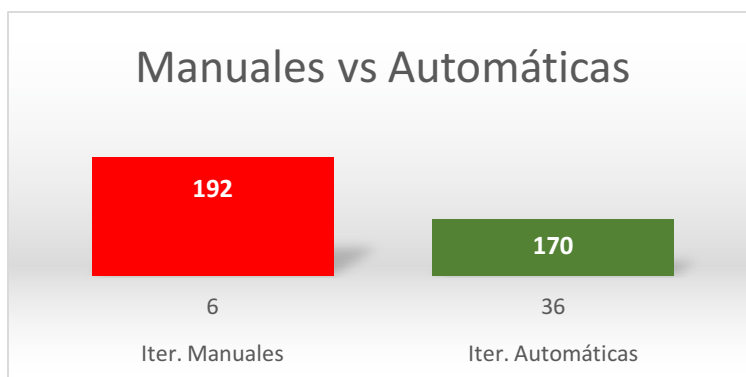


Imagen 24. Tiempos test manuales vs test automáticos



## 6 Conclusiones

De los resultados y análisis obtenidos con la ejecución de pruebas automáticas en el capítulo anterior se desprenden varias afirmaciones:

- Todas las herramientas que se han instalado no han tenido absolutamente ningún coste en licencias, todas han sido OpenSource, por tanto el ahorro es sustancial si comparamos lo que cuesta la licencia de otras herramientas destinadas a la misma tarea.
- La adopción de una metodología ágil de desarrollo que abarca todas las etapas del ciclo de vida del software ha permitido que en todas trabajemos bajo la misma disciplina, pese a que cada una ejecute tareas independientes, al 'hablar' el mismo lenguaje ha provocado que todo el proceso se desarrolle de forma más ágil
- La implementación de sistemas de Integración Continua no solo ha agilizado la ejecución de tests automáticos, sino que ahora se ha reducido la cantidad de errores manuales que acarreaban los otros tests, menos recursos destinados a la etapa de testing, y la gran ventaja de disponer de todos los mecanismos que aporta este sistema.
- Las pruebas realizadas han servido para demostrar que el tiempo destinado a testing manual es mucho mayor que el destinado a testing automático, todo y que al principio la curva de aprendizaje pueda resultar un poco elevada y requiera más tiempo, a medio-largo plazo los beneficios se notan.

Concluir diciendo que se han cumplido todos los objetivos marcados al principio del proyecto. Se ha conseguido reducir los tiempos por consiguiente el dinero destinado a tales fines.

## 7 Contribuciones

Se han hecho algunas mejoras en las tres principales librerías para adaptarlas a nuestro idioma 'Español', de lo contrario tendría que haber combinado inglés y español, o escribir todas los pasos en inglés. Cosa que limita el número de usuario del IDE porque no todo el mundo habla inglés.

El segundo cambio que se hizo fue para poder hacer instancias del controlador del navegador Firefox ejecutado por *Selenium WebDriver* desde nuestras propias librerías.

Lo que conseguimos con esto es, poder ejecutar acciones de las librerías ya desarrolladas (*Selenium2Library*) y poder implementar nuestro propio framework. Ambos actuarán sobre el mismo navegador.

Estos cambios ya están implementados en los respectivos paquetes y subidos al repositorio público.

El repositorio principal está dividido en 4 ramas (branches en inglés) [TestTools](#):

- ↳ [Master](#) que aloja los cuatro paquetes principales en estado original, es decir, los cambios que nombro arriba no están.
- ↳ [robotframework-2.9.2](#)
- ↳ [robotframework ride-1.5a2](#) Antes sólo se podía escribir test usando sintaxis Gherkin para BDD en inglés, ahora además del inglés podemos emplear su traducción en Español.
  - Given ... Dado
  - When ... Cuando
  - Then ... Entonces
  - And ... Y
- ↳ [robotframework-selenium2library-1.7.4](#) Uso compartido de controlador del navegador Firefox con nuestras librerías.

### Repositorios del código fuente

<https://github.com/CristhianCarmona/TestTools>

<https://github.com/CristhianCarmona/Sources>

<https://github.com/CristhianCarmona/TestCases>

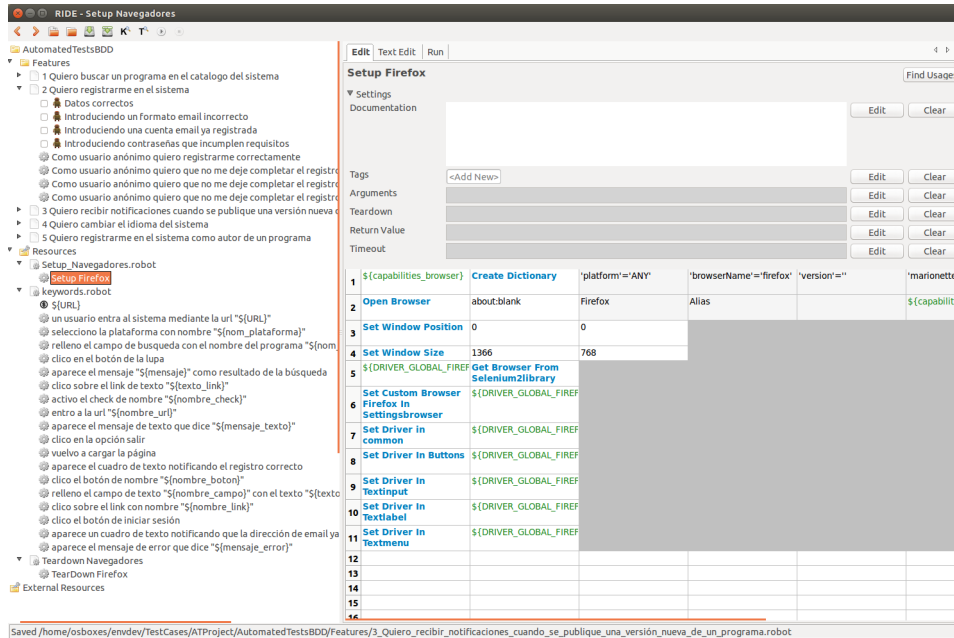
<https://github.com/CristhianCarmona/TestLibs>

**Además se adjunta todo en un fichero ZIP.**

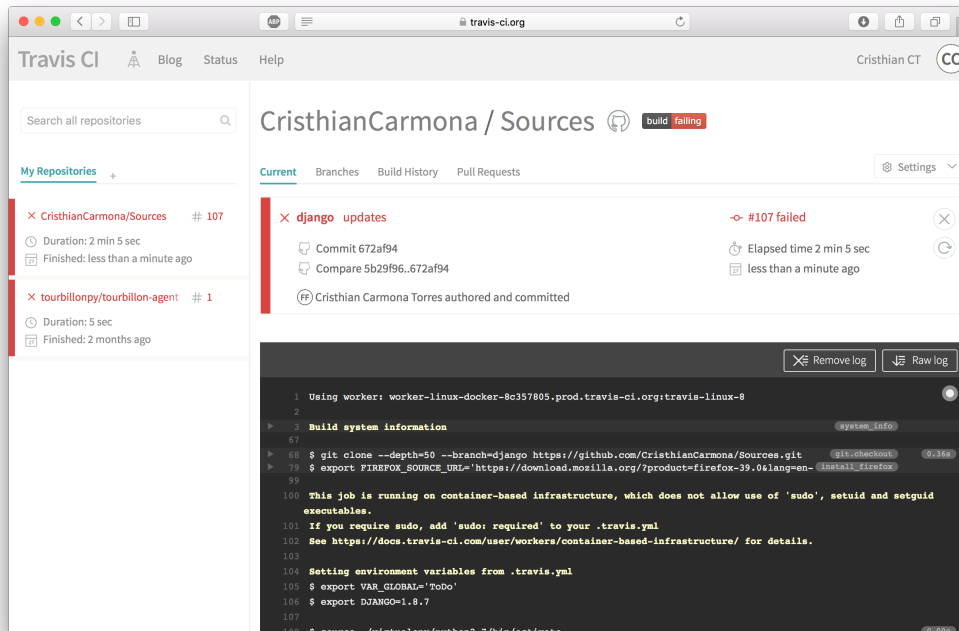
## 8 Anexos

### 8.1 Capturas de pantalla

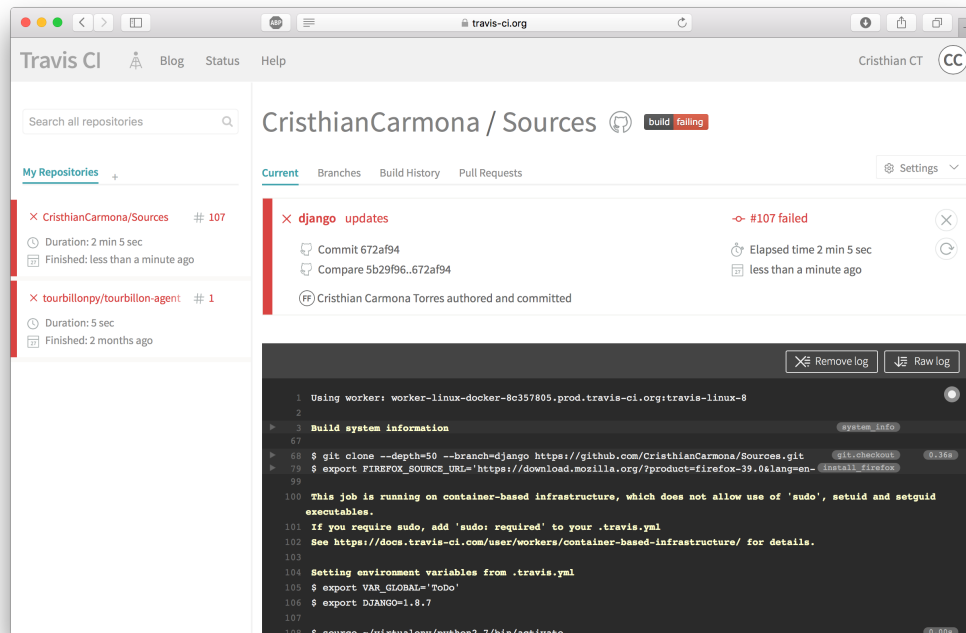
#### Configuración del navegador Firefox



#### Pantalla de ejecución fallida de tests automáticos en Travis.



Pantalla de ejecución correcta de tests automáticos en Travis.



## 8.2 Versiones de las herramientas aquí utilizadas

- Robotframework-2.9.2
- Robotframework-selenium2library-1.7.4
- Robotframework\_ride-1.5a2
- PyCharm 5.0.3
- SourceTree 2.0.5.8
- Firefox 43.0.4
- Firebug 2.0
- Python 2.7.11
- Virtualenv (13.1.2)
- Xcode 7.2
- wxPython 2.9.5.0
- VM Ware Fusion (trial)
- Imagen Ubuntu 14.4
- Travis-CI (Registro gratis)
- GitHub (Registro gratis)