Universidad de Valladolid
E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Mención en Sistemas de Telecomunicación

ANÁLISIS DEL PROCESO DE PRUEBAS Y SU ENTORNO. DESARROLLO DE VARIOS CASOS PRÁCTICOS

AUTOR: RODRIGO SÁNCHEZ ALONSO

TUTORA: MÍRIAM ANTÓN RODRÍGUEZ

VALLADOLID, JULIO 2020

TÍTULO: Aprendizaje y desarrollo de las pruebas y su entorno

AUTOR: Rodrigo Sánchez Alonso

TUTORA: Míriam Antón Rodríguez

DEPARTAMENTO: Teoría de la Señal y Comunicaciones e Ingeniería Telemática

TRIBUNAL

PRESIDENTA: Mª ÁNGELES PÉREZ JUÁREZ

SECRETARIO: DAVID GONZÁLEZ ORTEGA

VOCAL: MIRIAM ANTÓN RODRÍGUEZ

SUPLENTE 1: JAVIER AGUIAR PÉREZ

SUPLENTE 2: MARIO MARTÍNEZ ZARZUELA

FECHA:

CALIFICACIÓN:

Resumen

El objetivo del presente trabajo de fin de grado es el desarrollo de dos proyectos de automatización de pruebas, junto con todo su entorno. El proyecto empieza creando toda la infraestructura para poder ejecutar nuestras pruebas automáticas. En un mundo cada vez más tecnológico, es cada vez más aconsejable contar con equipos que aseguren la calidad de tu producto de software o hardware.

Primero expone toda la teoría oficial para poder contextualizar al usuario. A continuación, presenta todas las herramientas que conforman la infraestructura y cómo configurar cada una de ellas. Por último, incluye dos ejemplos de proyectos de pruebas. El primero de ellos se ejecuta contra servicios REST, y el segundo de ellos se ejecuta contra un servicio WEB.

El lenguaje que utilizado para la creación del proyecto es JAVA puesto que ofrece un alto rendimiento y una alta capacidad de aplicación en diferentes sistemas, y sobre todo un entorno fiable y seguro. Todas las tecnologías utilizadas han sido seleccionadas en base a la experiencia del autor, y toda posible mejora está incluida en el último capítulo.

Palabras clave: *Pruebas, REST, Web, Automatización, Serenity, Cucumber, Jenkins, Docker, Bitbucket, Git, Selenoid*

Abstract

The aim of this end-of-degree work is the development of two test automation projects, together with their entire environment. The project begins by creating the entire infrastructure to be able to execute our automatic tests. In an increasingly technological world, it is advisable to have equipment that ensures the quality of your software or hardware product.

First it exposes all the official theory to be able to contextualize the user. Then it presents all the tools that make up the infrastructure and how to configure each of them. Finally, it includes two examples of test projects. The first one is executed against REST services, and the second one is executed against a WEB service.

The language used for the creation of the project is JAVA since it offers a high performance and a high capacity of application in different systems, and above all a reliable and safe environment. All the technologies used have been selected based on author's experience, and any possible improvement is included in the last chapter.

Keywords: *Tests, REST, Web, Automatization, Serenity, Cucumber, Jenkins, Docker, Bitbucket, Git, Selenoid*

A mi familia, en especial a mis padres y a mi abuela.

A mis amigos, en especial a Carlos y Alvar.

A mi tutora Míriam.

Índice de figuras

1. Introducción

Los sistemas de software son ya una parte integral de nuestras vidas, desde las aplicaciones de negocio (como, por ejemplo, la banca) hasta los productos de consumo más cotidianos como pueden ser los automóviles. La mayoría de las personas ha sufrido la experiencia de no recibir el comportamiento esperado de algún producto software. Un error en una factura, debido al retraso en el cargo de una tarjeta de crédito, o el proceso de un sitio web que no se carga correctamente, son ejemplos comunes de los problemas que pueden ocurrir debido a un producto software que no funciona correctamente.

Un producto de software que no funciona correctamente puede causar muchos problemas, tales como la pérdida de dinero, tiempo o reputación en el ámbito del negocio, e incluso lesiones de muerte en productos de uso crítico. Hacer pruebas de software es una forma de evaluar la calidad del software y de reducir el riesgo de fallos en un entorno de operaciones o en producción. Algunos ejemplos de fallos notables en sistemas software son:

- El 20 de Noviembre del año 2009, durante su primera semana de funcionamiento comercial en la ruta París-Nueva York, la función del piloto automático del Airbus A380 sufrió tal fallo en su software que se vio obligado a regresar a Nueva York (Samuel, 2009)
- En 1998, la misión de la NASA, Mars Climate Orbiter, donde por un problema de conversión de unidades trajo consigo la pérdida de la nave espacial y de toda la misión (Rus, 2020).
- Mars Polar Lander de la NASA, donde una partícula de polvo generó una respuesta incorrecta de uno de los tres trenes de aterrizaje, causando el apagado del motor de la sonda a 40 metros de altura sobre la superficie. Supuso la pérdida de la sonda y de la misión (Europa Press, 2019).

El proceso de pruebas del software es necesario para reducir el riesgo de pérdida de gastos financieros, de daños, bienes o hasta la muerte.

Un proceso de pruebas riguroso es necesario durante el desarrollo y mantenimiento para identificar los defectos y así reducir los fallos en el entorno operativo y aumentar la calidad del sistema. La ejecución de las pruebas nos ayuda a medir la calidad del producto y por lo tanto del servicio. También se revisan los procesos; por ejemplo, por una auditoría, donde varios métodos pueden utilizarse para revisar trabajos. A su vez, es posible que el proceso de pruebas de software requiera satisfacer requisitos legales o contractuales, o estándares específicos para la industria. Cuanto más crítico es el sistema de software para la industria, más probabilidades hay de que exista un estándar de pruebas (ej. DO -178B para la aviación).

El proceso de pruebas nos ayuda a medir la calidad de software. La calidad es el grado en el cual un componente, sistema o proceso satisface requisitos especificados y/o necesidades y expectativas del usuario/cliente (IEE Std 610). Por lo tanto, podemos definir la calidad del software como la totalidad de la funcionalidad y prestaciones de un producto de software que están relacionadas con su capacidad de satisfacer las necesidades explícitas o implícitas. El proceso de pruebas puede aportar fiabilidad a la calidad del software probado si se encuentran unos pocos defectos o ninguno. Una prueba diseñada de forma apropiada reduce el nivel de riesgo general en un sistema. Cuando el proceso de pruebas sí que encuentra defectos, la calidad del sistema software aumenta cuando se solucionan dichos defectos. Aún así, se deben aprender lecciones de proyectos previos. Analizando la causa raíz podemos anticipar defectos en proyectos posteriores, mejorando de esta manera la calidad en futuros sistemas. Las pruebas

deben estar integradas como una actividad más del proceso de aseguramiento de la calidad (es decir, junto con el desarrollo de estándares, formación y análisis de defectos).

¿Cuántas pruebas son suficientes? Es una pregunta con una compleja respuesta. Probar completamente todo, incluso si lo piden los clientes o los gerentes, simplemente no se puede pagar. Se necesita un enfoque de pruebas que proporcione una cantidad correcta de pruebas para el proyecto, clientes (otras partes implicadas) y el software. Para hacerlo hay que basarse en los riesgos, incluyendo los riesgos técnicos y de negocios relacionados con los productos y proyectos, y las restricciones tales como el tiempo o el presupuesto. Además, el proceso de pruebas debe proporcionar suficiente información a los implicados a la hora de tomar decisiones bien fundamentadas sobre el lanzamiento del software o del sistema al que se le está realizando el proceso de pruebas, para el próximo paso o periodo de desarrollo a los clientes (ISTQB, 2018).

2. Proceso de pruebas de Software

¿Que son las pruebas?

El proceso de pruebas es el proceso que incluye todas las actividades, tanto estáticas como dinámicas, del ciclo de vida con respecto a la planificación, preparación y evaluación de los productos software y los documentos relacionados, para así comprobar que los productos software satisfacen los requisitos, para comprobar que se ajustan al uso previsto y para hallar defectos.

La definición se puede desglosar en dos partes. La primera de ellas se fija en el proceso de pruebas como un proceso en vez de una actividad individual, y que involucra una serie de actividades. El proceso de pruebas como proceso se produce durante el ciclo de vida del desarrollo del software. Cuanto más tarde encontremos defectos en el ciclo de vida del software, más caros serán de arreglar. Si podemos encontrar y corregir los defectos de requisitos en la etapa de requisitos, construiremos un software más adecuado a las necesidades del cliente, de forma correcta, y a un coste reducido. Así mismo, podemos desglosar una serie de procesos derivados del proceso de prueba (Crispin & Gregory, 2008):

- **Ejecución** de pruebas estáticas y dinámicas. En las pruebas estáticas no hace falta ejecutar el objeto de prueba. Puede ser una revisión de documentos o análisis estático. Ésta es una manera útil y rentable de realizar pruebas.
- **Planificación** de las actividades del proceso de pruebas. Ésta tiene lugar antes y después de la ejecución de las pruebas. Hay que gestionar las pruebas a realizar: planificar qué queremos hacer, controlar las actividades de prueba, informar del progreso de las pruebas y del estado del software bajo prueba y finalizar o cerrar las pruebas cuando se completa una fase.
- **Preparación:** necesitamos decidir qué proceso de pruebas vamos a hacer al seleccionar las condiciones de uso y al diseñar los casos de prueba.
- **Evaluación:** revisión de los resultados y evaluar el software y los criterios de salida que se prueban.

La segunda parte de la definición cubre alguno de los objetivos para el proceso de pruebas (Crispin & Gregory, 2008):

- Determinar que (los productos software) satisfacen los requisitos específicos. Si los productos satisfacen su especificación, podemos proporcionar esa información para ayudar a los implicados a juzgar la calidad del producto y decidir si está listo para ser utilizado.
- Demostrar que (los productos software) sirven su propósito. Ver si el software hace lo suficiente para ayudar a que los usuarios lleven a cabo sus tareas y ver si el software hace lo que el usuario juzgue razonable.
- Detectar defectos: encontrar defectos nos ayuda a entender los riesgos asociados al poner en funcionamiento el software y solucionar los defectos ayuda a mejorar la calidad

de los productos. Sin embargo, identificar los defectos no tiene otro beneficio; con el análisis subyacente, también podemos mejorar el proceso de desarrollo y nos ayuda a realizar menos errores en productos futuros.

Es evidente, con esta definición, que el proceso de pruebas no solo consiste en ejecutar todos los casos de prueba. Esta es una de las actividades del proceso de pruebas, pero no es todo el proceso de pruebas.

La definición del proceso de pruebas inicial es válida para cualquier nivel de pruebas, desde las pruebas de componente hasta las pruebas de aceptación. Atendiendo tanto al proceso como a los distintos objetivos de estos niveles de pruebas diferentes, contamos con: Pruebas de componente (unitarias), pruebas de integración, pruebas de sistema y pruebas de aceptación, las cuales describiré más adelante.

El proceso de pruebas puede contar con los siguientes objetivos (Crispin & Gregory, 2008):

- **Identificar defectos**
  El objetivo principal en las pruebas de software puede ser el causar tantos fallos como sean posible para que los defectos en el software se identifiquen y se solucionen.
- **Aumentar la confianza sobre el nivel de calidad**
  Objetivo principal en las pruebas de aceptación. Puede ser el confirmar que el sistema funciona como se esperaba.
- **Facilitar información para la toma de decisiones**
  Medir la calidad del software e informar a los implicados sobre el riesgo de lanzar el producto en un momento dado.
- **Prevenir los defectos**
  Como en las pruebas estáticas, para reducir los costes de detectar y arreglar defectos durante la prueba dinámica.

Inicialmente, solo conocemos el efecto de un defecto pero no la ubicación específica en el software. Para poder corregir un defecto o bug, primero se debe ubicar en el software: la localización y la corrección de los defectos son tareas del programador del software (generalmente) y se los conoce como depuración. Por lo general, reparar un defecto aumenta la calidad del producto, siempre y cuando no se introduzcan nuevos defectos a la hora de intentar corregirlo. Por esta razón, depuración y pruebas son dos conceptos y actividades completamente diferentes. La depuración es la tarea de localizar y corregir defectos. El objetivo de las pruebas es detección de fallos (que indican la presencia de defectos) (Patton, 2001).

Siete principios del proceso de pruebas

Principio 1: Las pruebas demuestran la presencia de defectos.

Las pruebas pueden demostrar que un producto falla; es decir, que hay defectos. Las pruebas no pueden demostrar que un programa está libre de defectos. Incluso si no se han encontrado fallos durante las pruebas, esto no prueba que no existan defectos. Es decir, las pruebas apropiadas minimizan las probabilidades de que existan defectos ocultos en el objeto de la prueba (ISTQB, 2018).

Principio 2: Las pruebas exhaustivas no son posibles.

Unas pruebas exhaustivas donde se ejecutan todos los valores posibles para todas las entradas y sus combinaciones, teniendo en cuenta todas las precondiciones diferentes, requeriría una gran cantidad de casos de prueba, lo cual no es realista para diseñar o ejecutar. Por lo tanto las pruebas exhaustivas son imposibles (excepto en casos triviales excepcionales). Esto implica elegir qué pruebas diseñar y ejecutar. El esfuerzo de las pruebas debe gestionarse teniendo en cuenta el riesgo del producto y las prioridades (ISTQB, 2018).

Principio 3: Pruebas tempranas

Las actividades del proceso de pruebas deberían empezar lo más temprano posible en el ciclo de la vida del software y se deberían enfocar en los objetivos definidos. Todo ello contribuye a la detección temprana de defectos. Son fundamentales ya que el coste de detectar y reparar los defectos aumenta con el tiempo (ISTQB, 2018).

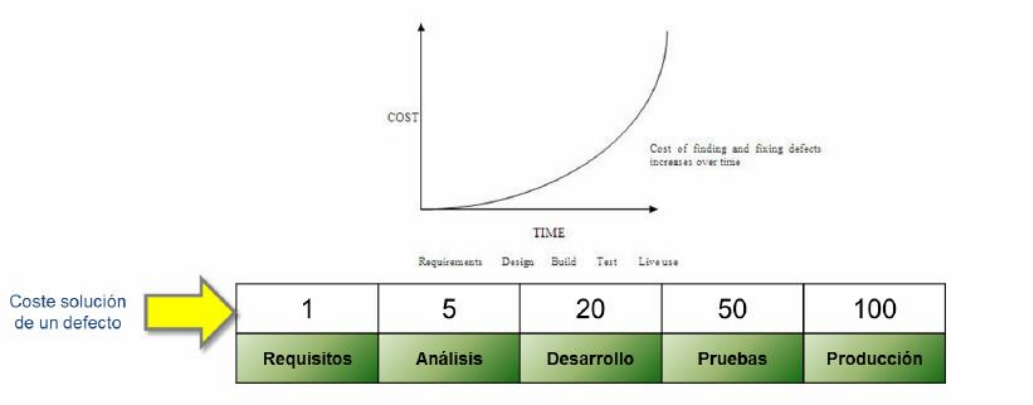


Figura 1: Esquema de la realización de las pruebas tempranas

Principio 4: Agrupación de defectos

La probabilidad de la existencia de más defectos en un área de un software es proporcional al número de defectos ya encontrados en dicha área. Por lo tanto, los defectos ni siquiera están distribuidos, si no que tienen a venir agrupados: en un software típico algunas áreas parecen ser más propensas a defectos que otras. En términos de rendimiento de las pruebas, los esfuerzos de pruebas adicionales se concentrarán en estas áreas propensas a defectos (Patton, 2001).

Principio 5: Paradoja del pesticida

Si las mismas pruebas se repiten una y otra vez, tienden a perder su efectividad, ya que no descubren nuevos defectos. Para mantener la efectividad de las pruebas se debería desarrollar casos de pruebas nuevos y modificar los existentes. Probar aquellas partes del software a las que hasta ahora no se les había realizado unas pruebas o realizar combinaciones de entradas sin utilizar. De esta manera, puede que se detecten más defectos (ISTQB, 2018).

Principio 6: Las pruebas dependen del contexto.

Las pruebas se deben adaptar a los riesgos y al entorno de la aplicación. Por lo tanto, no se debería realizar pruebas a dos sistemas de la misma manera. Por ejemplo: un sistema de seguridad crítica requiere pruebas diferentes a una aplicación de comercio electrónico o un videojuego. El resultado de las pruebas puede variar según el entorno en que probemos. Se debe probar en un entorno lo más similar posible al de producción. Un objeto de prueba que

funciona correctamente fuera de su contexto puede no ser válido para otro contexto (ISTQB, 2018).

Principio 7: Falacia de la ausencia de errores

Este principio se refiere a la falacia de asumir que si no hay fallos, el sistema es adecuado. Un software al que no se le han encontrado defectos no significa, necesariamente, que esté libre de ellos. Detectar fallos y reparar defectos no garantiza que el sistema satisfaga las expectativas y necesidades del usuario. La participación temprana de los usuarios en el proceso de desarrollo y el uso de prototipos son medidas preventivas para evitar problemas (Patton, 2001).

Proceso básico de las pruebas

El proceso básico de las pruebas consiste en planificar y controlar las pruebas, analizar y diseñarlas, implementarlas y ejecutarlas, evaluar los criterios de salida e informes y proceder a las actividades de cierre de pruebas.

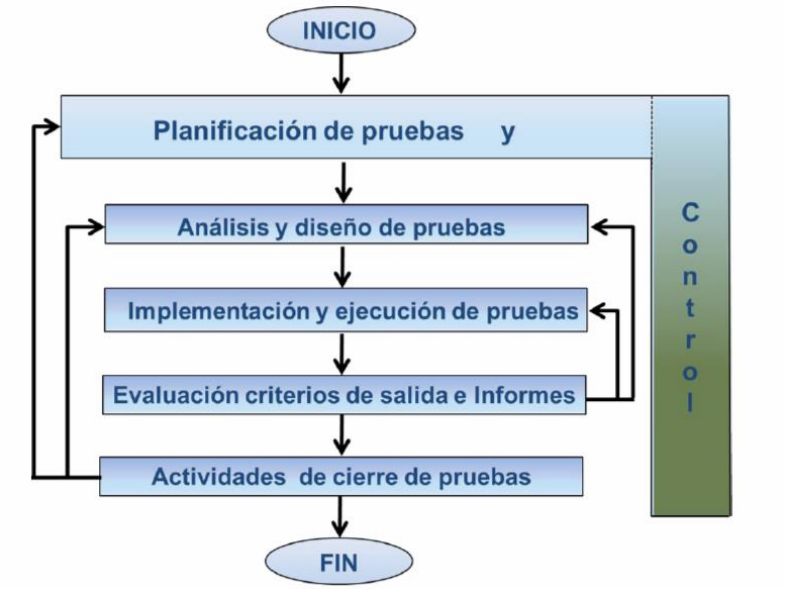


Figura 2: Proceso básico de pruebas

Planificación y control de las pruebas

La planificación de las pruebas es la actividad de definir los objetivos de las pruebas y la especificación de las actividades de pruebas con vistas a cumplir los objetivos y la misión establecidos. La planificación del proceso de pruebas comienza al principio del proyecto de desarrollo de software. Durante la planificación de las mismas, tenemos que entender los objetivos de los clientes, partes implicadas y del proyecto, a la vez que los riesgos que debe abordar el proceso de pruebas. Esto permite definir la misión y los objetivos del proceso de pruebas y derivar un enfoque y un plan para las pruebas, incluyendo también la especificación de las actividades de pruebas. Otro objetivo importante es la estimación de los recursos necesarios para el proceso de pruebas.

Éste, y otros aspectos de la planificación deben documentarse en los entregables fundamentales de esta fase: el plan de pruebas. La principal tarea de la planificación es determinar la estrategia de las pruebas. La estrategia de las pruebas es una descripción de alto nivel de los niveles de pruebas a ser realizados y al proceso de pruebas dentro de esos niveles para una organización o programa (uno o más proyectos). Ya que no son posibles unas pruebas exhaustivas, se deben establecer las prioridades basándose en la evaluación del riesgo. Las actividades de pruebas se deben distribuir a los subsistemas individuales, dependiendo del riesgo esperado y la severidad de los efectos del fallo. El objetivo de la estrategia de las pruebas es la distribución óptima de las pruebas a las partes "indicadas" del sistema de software.

La planificación de las pruebas también incluye las definiciones de los criterios de salida, como pueden ser el alcance de las pruebas o el índice de los fallos. Los criterios de salida están relacionados con la intensidad de las pruebas y deben determinarse basados en el riesgo. Una vez definidos todos los criterios de salida de las pruebas, se evalúan sus valores tras ejecutar los casos de prueba, para decidir si se puede finalizar el proceso de pruebas. Se deben considerar cuidadosamente los aspectos de tiempo durante la planificación. La priorización de las pruebas es fundamental ya que garantiza que las partes críticas del software se prueben primero, en el caso de que las restricciones de tiempo no permitan ejecutar todas las pruebas planificadas. También se deben planificar cuidadosamente las necesidades específicas de herramientas de soporte, infraestructura, formación, etc…

El control de las pruebas es el seguimiento de las actividades de las pruebas y la comparación de lo que realmente sucede durante el proyecto con la planificación, informando del estado de las desviaciones del plan y tomando las medidas para cumplir con la misión y los objetivos de la nueva situación. La planificación de las pruebas debe actualizarse continuamente, teniendo en cuenta la retroalimentación del seguimiento y el control. Parte de las tareas de la gestión de las pruebas son las de administrar y mantener los procesos de las pruebas, la infraestructura de las pruebas y el producto de soporte de prueba. Entendiendo como infraestructura de las pruebas a los artefactos organizativos necesarios para realizar el proceso de pruebas, formados por los entornos de las pruebas, las herramientas de pruebas, el entorno de trabajo y los procedimientos, y entendiendo como producto de soporte de prueba a los artefactos producidos durante el proceso de pruebas. Son resultado de planificar, diseñar y ejecutar pruebas, tales como la documentación, las contribuciones, los resultados esperados, los procedimientos de configuración y resolución, los archivos, las bases de datos, el entorno y cualquier software o utilidades adicionales utilizadas en el proceso de pruebas (Whittaker, 2002).

Análisis y diseño de pruebas

Entendemos por base de prueba a todos los documentos de donde los requisitos de un componente o sistema pueden ser inferidos. La documentación en la que se basan los casos de prueba. Entendemos por condición de prueba a un elemento o suceso de un componente o sistema que puede ser verificado por uno o más casos de prueba. Ejemplo: una transacción o un elemento estructural. Entendemos por objeto o elemento de prueba al componente o sistema a ser probado.

El análisis y diseño de las pruebas es la actividad durante la cual los objetivos de las pruebas se transforman en las condiciones y casos de pruebas tangibles. Consta de las siguientes actividades principales (Crispin & Gregory, 2008):

- Revisar las bases de las pruebas (tales como el análisis de riesgo del producto, los requisitos, la arquitectura, las especificaciones de diseño y las interfaces). Al analizar las

bases de las pruebas, es posible identificar las diferencias y ambigüedades en las especificaciones, lo que previene que aparezcan defectos en el código.

- Evaluar la capacidad de prueba de las bases y de los objetos de las pruebas. Ejemplo: Realizar pruebas para saber con qué facilidad se pueden abordar las interfaces y en qué partes se puede dividir el objeto de las pruebas para hacerlo más fácil de probar.
- Identificar y priorizar las condiciones de las pruebas basándose en el análisis de los elementos de prueba, la especificación, el comportamiento y la estructura del software. En resumen, una lista de alto nivel sobre lo que debe ser probado.
- Diseñar y priorizar los casos de pruebas de alto nivel. Se debe utilizar técnicas de pruebas para ayudar a seleccionar las pruebas representativas que se relacionan con los aspectos particulares del software y que conllevan riesgos o que son de interés particular, basándose en las condiciones de las pruebas y entrando en detalle.
- Identificar los datos de las pruebas necesarios para respaldar las condiciones de las pruebas y los casos de pruebas. Para los casos de pruebas lógicos (alto nivel) / concretos (bajo nivel).
- Diseñar la configuración del entorno de las pruebas e identificar cualquier herramienta e infraestructura requerida.
- Crear una trazabilidad bidireccional entre la base de pruebas y casos de prueba. Es esencial cubrir con casos de prueba los elementos de la base de pruebas que han de analizarse. Esto también asegura que los cambios que se realizan en los objetos de la base de pruebas o en los casos de prueba son trazables.

Para cada caso de prueba, se debe describir la situación inicial (precondición). Asimismo, se deben definir por adelantado cuáles serán los resultados y los comportamientos que se esperan. los resultados incluyen las impresiones, los cambios a los datos y estados globales (persistente) y cualquier otra consecuencia del caso de prueba (postcondición). Para definir los resultados esperados, el probador debe obtener la información de alguna fuente adecuada. El oráculo de pruebas es la fuente para determinar resultados esperados para compararlos con los resultados reales del software. Un oráculo puede ser un sistema existente (para una evaluación comparativa), un manual de usuario o el conocimiento especializado de un individuo, pero no debería ser el código (Crispin & Gregory, 2008).

Las especificaciones de las pruebas se diseñan en dos pasos (Patton, 2001):

1. Se tienen que definir a nivel lógico (casos de prueba de alto nivel) durante la fase de análisis y diseño de pruebas: los casos de prueba de alto nivel carecen de datos de prueba.
2. Se detallan en casos de prueba concretos (casos de prueba de bajo nivel) durante la fase de implementación y ejecución: Los casos de prueba de bajo nivel sí que tienen datos de prueba.

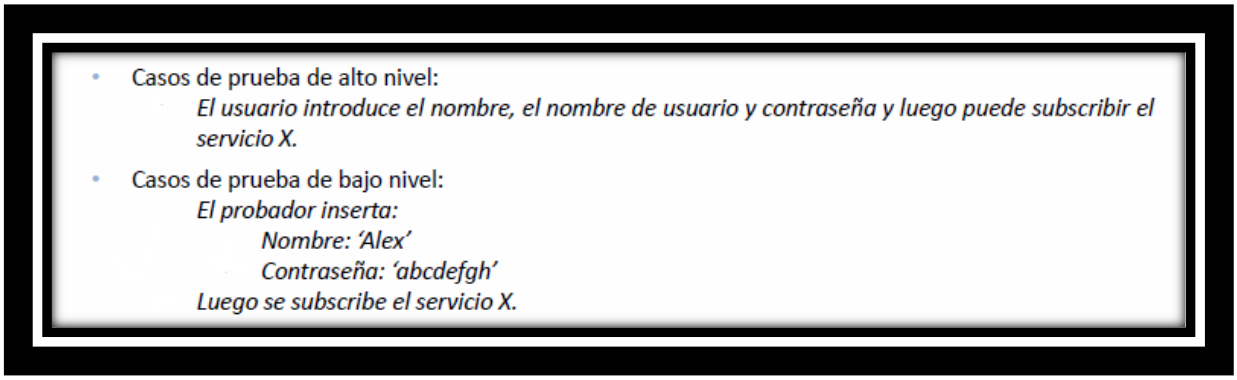


Figura 3: Ejemplo de casos de prueba de alto nivel y de bajo nivel.

Implementación y ejecución de pruebas

La implementación y ejecución de pruebas son las actividades donde las condiciones de las pruebas y los casos de prueba lógicos se transforman en casos de prueba concretos, todos los detalles del entorno se configuran para dar soporte a la actividad de ejecución de las pruebas y se ejecutan y registran las pruebas. Las funciones principales de esta fase son (Crispin & Gregory, 2008):

- Finalizar, implementar y priorizar los casos de prueba (incluyendo la identificación de los datos de prueba).
- Desarrollar y priorizar los procedimientos de las pruebas, crear los datos de prueba y a modo opcional, preparar los arnés de prueba y escribir los guiones de prueba automatizados. El arnés de prueba es el entorno de prueba constituido por stubs y controladores necesarios para llevar a cabo una prueba. Los procedimientos de prueba representan instrucciones para llevar a cabo las pruebas, en algunos casos se requiere automatizar algunas pruebas utilizando el arnés de pruebas y los guiones de pruebas automatizados.
- Crear los juegos de pruebas a partir de los procedimientos para la ejecución efectiva de las pruebas. Un conjunto de pruebas es una colección lógica de casos de prueba que, naturalmente, trabajan juntos (ejecución eficiente y un resumen más fácil).
- Programar un calendario para la ejecución de la prueba.
- Verificar que se ha configurado el entorno de pruebas de manera correcta. Se pueden realizar pruebas específicas para ello.
- Verificar y actualizar una trazabilidad bidireccional entre la base de pruebas y los casos de prueba.
- Ejecutor los procedimientos, ya sea manualmente o utilizando las herramientas de ejecución de pruebas, según la secuencia prevista.
- Registrar el resultado de la ejecución de las pruebas, identidades y versiones del software probado, las herramientas de pruebas y los productos de soporte de pruebas. Se deben informar los defectos frente a las versiones específicas. El registro de las pruebas proporciona un registro de auditoría. El registro de auditoría es el camino por el cual se puede recuperar la entrada original a un proceso (por ejemplo datos) a través del mismo proceso, tomando como punto de partida la salida del proceso. Esto facilita el análisis de defectos y permite llevar a cabo la auditoría de un proceso.
- Comparar los resultados reales con los resultados esperados.
- Reportar las discrepancias como incidencias y analizarlas para establecer sus causas. Por ejemplo: Un defecto en el código, en datos de prueba específicos o un error en la manera en la que se ha ejecutado una prueba.
- Reportar las discrepancias en forma de incidencias si durante la ejecución de las pruebas se muestra una diferencia entre los resultados esperados y los reales. Posteriormente analizarlas para decidir si realmente es un fallo o no.

Las correcciones pueden conllevar a nuevos defectos. Por lo tanto es necesario repetir actividades de pruebas como (Whittaker, 2002):

- Prueba de confirmación: es la re-ejecución de una prueba que falló previamente para confirmar una solución del defecto.
- Pruebas de regresión para asegurar que los defectos no se hayan introducido en áreas inalteradas del software o que la solución del defecto no haya revelado otros defectos.

Evaluación de los criterios de salida e informes

Evaluar los criterios de salida e informes es la actividad donde se evalúa el objeto de las pruebas contra los objetivos definidos; los criterios de salida de las pruebas (también conocidos como criterios de finalización). Consta de las siguientes tareas principales (Whittaker, 2002):

- Comprobar los registros de prueba contra los criterios de salida especificados en la planificación e las pruebas. Revisar la evidencia de qué pruebas han sido ejecutadas y revisadas y qué defectos han surgido, han sido solucionados, han sido confirmados o han sido destacados.
- Evaluar si se necesitan más pruebas o si los criterios de salida especificados deben cambiarse. Se puede decidir una terminación normal de todas las pruebas si se satisfacen todos los criterios, o bien, se decide que se deben ejecutar casos de prueba adicionales, o bien que los criterios tenían un nivel excesivamente alto.
- Elaborar un informe resumen de pruebas para los implicados. No es suficiente que los probadores sepan el resultado de las pruebas. Cuando se cumplen los criterios de prueba o se clarifica su incumplimiento, se debe redactar un informe. En los niveles de prueba más bajos (pruebas de componentes) esto podría tomar la forma de un mensaje al jefe del proyecto sobre como satisfacer los criterios de salida. En las pruebas de alto nivel (pruebas de integración, pruebas de sistemas) es posible que se realice un informe oficial.

Al tener en cuenta el riesgo, se debe determinar un criterio de salida adecuado para cada técnica de prueba utilizada. Por ejemplo, unas pruebas son suficientes si el 80% de los casos de prueba se ejecutaron de manera satisfactoria durante la ejecución de las mismas pruebas. Si no se cumple al menos un criterio de salida después de ejecutar todas las pruebas, se deben ejecutar más pruebas. Debe prestarse atención para asegurar que los nuevos casos de prueba conllevan a cumplir los criterios de salida respectivos, en caso contrario será trabajo adicional pero ninguna mejora (Patton, 2001).

Existen dos puntos principales relacionados con el no cumplimiento de los criterios de salida de las pruebas (ISTQB, 2018):

1. El esfuerzo extra no es justificado
   Un análisis más cercano del problema puede a su vez mostrar que el esfuerzo necesario para cumplir los criterios de salida no es apropiado. En esa situación, se cancelan las otras pruebas. Una decisión de tal magnitud debe considerar el riesgo asociado.
2. Es imposible cumplir un criterio específico en el caso concreto.

Si se planifican otras pruebas, se debe reanudar el proceso de pruebas y se debe decidir en qué punto el proceso de pruebas se reanudará. A veces, hasta puede ser necesario revisar el plan de pruebas o modificar las especificaciones de las pruebas. Además de los criterios de alcance de pruebas, se pueden utilizar otros criterios para definir el final de las pruebas.

Se debe considerar realizar varios ciclos de pruebas. Es ilógico no planificar tales ciclos de corrección solo porque se asume que no se encontrarán fallos durante las pruebas. Como se asume que las pruebas obtendrán nuevos fallos, se deberán eliminar los defectos adicionales y probar en otro ciclo de pruebas. Si se ignora este ciclo, los retrasos del proyecto se vuelven habituales. El final de una prueba se define por factores que no tienen conexión directa con las pruebas: tiempo y costes. Si estos factores conllevan a interrumpir las actividades de pruebas es porque no se proporcionó los recursos suficientes en el plan de proyecto o porque se subestimó el esfuerzo para unas pruebas determinadas (Whittaker, 2002).

Actividades del cierre de pruebas

Las actividades de cierre de pruebas deberían hacerse durante esta última fase en el proceso de pruebas, para así analizar la experiencia obtenida durante el trabajo de pruebas y para poner a disposición esta experiencia en proyectos futuros. Los datos son recopilados de las actividades de pruebas completadas para consolidar la experiencia, los productos de soporte de pruebas y los hechos y números. Las actividades de cierre de pruebas tienen lugar en los hitos del proyecto, como cuando se lanza un sistema de software, cuando se completa un proyecto de pruebas (o cancela), cuando se haya alcanzado un hito, o en una versión de mantenimiento donde se hayan completado las pruebas. Son de mucho interés las desviaciones entre la planificación y la ejecución para las diferentes actividades, así como las causas que lo provocaron.

Las actividades de cierre de pruebas constan de las siguientes tareas principales (Patton, 2001):

- Comprobar cuáles de los entregables planificados se han entregado.
- Cerrar los informes de incidencias o aumentar los registros de cambios para cualquiera que permanezca abierto.
- Documentar la aceptación del sistema.
- Finalizar y archivar los productos de soporte de pruebas, el entorno de pruebas y la infraestructura de pruebas para un uso posterior.
- Se deberán entregar los productos de soporte de prueba a la organización de mantenimiento, de manera que pueda servir para una reutilización posterior en el mismo proyecto o en otros similares como guía o base para estos. Los productos de soporte de pruebas pueden proporcionar pruebas judiciales de las pruebas realizadas. La organización responsable del mantenimiento del software ya en producción tiene que tener un soporte para realizar reparaciones de bugs o cambios de mantenimiento, también puede utilizar los productos de soporte de pruebas en las pruebas de confirmación y en las pruebas de regresión.
- Analizar las lecciones aprendidas para determinar los cambios necesarios en los lanzamientos y proyectos futuros.
- Utilizar la información recolectada para mejorar la madurez de las pruebas. Una evaluación crítica de las actividades realizadas en el proceso de pruebas, teniendo en cuenta los recursos utilizados y los resultados alcanzados, probablemente muestre las posibilidades de mejora. Si se utilizan estos resultados en proyecto futuros, se podrá alcanzar una mejora continua del proceso, que es algo muy interesante y que debemos resaltar.

La psicología de las pruebas

¿Puede un programador probar su propio programa? Realmente no existe una respuesta universalmente válida. Las debilidades pueden ser (Whittaker, 2002):

- Tendencia a dar por hecho algo que realmente no lo es.
- Olvidar los casos de prueba efectivos ya que el programador está más interesado en programar que en probar, o porque solo prueba superficialmente.
- Si el programador implementa un error de diseño fundamental, entonces probablemente no lo encuentre o no lo vea, debido a una interpretación inadecuada.

Por otro lado, es una ventaja poseer un buen conocimiento de los objetos de las pruebas propias y se ahorra tiempo ya que no es necesario aprender el objeto de las pruebas. Se debe tomar una decisión entre estas dos opciones basándose en el carácter crítico del objeto de las pruebas y en el riesgo de fallo asociado.

Un equipo de pruebas independiente tiende a incrementar efectividad de las pruebas. El probador puede revisar el objeto de las pruebas sin sesgos. No es su producto y las posibles suposiciones y malos entendidos del programador no tienen que ser necesariamente las suposiciones y malos entendidos del probador. Se pueden establecer varios niveles de independencia tal y como se muestra a continuación (de menor a mayor) (Whittaker, 2002):

1. Las pruebas diseñadas por las personas que escribieron el software probado.
2. Las pruebas diseñadas por terceros (otros miembros del equipo de desarrollo).
3. Las pruebas diseñadas por personas provenientes de otro grupo de la organización (por ejemplo, un equipo de pruebas independiente) o especialistas de pruebas (por ejemplo: especialistas de pruebas de seguridad).
4. Las pruebas diseñadas por personas provenientes de una organización o empresa distinta (es decir, la externalización o certificación por un órgano exterior).

A menudo el proceso de pruebas se ve como una actividad destructiva, aun cuando es muy constructiva en la gestión de riesgos del producto. Buscar fallos en un sistema requiere curiosidad, "pesimismo profesional", un ojo crítico, atención a los detalles, buena comunicación con los compañeros de desarrollo y experiencia en la cual poder basar la suposición de errores. Tanto el desarrollo como las pruebas son actividades constructivas, aunque la mentalidad de los programadores y de los probadores es diferente. Los programadores asumen que su desarrollo del producto de software es la implementación correcta de la especificación de requisitos. Los probadores asumen que un producto de software es inherentemente defectuoso y que es su trabajo identificar dichos defectos (Patton, 2001).

La forma como se reportan los fallos es fundamental para la cooperación entre los programadores y los desarrolladores; identificar los defectos del software realizado por otras personas no es un trabajo fácil y requiere de diplomacia. Se debe definir claramente y con antelación qué constituye un fallo o discrepancia para evitar las reacciones en contra de cualquier crítica. Hay varias maneras de mejorar la comunicación y las relaciones entre los probadores y las otras personas del equipo (ISTQB, 2002):

- Recordar a todos el objetivo común: sistemas de mayor calidad.
- Comunicar los resultados del producto de forma neutral y enfocada en los hechos, sin criticar a la persona que lo haya creado.
- Tratar de comprender el sentimiento de la otra persona y la razón de su reacción.
- Confirmar que la otra persona comprenda lo que has dicho, y viceversa.

Código deontológico

La participación en un proceso de pruebas de software permite a las personas tener conocimiento de información confidencial y privilegiada. Es necesario un código deontológico, entre otras razones, para garantizar que la información no se utilice inapropiadamente. Reconociendo los códigos deontológicos IEEE (*Insitute of Electrical and Electronics Engineers*) y ACM (*Association of Computing Machinery*), el ISTQB (*International Software Testing Qualifications Board*) establece el siguiente código deontológico (ISTQB, 2018):

Público

Los probadores de software certificados actuarán en coherencia con el interés público.

Cliente y empleador

Los probadores de software certificados actuarán de la manera más conveniente para su cliente y empleador, en coherencia con el interés público.

Producto

Los probadores de software certificados garantizarán que los entregables que proporcionen (en los productos y sistemas que prueben) satisfacen los más altos estándares profesionales.

Criterio

Los probadores de software certificados mantendrán integridad e independencia en su criterio profesional.

Gestión

Los directivos y líderes certificados de software se suscribirán y promocionarán un enfoque ético a la gestión de pruebas de software.

Profesión

Los probadores de software certificados promoverán la integridad y reputación de la profesión en coherencia con el interés público.

Compañeros de trabajo

Los probadores de software certificados serán equitativos y apoyarán a sus compañeros de trabajo fomentando la cooperación con los desarrolladores de software.

Nivel individual

Los probadores de software certificados participarán en un aprendizaje permanente de las prácticas de su profesión y promoverán un enfoque ético a la práctica de la profesión.

Modelos de desarrollo del software

Como he explicado anteriormente, las pruebas se deben realizar con el comienzo de desarrollo del producto de software. Para entender mejor las pruebas y sus niveles, primero vamos a describir los distintos modelos de desarrollo (ISTQB, 2018) que nos podemos encontrar en la vida real.

Modelo en V (modelo de desarrollo secuencial)

El modelo en V contiene en su proceso de evolución e integración los niveles de prueba del modelo clásico en cascada. Aunque existen variantes del modelo en V, un tipo común de modelo en V utiliza cuatro niveles de pruebas, correspondientes a los cuatro niveles de desarrollo:

- Pruebas de componente (unitarias).
- Pruebas de integración.
- Pruebas de sistema.
- Pruebas de aceptación.

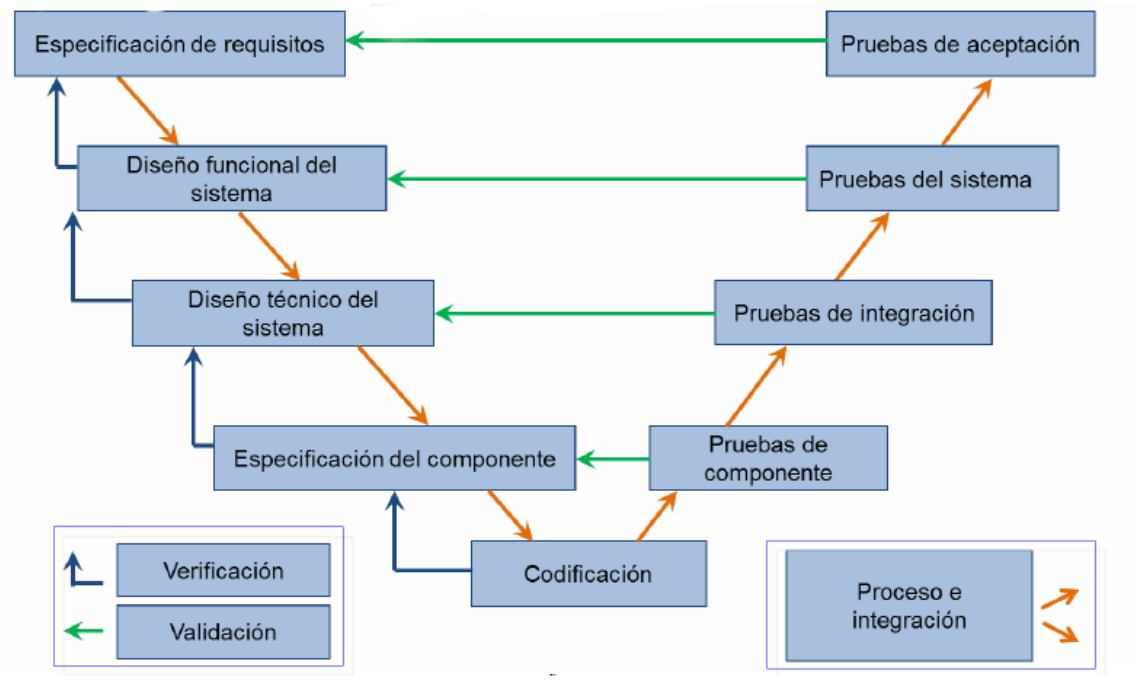


Figura 4: Modelo de desarrollo de software en "V"

La idea principal del modelo en V es, en general, que cada actividad de desarrollo tiene una actividad de pruebas correspondiente. La rama izquierda de la figura 4 representa el proceso de desarrollo durante el cual el sistema se está diseñando gradualmente y la codificación es el último paso. La rama derecha representa el proceso de pruebas, en los que los elementos del programa se integran sucesivamente para formar mayores subsistemas (integración), donde es probado. La integración y el proceso de pruebas finalizan al completarse las pruebas de aceptación de todo el sistema desarrollado según este proceso. El modelo en V requiere de la validación y la verificación. La verificación es el proceso de evaluar el software para determinar si los productos de una etapa de desarrollo en concreto satisfacen las condiciones impuestas al comienzo de la etapa. La validación es el proceso de evaluar el software durante o al final del proceso de desarrollo para determinar si este satisface los requisitos especificados (Patton, 2001).

Modelos de desarrollo iterativos – incrementales

También existen ciclos de vida iterativos e incrementales en donde pasamos por varias pequeñas fases del ciclo de vida independientes para el mismo proyecto, en vez de un solo calendario amplio de desarrollo de principio a fin.

La entrega se divide en incrementos o se estructura cada incremento al añadir nuevos requisitos: el incremento inicial contendrá la infraestructura necesaria para ofrecer soporte a la funcionalidad de estructura inicial. Se puede probar el incremento producido por una iteración en varios niveles como parte de su desarrollo.

Incrementos subsecuentes necesitarán pruebas para la nueva funcionalidad, las pruebas de regresión de la funcionalidad existente y las pruebas de integración de las partes nuevas y las existentes. Las pruebas de regresión cada vez son más importantes en todas las iteraciones después de la primera. Este ciclo de vida puede ofrecer una presencia de mercado temprana con una funcionalidad crítica, puede ser más fácil de gestionar porque la carga de trabajo se divide en partes más pequeñas y puede reducir la inversión inicial. A su vez, la presencia temprana del mercado significa que las pruebas de validación se llevarán a cabo en cada

incremento, logrando ofrecer una retroalimentación temprana en el valor del negocio y en la idoneidad del producto. Algunos de los ejemplos de modelos de ciclo de vida iterativos-incrementales son:

- Los prototipos.
- El desarrollo rápido de aplicaciones (RAD).
- El proceso unificado racional (RUP).
- Desarrollo ágil.

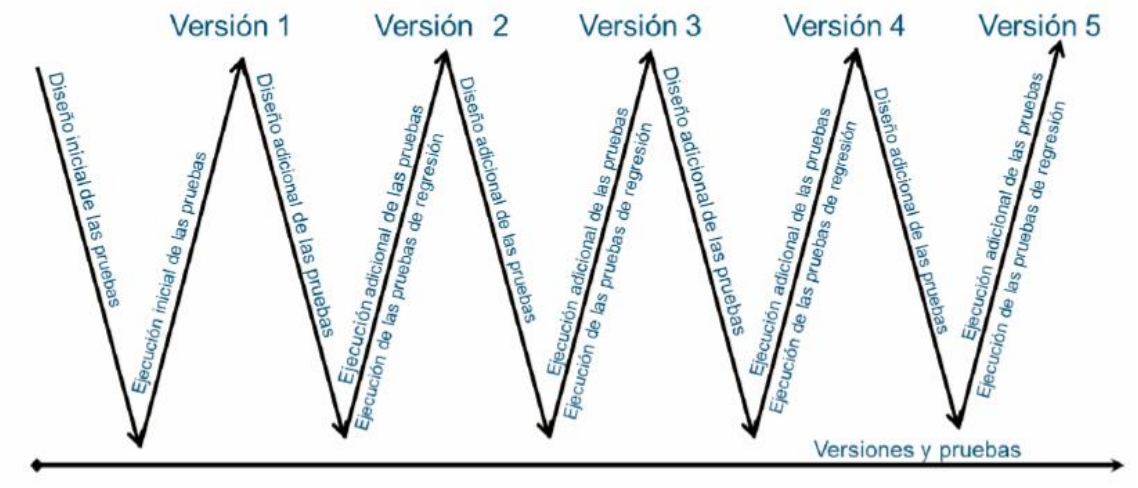


Figura 5: Ilustración sobre cómo se distribuyen las pruebas durante un ciclo de vida de un desarrollo iterativo-incremental

En cualquier modelo de ciclo de vida se dan varias características para que una prueba de software sea una buena prueba:

- Para cualquier actividad de desarrollo existe una actividad de pruebas correspondiente.
- Cada nivel de pruebas posee objetivos de prueba específicos para ese nivel.
- El análisis y diseño de las pruebas para un nivel de pruebas determinado debería comenzar durante la actividad de desarrollo correspondiente.
- Los probadores deben iniciar la revisión de documentos tan pronto como estén disponibles en los proyectos en el ciclo de vida de desarrollo.

Se pueden combinar o reorganizar los niveles de pruebas dependiendo de la naturaleza del proyecto o de la arquitectura del sistema. Por ejemplo, podemos hacer la integración de un producto de software de distribución masiva (COTS – *Commercial Of-The-Shelf*) dentro de cualquier otro sistema (ISTQB, 2018).

Niveles de pruebas

Para cada nivel de prueba, pueden identificarse los siguientes aspectos (ISTQB, 2018):

- Los objetivos genéricos.
- El producto utilizado para generar casos de prueba (la base de las pruebas).
- El objeto de la prueba (lo que se está probando).
- Los defectos o fallos típicos a ser encontrados.

- Requisitos del arnés de pruebas y el soporte de herramientas de pruebas.
- Enfoques específicos y responsabilidades.

El modelo en V puede dar la impresión de que las pruebas comienzan relativamente tarde, después de la codificación, pero no es así. Las pruebas de la rama de la derecha (véase figura 4) del modelo serán interpretadas como niveles de pruebas. Las actividades previas a la preparación de las pruebas (planificación y control de las pruebas, análisis y diseño de pruebas) comienza antes y de desarrolla en paralelo a las fases de desarrollo de la rama izquierda.

La diferenciación de los niveles de prueba en el modelo en V es más que una subdivisión temporal de las actividades de pruebas. Es un proceso de definición de modelos de niveles de abstracción de pruebas. Estos niveles son técnicamente muy diferentes e implican distintos métodos y herramientas, y personal con diferente conocimiento (ISTQB, 2018). Pasamos a describir uno a uno cada uno de esos niveles de prueba (ver figura 4 rama derecha).

Pruebas de componente

También conocido como pruebas de módulo o pruebas unitarias. Chequea aspectos internos de los componentes, estos son probados individualmente y aislados de los demás componentes del sistema. El aislamiento es necesario para evitar influencias externas sobre los componentes. Si las pruebas detectan un problema, éste será originado con toda seguridad por el componente probado. Éste puede ser también producto del ensamblaje de varios módulos más pequeños. De todas formas es crucial que el probador se centre en los aspectos internos y el comportamiento del componente. Las pruebas de componente no prueban la interacción con otros componentes vecinos.

Debido a que las pruebas de componente deben ser hechas de forma aislada al resto del sistema, se utilizan componentes software llamados "*stubs*" y controladores para reemplazar el software externo y simular de forma sencilla el interfaz entre componentes software. El *stub* es llamado desde el componente probado y el controlador llama al componente probado. Un *stub* es, en el contexto del testeo del software, un trozo de código usado como sustituto de alguna otra funcionalidad. Un *stub* puede simular el comportamiento de código existente (tal como un procedimiento en una máquina remota) o ser el sustituto temporal para un código aún no desarrollado.

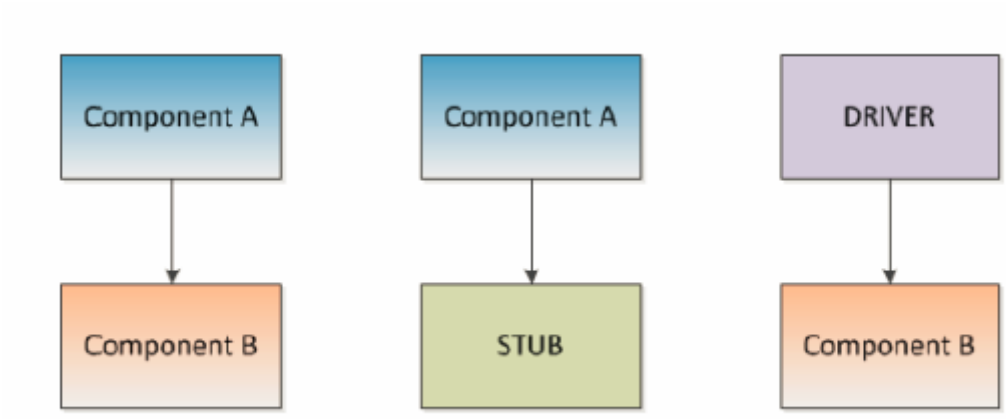


Figura 6: Representación de las pruebas de componente. Un *stub* es llamado desde el componente probado, y el controlador o driver llama al componente probado.

Las pruebas de componente tienen por objeto localizar defectos y comprobar el funcionamiento de módulos de software. Si se desarrollan casos de prueba de caja blanca o se mide el ámbito de las pruebas de caja blanca, el código fuente por si mismo puede ser parte de la base de las

pruebas. En cualquier caso, el comportamiento del componente debe ser comparado con las especificaciones del componente. Los objetos de prueba típicos son módulos o unidades de programas o clases, scripts de base de datos y otros componentes software.

Lo más importante de las pruebas de componentes es garantizar que un objeto de prueba concreto ejecuta toda su funcionalidad correcta y completamente de acuerdo con las especificaciones (prueba funcional y no funcional). Es también importante probar la robustez del módulo o unidad mediante pruebas negativas. Todas las características que tienen influencia crucial en su calidad y que no pueden ser probadas en niveles de pruebas más altos (o solo con un gasto mucho mayor) también deberían ser probadas:

- Eficiencia: uso de memoria, tiempo de proceso, tiempo de acceso a disco o a red, tiempo requerido para ejecutar las funciones y algoritmos del componente, etc.
- Mantenibilidad: estructura del código, modularidad, calidad de los comentarios en el código, adhesión a estándares y cumplimiento de los mismos, etc.

Los defectos de software típicos encontrados durante las pruebas de componentes suelen ser errores de cálculo, o lógica que está mal implementada. Para eso utilizamos las pruebas de caja negra y las pruebas de caja blanca. Un enfoque a seguir en las pruebas de componente es preparar y automatizar casos de prueba antes de la codificación (esto se denomina un primer enfoque o desarrollo guiado por pruebas). Esto es un enfoque altamente iterativo y está basado en la realización de ciclos de desarrollo de casos de prueba para construir después pequeñas partes de código (Patton, 2001).

Pruebas de integración

Las pruebas de integración suponen que los objetos de pruebas sometidos a ellas (componentes) ya han sido probado y los defectos encontrados, ya deberían haber sido corregidos. Estos componentes se agrupan para formar unidades estructurales y subsistemas más grandes. Esta conexión de componentes se llama integración y se hace por desarrolladores, evaluadores o equipos especiales de integración. Después de integrar los componentes, se deben comprobar a través de pruebas que todos los componentes colaboran correctamente. Por lo tanto, el objetivo de la prueba de integración es mostrar los fallos en las interfaces y en la interacción entre los componentes integrados.

Puede haber más de un nivel de pruebas de integración y puede ser llevado a cabo sobre los objetos de prueba de distinto tamaño de la siguiente forma. Las pruebas de integración de componentes prueban las interacciones entre los componentes de software y se realizan después de las pruebas de componentes (integración a pequeña escala). Las pruebas de integración de los sistemas prueban las interacciones entre los diferentes sistemas o entre el hardware y el software y se pueden hacer después de la prueba del sistema (integración a gran escala). Cuanto mayor sea el alcance de las pruebas más difícil será aislar los defectos encontrados.

Los objetos de prueba típicos son:

- Las interfaces
- Implementación de bases de datos de subsistemas.
- La infraestructura.

La base de pruebas compone:

- Requisitos de diseño de software y sistema.
- Arquitectura.
- Flujo de trabajo.

- Casos de uso.

Durante la integración, los componentes ya integrados pueden convertirse en objetos de prueba para otras pruebas de integración posteriores. Se pueden utilizar stubs o drivers dependiendo de la estrategia seleccionada para realizar la integración. Durante las pruebas de integración, se pueden utilizar herramientas llamadas “monitores” que se ejecutan de forma concurrente con el componente o sistema en pruebas y supervisa, registra y/o analiza el comportamiento de dicho componente o sistema (ISTQB, 2018).

El objetivo del nivel de prueba de la prueba de integración es poner de manifiesto los problemas de interfaz y de cooperación, así como los conflictos entre las partes integradas. Los problemas más difíciles de encontrar, sin embargo, se deben a la ejecución de las partes de los programas conectados. Éstos solo se pueden encontrar mediante pruebas dinámicas y son fallos en el intercambio de datos o en la comunicación entre los componentes (ISTQB, 2018):

- Un componente no transmite datos o son sintácticamente incorrectos. El componente receptor no puede funcionar o se bloquea.
- La comunicación funciona, pero los componentes implicados interpretan los datos recibidos de una manera diferente.
- Los datos se transmiten correctamente, pero se transmite en el momento equivocado, o llega tarde (problemas de sincronización), o los intervalos entre las transmisiones son demasiado cortos (problema de rendimiento, de carga, o de capacidad).

Existen dos tipos de estrategias en las pruebas de integración (ISTQB, 2018): las sistemáticas y las no sistemáticas. En el primer grupo tenemos subdivididas las que son basadas en la arquitectura, y las que son basadas en la funcionalidad. Pasaremos a describir cada una de ellas.

Estrategia sistemática basada en la arquitectura – Top Down

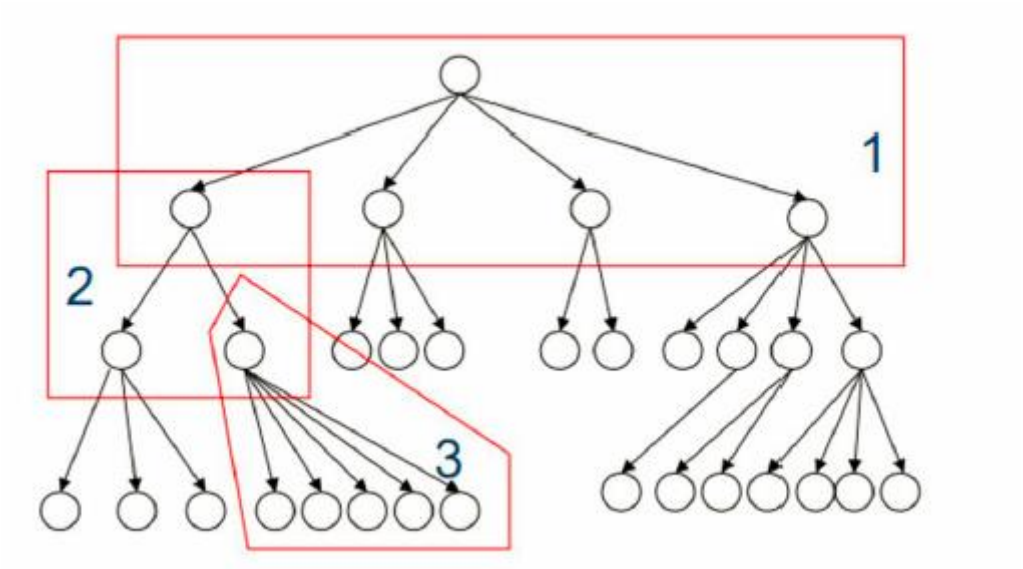


Figura 7: Estrategia sistemática basada en la arquitectura - Top Down

La ventaja es que no se necesitan controladores o en caso de necesitarse serían muy sencillos porque los componentes de nivel más alto que ya han sido probados sirven como parte principal de las pruebas de entorno. La desventaja es que los componentes de nivel inferior que aún no han sido integrados deben ser sustituidos por *stubs*. Esto puede ser un poco costoso.

Estrategia sistemática basada en la arquitectura – Bottom Up

La ventaja de esta estrategia es que no se requieren *stubs*. Como desventaja contamos que los componentes de nivel superior deben ser simulados por controladores. Ver figura 8.

Estrategia sistemática basada en la arquitectura – Ad Hoc

La ventaja de esta estrategia es que se ahorra mucho tiempo ya que cada componente está integrado tan pronto como sea posible en su entorno. La desventaja es que se requieren *stubs* y controladores.

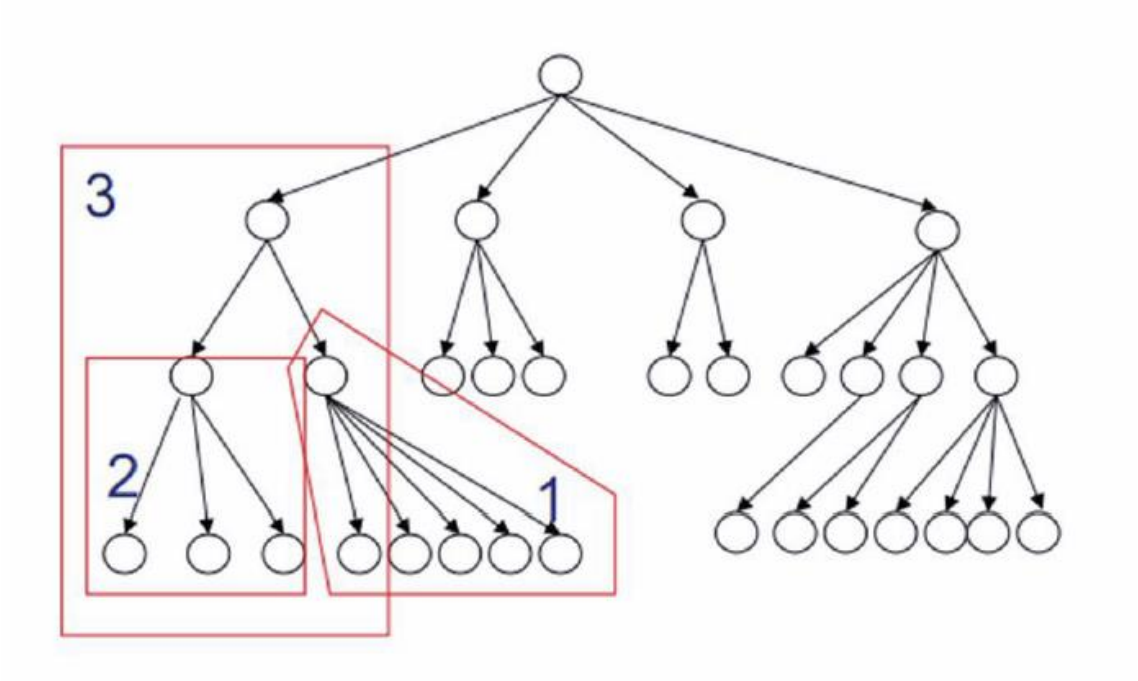


Figura 8: Estrategia sistemática basada en la arquitectura - Bottom Up

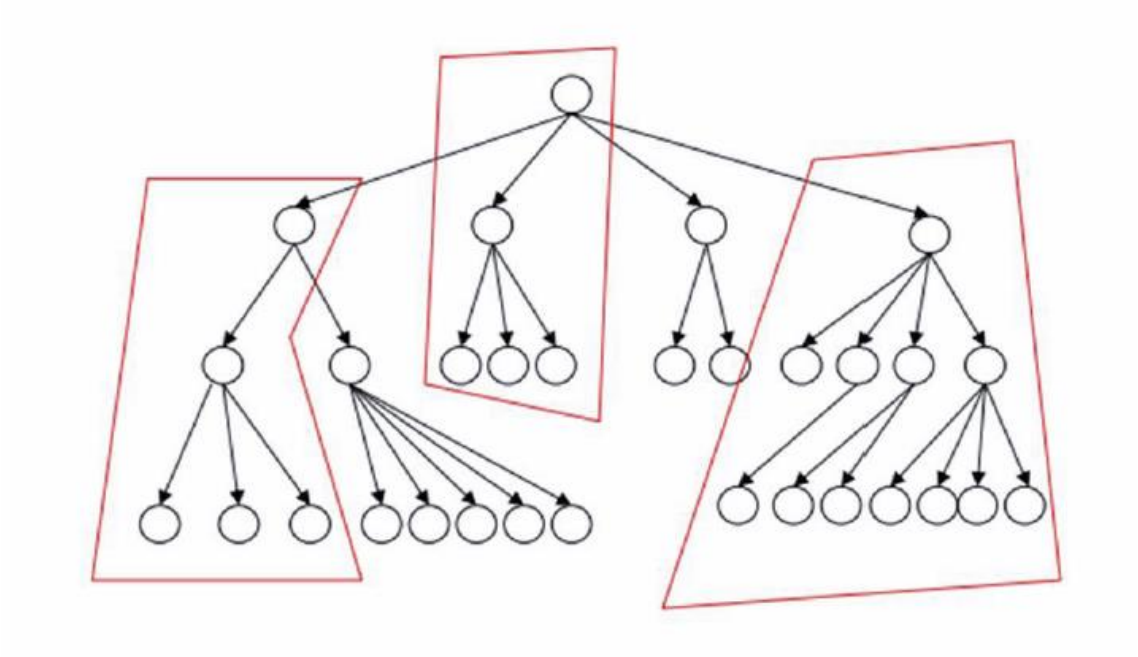


Figura 9: Estrategia sistemática basada en la arquitectura – Ad Hoc.

Estrategia sistemática basada en la arquitectura - Backbone

En esta estrategia se construye un esqueleto o columna vertebral en la que los componentes se integran gradualmente. La ventaja es que los componentes pueden ser integrados en cualquier orden. La desventaja es que la definición de la columna vertebral puede ser muy difícil y consume recursos. También se necesitan implementar stubs y controladores (ISTQB, 2018).

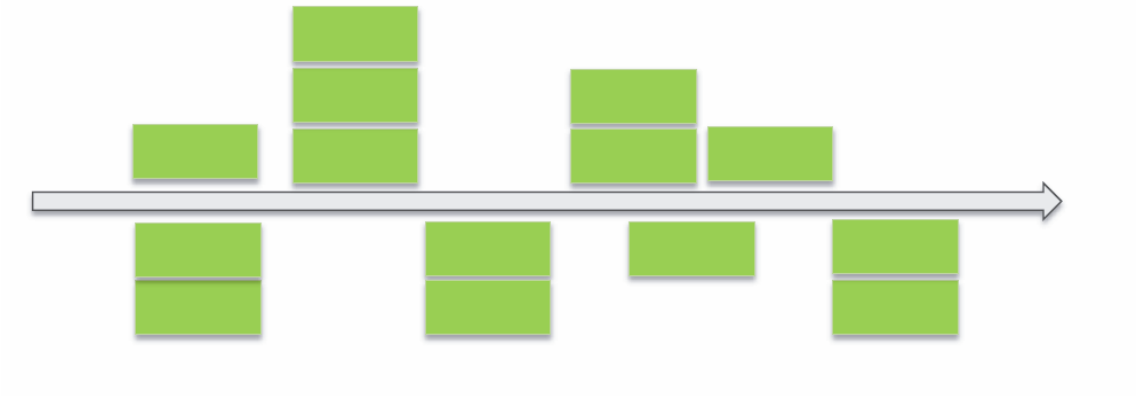


Figura 10: Estrategia sistemática basada en la arquitectura - Backbone

Estrategia no sistemática – Big Bang

La integración en "*big bang*" consiste en esperar hasta que se desarrollen todos los elementos de software y luego lanzar todas las pruebas a la vez. La ventaja es que no hace falta construir *stubs* ni controladores. La desventaja es que todos los fallos se producen al mismo tiempo, que es una estrategia complicada y que requiere mucho tiempo localizar y corregir defectos.

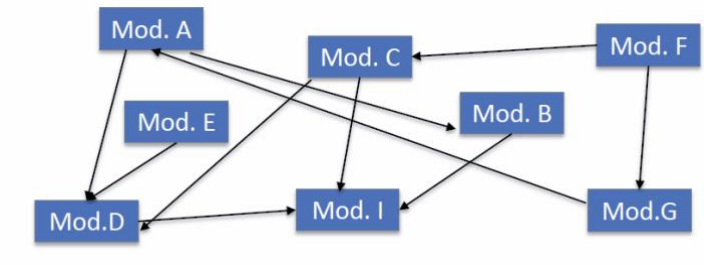


Figura 11: Estrategia no sistemática – Big Bang

Las pruebas de integración se planifican antes de construir los componentes o sistemas (para una construcción más eficiente del controlador y los componentes). El jefe de pruebas tiene que elegir la estrategia de integración para su optimización. La estrategia óptima (menor tiempo y menor coste) tendrá en cuenta los siguientes elementos (Patton, 2001):

- La arquitectura del sistema que determina cuantos y qué componentes de todo el sistema son necesarios y de qué manera dependen unos de otros.
- El plan de proyecto que determina en qué momento del transcurso del proyecto se debe desarrollar cada componente y cuándo deben estar listos para ser probados.
- El plan de prueba que determina qué aspectos del sistema han de ser probados, la intensidad con que esto se hará, y en qué nivel de prueba esto tiene que suceder.

Pruebas de sistema

El objetivo de las pruebas del sistema es comprobar si el producto integrado cumple con los requisitos funcionales y no funcionales especificados y las características de calidad de los datos.

Las pruebas del sistema siguen siendo necesarias después de la prueba de componentes e integración. Los niveles anteriores daban un punto de vista técnico del productor del software. Las pruebas de sistemas dan una perspectiva del cliente y del usuario. Muchas de las funciones y características del sistema resultan de la interacción de todos los componentes del sistema, por lo tanto, sólo son visibles a nivel de todo el sistema.

La base de pruebas para las pruebas de sistema se compone de toda la información y los documentos que describe el sistema en su conjunto (especificación de requisitos del sistema y software, especificación, casos de uso, especificación funcional, los informes de análisis de riesgos). Los objetos de prueba típicos son:

- Manuales de sistema, usuario y funcionamiento.
- Configuración del sistema.
- Datos de configuración.
- El sistema en sí.

Después de la finalización de las pruebas de integración, el sistema de software está completamente integrado y las pruebas de sistema ven el sistema como un todo. Esto se realiza en un entorno tan similar como sea posible al entorno operativo previsto. Los controladores y los *stubs* deben ser sustituidos por los productos de hardware y software instalados en un entorno de prueba independiente.

En un intento de ahorrar costes y esfuerzos se comete el error de no probar en un entorno independiente y la prueba de sistema se ejecuta en el entorno de producción, lo que puede generar posibles daños y dificultad para reproducir los resultados. Todas las pruebas de sistema a menudo son realizadas y controladas por un equipo de pruebas independiente (Whittaker, 2002).

En la práctica, contamos con varios problemas o riesgos en las pruebas de sistema como pueden ser:

- Requisitos del sistema incompletos: Cuando los probadores tienen que lidiar con los requisitos incompletos o indocumentados pueden solicitar pruebas exploratorias.
- Decisiones no tomadas: Los probadores responsables de las pruebas del sistema no sólo deben recoger información sobre los requisitos, sino que también deben tener en cuenta las decisiones tomadas durante las pruebas,
- Los proyectos pueden fallar si los requisitos no se especifican al principio del mismo.

Pruebas de aceptación

El objetivo de las pruebas de aceptación es establecer la confianza en el sistema, partes del sistema o características específicas no funcionales del sistema, como puede ser la usabilidad del sistema. Las pruebas de aceptación se centran a menudo en el tipo de validación de las pruebas, por el que estamos tratando de determinar si el sistema es apto para el propósito. Aunque valora la disposición del sistema para su despliegue, no es necesariamente el nivel final de la prueba (por ejemplo, una prueba de integración de sistemas a gran escala puede venir después de la aceptación). La base de pruebas de aceptación puede ser cualquier documento

que describe el sistema desde el punto de vista del usuario o del cliente (requisitos de usuario y del sistema, casos de uso, procesos de negocio e informes de análisis de riesgos).

Las pruebas de aceptación pueden ser ejecutadas dentro de los niveles más bajos de las pruebas o distribuirse en varios niveles de prueba (Whittaker, 2002):

- Un producto COTS (producto de software de distribución masiva) puede ser prueba de aceptación cuando se ha instalado o integrado.
- Las pruebas de aceptación de la usabilidad de un componente se pueden realizar durante las pruebas de componente.
- Las pruebas de aceptación de la nueva funcionalidad de un producto pueden venir antes de las pruebas del sistema (utilizando un prototipo).

Las formas habituales de las pruebas de aceptación son las siguientes (Whittaker, 2002):

- Pruebas de aceptación contractual y normativa.
- Pruebas de aceptación del usuario (UAT).
- Pruebas operativas (OAT).
- Pruebas alfa y beta (o pruebas de campo).

Las pruebas de aceptación contractual se realizan en función de criterios de aceptación contractual para la producción de software desarrollado a medida. Si el software de cliente específico se desarrolló, el cliente (en cooperación con el proveedor) llevará a cabo las pruebas de aceptación de acuerdo con el contrato. Ya que el proveedor puede haber entendido mal los criterios de aceptación, es de crucial importancia que los casos de prueba de aceptación sean diseñados por el cliente o, al menos, revisados profundamente por éste.

Las pruebas de aceptación normativas se llevan a cabo contra todas las normas que se deben cumplir, como las normas legales o normas de seguridad.

Las pruebas de aceptación del usuario (UAT) son la última fase de validación del usuario. Esta prueba es especialmente recomendable si el cliente y el usuario son diferentes entidades. Cada grupo de usuarios debe ser incluido en la aceptación del sistema: si solo un grupo de usuarios rechaza el sistema, esto puede conducir a problemas con la introducción del sistema. Para evitar desastre durante estas pruebas, es aconsejable presentar prototipos iniciales a los usuarios.

Las pruebas de aceptación operativas aseguran la aceptación del sistema por parte de los administradores de sistemas. Esto puede incluir: la prueba de copia de seguridad, restaurar los ciclos, la recuperación de desastres, gestión de usuarios, tareas de mantenimiento y controles de las vulnerabilidades de seguridad.

Las pruebas alfa y beta (de campo). Si el sistema ha sido desarrollado para el mercado masivo (por ejemplo COTS), las pruebas de usuarios o clientes individuales no son prácticas ni posibles en algunos casos. Se necesita información de los usuarios potenciales o existentes en su mercado antes de que el producto sea sacado a la venta. Por lo general, hay dos grandes tipos (Patton, 2001):

- Pruebas alfa: se llevan a cabo en las instalaciones del proveedor. Se invita a una sección transversal de los usuarios potenciales y a los miembros de la organización de productores a utilizar el sistema. Desarrolladores (o un grupo de pruebas independiente) se registran las actividades de los usuarios y se analiza lo que han hecho.
- Pruebas beta o de campo: se llevan a cabo en las instalaciones del cliente. El sistema se envía a una muestra representativa de los usuarios que tienen que instalarlo y usarlo bajo condiciones reales de trabajo. Los usuarios envían los registros de incidencias con el sistema a la organización de desarrollo para que se reparen los defectos.

Tipos de prueba

Pruebas funcionales

Las funciones de un sistema (o componente) son “lo que hace un sistema”. Esto se describe habitualmente en una especificación de requisitos, una especificación funcional, o en casos de uso. Puede haber algunas funciones que se asumen que deben proveerse, no están documentadas y son también parte de los requisitos de un sistema aunque es difícil probar contra unos requisitos no documentados o implícitos.

Las pruebas funcionales se basan en estas funcionalidades, descritas en los documentos o entendidas por los probadores y se pueden realizar en todos los niveles de prueba. Las pruebas funcionales consideran el comportamiento especificado y, a menudo se las conoce también como pruebas de caja negra (lo que no es del todo cierto, ya que las pruebas de caja negra también incluyen pruebas no funcionales). Más adelante se describirán este tipo de pruebas.

Las pruebas funcionales pueden realizarse desde varias perspectivas:

- Las técnicas utilizadas para las pruebas funcionales basadas en requisitos: se utiliza una especificación de los requisitos funcionales para el sistema como base para el diseño de las pruebas.
- Pruebas basadas en procesos de negocio: se utiliza el conocimiento de los procesos de negocio. Los procesos de negocio describen los escenarios involucrados en el uso del día a día del negocio del sistema.

Las técnicas utilizadas para las pruebas funcionales son a menudo basadas en la especificación, u otras técnicas como las basadas en la experiencia. Las condiciones de prueba y casos de prueba se derivan de la funcionalidad del componente del sistema (Elfriede, Thom & Bernie, 2009).

Pruebas no funcionales

Los requisitos no funcionales no describen las funcionalidades, sino los atributos del comportamiento funcional, o los atributos del sistema en su totalidad: “¿cómo de bueno es?” o “¿cómo funciona?” el sistema. ¿Con qué calidad lleva a cabo su función?. Estas pruebas, como las funcionales, se llevan a cabo en todos los niveles de prueba. Tienen en cuenta el comportamiento externo del software y, en la mayoría de los casos, utilizan técnicas de diseño de caja negra. Existen distintos modelos que tratan de describir la calidad del software en un conjunto de características y sub-características relacionadas, como pueden ser la norma ISO 9126 e ISO 25010 (Elfriede et al., 2009).

Figura 12: Estándar internacional ISO 9126 para la evaluación de la calidad del software

Características del estándar ISO 9126:

- Funcionalidad: Conjunto de atributos que actúan sobre la existencia de un conjunto de funcionalidades y sus propiedades específicas. Las funcionalidades son las que satisfacen las necesidades manifestadas o implícitas.
- Fiabilidad: Conjunto de atributos que influyen en la capacidad del software para mantener su nivel de rendimiento bajo las condiciones establecidas por un periodo de tiempo determinado.
- Usabilidad: Conjunto de atributos que actúan sobre el esfuerzo necesario para su uso, y en la valoración individual de tal uso, por un conjunto de usuarios declarados o implícitos.
- Eficiencia: Conjunto de atributos que actúan sobre la relación entre el nivel de rendimiento del software y la cantidad de recursos utilizados, en las condiciones establecidas.
- Capacidad de mantenimiento: Conjunto de atributos que actúan sobre el esfuerzo necesario para hacer modificaciones especificadas.
- Portabilidad: Conjunto de atributos que influyen en la capacidad del software para ser transferido de un entorno a otro.

Pruebas estructurales

El objetivo de estas pruebas es la estructura del sistema o componente (se refiere a ellas a menudo como pruebas de caja blanca). Las pruebas estructurales se utilizan más a menudo como una forma de medir la minuciosidad de las pruebas a través de la cobertura de un conjunto de elementos estructurales o elementos de cobertura. Puede ocurrir a cualquier nivel de la prueba, aunque tiende a ser aplicado sobre todo en los componentes y la integración y, en general, es menos probable su uso en niveles posteriores de prueba. A nivel de componente aparecen a modo de declaraciones o decisiones. A nivel de integración de componente, pueden estar basadas en la arquitectura del sistema. A nivel de integración de sistemas / aceptación, pueden estar basadas en un modelo de negocio o en la estructura del menú.

Las técnicas de pruebas estructurales de caja blanca es mejor usarlas después de las técnicas de caja negra, con el fin de ayudar a medir el rigor de las pruebas a través de la evaluación de la cobertura de un tipo de estructura. A nivel de componente, y a menor medida en las pruebas de integración de componentes, hay un buen soporte de herramientas para medir la cobertura de código. Estas herramientas se las conoce como herramientas de cobertura de código. La cobertura de código es el porcentaje de elementos de código (por ejemplo, las sentencias

ejecutables o resultados de la decisión) que se han ejecutado respecto a un conjunto de pruebas. Si la cobertura de código no es del 100%, se pueden necesitar pruebas adicionales para cubrir aquellas partes que no han sido ejecutadas todavía, dependiendo de los criterios de salida (Elfriede et al., 2009).

Repetición de pruebas y pruebas de regresión

Debido a modificaciones en el software, como puede ser una solución de defectos, o en el entorno (actividades de mantenimiento), siempre es necesario comprobar que todo está correcto. Para ello utilizamos las pruebas de confirmación y las pruebas de regresión. Estas pruebas asociadas a los cambios se pueden realizar en todos los niveles y pueden involucrar pruebas funcionales, no funcionales y estructurales.

Las pruebas de confirmación deben demostrar que los defectos anteriores han sido realmente reparados. Es la repetición de la prueba que mostró el fallo. En este tipo de pruebas es importante asegurarse de que la prueba se ejecuta exactamente de la misma manera que cuando se generó el fallo. Utilizando las mismas entradas, datos y entorno.

Las pruebas de regresión son la prueba reiterada de un programa ya probado, después de haber sido modificado. Con vistas a localizar defectos surgidos o no descubiertos como resultado del cambio o cambios. El objetivo de las pruebas de regresión es verificar que las modificaciones en el software o el entorno no han causado efectos secundarios adversos no intencionados y que el sistema sigue cumpliendo sus requisitos. Todos los casos de prueba en un conjunto de pruebas de regresión deberían ser ejecutados cada vez que se produce una nueva versión del software, esto los hace candidatos ideales para la automatización. El alcance de las pruebas de regresión va en función de los riesgos de los cambios efectuados. Las pruebas de regresión se pueden realizar en todos los niveles de pruebas, e incluye pruebas funcionales, no funcionales y estructurales (Elfriede et al., 2009).

Pruebas de mantenimiento

Una vez desplegado, un sistema de software suele estar en servicio durante años o décadas. Durante este tiempo, el sistema, sus datos de configuración o su entorno son objeto de frecuentes modificaciones o ampliaciones. Las pruebas que se ejecutan durante esta fase del ciclo de vida se llaman pruebas de mantenimiento.

Habitualmente estas pruebas tienen dos partes. La prueba de los cambios (prueba de lo modificado) y las pruebas de regresión, para mostrar que el resto del sistema no ha sido afectado por el mantenimiento. Se debe realizar un análisis de impacto, para determinar qué partes del sistema podrían haber sido afectadas involuntariamente. No se suelen repetir todas las pruebas de regresión existentes. Si se mantienen las especificaciones de las pruebas del desarrollo original del sistema, entonces es suficiente con volver a utilizarlas para las pruebas de regresión y para adaptarlas a los cambios en el sistema. Una ampliación o mejora del sistema puede

significar que se han especificado nuevas áreas (o especificaciones). En este caso se elaborarían nuevas pruebas al igual que para el desarrollo.

Las pruebas de mantenimiento se realizan en un sistema operativo existente, y se activan a partir de modificaciones, migración, o la retirada del sistema:

- Modificaciones: se incluyen modificaciones de mejora planificadas (debido a diferentes versiones), cambios correctivos, de emergencia, del entorno, etc...
- Migración: desde una plataforma a otra. Se deben repetir las pruebas operativas dentro del nuevo entorno. Si se necesita convertir o migrar datos, se debería incluir la corrección de dicho tratamiento en la prueba.
- Retirada: debe incluir las pruebas de los archivos de datos o la migración de datos en el sistema futuro.

Las modificaciones suelen ser la parte principal de las pruebas de mantenimiento para muchas organizaciones. Desde el punto de vista de las pruebas, hay dos tipos de modificaciones:

- Previstas: en las que se pueden plantear las pruebas. A su vez, éstas pueden ser:
    - Mejoras: adaptando el software a los deseos del usuario, por ejemplo, mediante el suministro de nuevas funciones o la mejora del rendimiento.
    - Adaptaciones: adaptación del software a los cambios del entorno, tales como un nuevo hardware, software o nuevos sistemas de nueva legislación.
    - Correctivas: corrección de defectos.
- De emergencia ad-hoc: en las que las pruebas no se pueden planificar en absoluto. Son correcciones de emergencia (o correcciones de proceso) que se ocupan de los defectos que requieren una solución inmediata (por ejemplo, una red que se cae con cientos de usuarios en línea, vulnerabilidades del sistema). Este tipo de pruebas de mantenimiento se presenta como los “primeros auxilios” o parche y, en una etapa posterior del proceso de prueba estándar, se establecerá una solución robusta, que será probada y documentada. Si se cambie el entorno, aunque no cambie el sistema, las pruebas de mantenimiento serían necesarias.

El ámbito de aplicación de las pruebas de mantenimiento está relacionado con el riesgo del cambio, el tamaño del sistema existente y el tamaño del cambio. En función de los cambios, las pruebas de mantenimiento se pueden realizar en cualquiera o en todos los niveles de prueba, y para alguno o todos los tipos de prueba (Patton, 2001).

Técnicas estáticas

En la definición de las pruebas mencioné dos enfoques prácticos: las pruebas estáticas y las pruebas dinámicas. Las técnicas de pruebas estáticas se basan en el examen manual (revisión) y el análisis automático (análisis estático) del código u otra documentación del producto, sin ejecución de código. En principio, todos los productos software pueden probarse mediante revisiones: especificaciones de requisitos, documentos de diseño, planes de pruebas, manuales de usuario, código fuente, etc… Las técnicas dinámicas y estáticas son métodos complementarios ya que tienden a encontrar distintos tipos de defectos. Los defectos que son fáciles de encontrar durante las pruebas estáticas son:

- Desviaciones de las normas.

- Defectos de requisitos.
- Defectos de diseño.
- Mantenibilidad insuficiente.
- Especificaciones de interfaz incorrectas.

Comparadas con las pruebas dinámicas, las técnicas estáticas encuentran las causas de fallos (los defectos) más que los propios fallos. Las revisiones a menudo representan hitos del proyecto, y apoyan el establecimiento de una línea base del producto software. Los exámenes de verificación al final de una fase en el Modelo-V en general utilizan revisiones (comúnmente conocidos como revisiones de salida). Idealmente las revisiones deben realizarse lo antes posible, para encontrar pronto inconsistencias y errores. El proceso de eliminar defectos conduce a una mejor calidad de los documentos y a una influencia positiva en el proceso completo de desarrollo. Adicionalmente a la reducción de defectos, las revisiones tienen también los siguientes efectos positivos:

- Eliminación de defectos más barata: esto implica mayor productividad en el desarrollo.
- Tiempo de desarrollo más corto.
- Si los defectos se reconocen pronto, el coste y el tiempo necesarios para la ejecución de las pruebas dinámicas se reducen.
- A causa del menor número de defectos, cabe esperar una reducción de costes en el ciclo de vida completo del producto.
- Tasa de errores reducida durante la operación del sistema.
- Como las revisiones se hacen utilizando un equipo de personas, éstas conducen al aprendizaje mutuo mejorando así sus métodos de trabajo.
- Como en una revisión están implicadas varias personas, se requiere una descripción de los hechos clara y comprensible.
- El equipo se siente responsable de la calidad del objeto examinado y el grupo gana una comprensión común del mismo.

En algunos casos, los clientes/usuarios participan en la reunión de revisión y proporcionan retroalimentación al equipo de desarrollo, de modo que las revisiones son también un medio para la comunicación con los clientes/usuarios. Los costes causados por las revisiones se estiman en un 10-15% del presupuesto de desarrollo. Los costes incluyen las actividades del proceso de revisión propiamente dicho, el análisis de los datos de las revisiones, y el esfuerzo de su implementación para la mejora del proceso. El ahorro se estima en alrededor del 15-25% (el esfuerzo extra para las revisiones propiamente dichas está incluido en el cálculo). Si las revisiones se usan sistemáticamente y se realizan de forma eficiente, pueden encontrarse y repararse más del 70% de los defectos de un documento antes de ser heredados inadvertidamente por los siguientes pasos del proceso (Elfriede et al., 2009).

Proceso de revisión

Las revisiones varían desde muy informales a las más formales. La inspección es la técnica de revisión más formal, por ejemplo. La formalidad del proceso de revisión se relaciona con factores como la madurez del proceso de desarrollo, requisitos legales o regulatorios, o la necesidad de un registro de auditoría. Las revisiones informales se aplican en varios momentos durante las primeras etapas del ciclo de vida de un documento. Un equipo de dos personas puede conducir una revisión informal, ya que el autor puede pedir a un colega que revise un documento o el

código. En etapas posteriores estas revisiones implican a más personas y una reunión. Las personas implicadas son usualmente iguales del autor, por eso las revisiones se llaman revisiones entre pares.

Las revisiones formales siguen un proceso formal. El proceso de revisión formal típico consta de seis pasos principales (ISTQB, 2018):

1. Planificación
2. Arranque o lanzamiento
3. Preparación individual.
4. Reunión de revisión.
5. Adaptar el trabajo - repetición del trabajo (reconstrucción).
6. Seguimiento

Planificación

Se establece la definición de los criterios de revisión y de entrada y salida (para los tipos de revisión más formales). Los criterios de entrada se comprueban para asegurar que el tiempo de los revisores no se desperdicia en un documento que no está todavía listo para la revisión. Los criterios de salida permiten determinar cuándo terminar la revisión. Otro de los objetivos que toma esta etapa es la de seleccionar qué partes de los documentos revisar. La mente humana puede comprender sólo un número limitado de páginas a la vez; por tanto, el número de páginas no puede ser demasiado alto. En la etapa de planificación también se hace una selección del personal y asignación de roles. Estos roles ayudan a los revisores a enfocarse en ciertos tipos de defectos durante la comprobación. Por último, se realiza una comprobación de los criterios de entrada para los tipos de revisión más formales. Si los criterios de entrada no se cumplen, la revisión se cancela; así la organización ahorra tiempo que de otro modo se desperdiciaría revisando material que no puede ser lo suficientemente bueno.

Arranque o Inicio

Se encarga de las siguientes tareas:

- Distribución de documentos.
- Explicación de objetivos, proceso y documentos a los participantes. Durante el arranque los revisores reciben: una explicación sobre los objetivos de la revisión y los documentos, las relaciones entre el documento bajo revisión y los demás documentos (fuentes), las asignaciones de roles, tasa de comprobación, páginas a comprobar, cambios al proceso y se distribuye el documento bajo revisión, los documentos fuentes y otra documentación que esté relacionada.

En general la reunión de arranque se recomienda encarecidamente pues ejerce un fuerte efecto positivo en la motivación de los revisores y, por tanto, en la efectividad del proceso de revisión.

Preparación individual

Se realiza la preparación de la reunión de revisión mediante la revisión individual de los documentos. Se señalan los defectos potenciales, las preguntas y los comentarios. Una reunión de revisión exitosa solo es posible con preparación adecuada. Los revisores deben estudiar intensivamente y el objeto revisado y comprobarlo contra los documentos dados como base para él. Los participantes individuales identifican defectos, preguntas y comentarios, de acuerdo con su conocimiento del documento y su rol. Todas las cuestiones se registran, preferiblemente

usando un formulario de registro. Utilizar listas de comprobación en esta fase puede hacer las revisiones más efectivas y eficientes.

Reunión de revisión

Es una discusión o registro, con minutas o resultados documentados (para los tipos de revisión más formales). Se encarga de señalar defectos, de hacer recomendaciones sobre como manejarlos, y tomar decisiones sobre los mismos. También se encarga de evaluar y registrar cuestiones durante las reuniones físicas o registrar las comunicaciones electrónicas del grupo (Whittaker, 2002).

La reunión de revisión típicamente consta de los siguientes elementos: fase de registro, fase de discusión y fase de decisión. Durante la fase de registro, las cuestiones que se han identificado durante la preparación se mencionan página por página, revisor por revisor y se registran. Para asegurar el progreso y la eficiencia, no se permite discusión real durante esta fase. Durante la fase de discusión se tratan las cuestiones clasificadas como elementos de discusión. Los participantes pueden tomar parte en la discusión planteando sus comentarios y razonando. Todos los elementos discutidos o bien deben tener un resultado final de la discusión, o deben ser definidos como un punto de acción si la discusión no puede resolverse durante la reunión. El resultado de las discusiones se registra para futuras referencias. Al final de la reunión, durante la fase de decisión, debe tomarse una decisión sobre el documento bajo revisión, en ocasiones basándose en los criterios formales de salida. Si no se encuentran elementos de discusión o estos no derivan en defectos, el documento podrá abandonar el proceso de revisión.

Adaptar/repetir el trabajo

Se trata de corregir los defectos encontrados basándose en los defectos detectados. El autor mejorará paso a paso el documento bajo revisión. Otra función es la de registrar el estado actualizado de los defectos (para los tipos de revisión formales). Los cambios hechos en el documento deberán ser fáciles de identificar durante el seguimiento. Por lo tanto, el autor deberá indicar donde se han realizado dichos cambios. Esto se puede hacer mediante el control de cambios de una herramienta de proceso de textos.

Seguimiento

La función de esta actividad es comprobar que los defectos se han considerado para asegurar que se han tomado acciones satisfactorias sobre todos los defectos, sugerencias de mejora del proceso y peticiones de cambios. También se encarga de recoger métricas para controlar y optimizar el proceso de revisión. El moderador recoge diversas métricas en cada paso del proceso. Una vez finalizado todo el seguimiento, se comprueban los criterios de salida (en los tipos de revisión más formales) (Patton, 2001).

Tipos de revisiones

Un único producto software puede ser objeto de una o más revisiones. Si se utiliza más de un tipo de revisión, el orden puede variar. De menos formal a más formal (Elfriede et al., 2009):

- **Revisión informal**: No tiene un proceso formal. Puede tomar la forma de programación por pares o revisión de diseño o código por un responsable técnico. Los resultados pueden documentarse. Su utilidad varía dependiendo de los revisores. Su propósito es tener una forma barata de obtener algún beneficio.
- **Revisión guiada**: Es una reunión dirigida por el autor. Puede tomar la forma de escenarios, simulacros o participación de grupos de pares. Son sesiones sin un final definido. El escriba es opcional y distinto del autor. En la práctica puede variar desde bastante informal a muy formal. Los propósitos principales son: aprender, entender y encontrar defectos. Debido a que el autor dirige la reunión, tiene gran influencia. Esto puede tener un efecto pernicioso en el resultado si el autor no permite la discusión de las partes críticas del objeto revisado. Este proceso es adecuado para equipos pequeños (5 a 10 personas) y consume poco esfuerzo porque no hay preparación ni seguimiento. Puede servir para comprobar documentos no críticos. Se pueden utilizar tutoriales para el control de documentos “no críticos”.
- **Revisión técnica**: Es un proceso definido y documentado de detección de defectos que incluye pares y expertos técnicos con participación opcional de la dirección. Puede consistir en una revisión por pares sin participación de la dirección. Idealmente dirigida por un moderador formado diferente al autor. Consta de una preparación previa por los revisores y un uso opcional de listas de comprobación. Se desarrolla un informe de revisión incluyendo la lista de puntos encontrados, la decisión sobre si el producto cumple con sus requisitos y, en su caso, recomendaciones relacionadas con los puntos encontrados. En la práctica puede varias de bastante informal a muy formal. Sus principales objetivos son: discutir, tomar decisiones, encontrar defectos, evaluar alternativas, resolver problemas técnicos y comprobar el cumplimiento de especificaciones, los planes, la normativa y los estándares.
- **Inspección**: Es dirigida por un moderador formado diferente al autor. Generalmente consiste en un examen por pares. Tiene los roles bien definidos. Incluye una recolección de métricas. Es un proceso formal basado en reglas y listas de comprobación. Los criterios de entrada y de salida son especificados para la aceptación del producto. Es necesaria una preparación previa a la reunión. Se incluye un informe de inspección incluyendo lista de puntos encontrados. Proceso de seguimiento formal (con componentes opcionales de mejora de procesos). Lector opcional. Su principal propósito es la identificación de defectos. Dependiendo de la organización y los objetivos del proyecto, las inspecciones pueden equilibrarse para obtener diversos objetivos.

Entre los factores de éxito de las revisiones se encuentran los siguientes (Elfriede et al., 2009):

- Cada revisión tiene claros objetivos predefinidos.
- El personal de pruebas son revisores valiosos que contribuyen a la revisión y también aprenden sobre el producto, lo que les permite preparar las pruebas antes.
- Los defectos encontrados son bienvenidos y expresados objetivamente.
- Se aplican revisiones técnicas adecuadas a los objetivos y al tipo y nivel de los productos software y los revisores.
- Se usan roles y listas de comprobación cuando es apropiado para aumentar la efectividad de la identificación de defectos.
- Se proporciona formación en técnicas de revisión, especialmente las más formales, como la inspección.
- La dirección respalda la implantación de un proceso de revisión, por ejemplo, asignado el tiempo necesario a las actividades de revisión en los planes de proyecto).
- Se hace énfasis en el aprendizaje y la mejora de procesos.

- Está implicada la gente adecuada para los objetivos de la revisión.
- La revisión discurre en un ambiente de confianza; el resultado no se utiliza para la evaluación de los participantes.
- Se gestionan los asuntos personales y los aspectos psicológicos.

Roles y responsabilidades

Una revisión formal típica incluye los siguientes roles (Patton, 2001):

- **Manager**: Decide sobre la ejecución de las revisiones, asigna tiempos en los planes de proyecto, y determina si se han conseguido los objetivos del proceso de revisión.
- **Moderador**: Lidera la revisión del documento con tareas relativas a la revisión, planificación y preparación, asegurando que la revisión se lleva a cabo de forma ordenada y cumple sus objetivos, recolección de datos y envío del informe de revisión.
- **Autor**: Es el responsable de que el objeto revisado cumpla sus criterios de entrada, contribuir a la revisión en base a su especial conocimiento y comprensión del documento, y de realizar cualquier trabajo necesario para que el objeto revisado cumpla sus criterios de salida.
- **Revisores**: son personas con experiencia específica en el ámbito técnico que deben comprobar el objeto revisado tras la preparación individual. Deben identificar y describir los problemas en el objeto revisado, representando diferentes puntos de vista.
- **Escriba** o registrador: Debe documentar los puntos encontrados por el equipo de revisión de forma concisa y precisa, capturando la esencia de la discusión.

Técnicas de diseño de pruebas (dinámicas)

El proceso de desarrollo de pruebas puede variar en la práctica desde muy informal (poca o ninguna documentación) a muy formal (bien documentado). El nivel de formalidad dependerá del contexto de las pruebas, incluyendo la madurez de las pruebas y de los procesos de desarrollo, las limitaciones de tiempo, la seguridad o los requisitos de seguridad o normativos, y las personas implicadas.

En un proceso de desarrollo de pruebas formales se pueden identificar tres pasos principales (Jorgensen, 2013):

1. Análisis de las pruebas: identificar las condiciones de prueba.
2. Diseño de las pruebas: especificar los casos de prueba.
3. Implementación de las pruebas: especificar el procedimiento de las pruebas.

Análisis de pruebas: El análisis de pruebas es el proceso de examinar las bases de las pruebas para obtener información de las pruebas. Desde una perspectiva de prueba, miramos las bases de las pruebas para comprobar que se puede probar. Una condición de prueba es simplemente algo que se puede probar. La definición de condiciones de pruebas es: un elemento o suceso de un componente o sistema que puede ser verificado por uno o más casos de pruebas, como por ejemplo, una función, transacción, característica, atributo de calidad, o elemento estructural. Las

condiciones de prueba se deberían de poder enlazar con su origen en las bases de las pruebas (trazabilidad). La trazabilidad debe ser horizontal a través de la documentación de las pruebas para un nivel de pruebas dado. La trazabilidad permite tanto determinar la cobertura de los requisitos para un grupo de pruebas como el impacto de análisis cuando los requisitos cambian. También es importante priorizar las condiciones de pruebas identificadas, antes de invertir mucho tiempo en el diseño de los casos de pruebas basados en ellas. Durante el análisis de las pruebas, el enfoque detallado de las pruebas es implementado para seleccionar las técnicas de diseño de pruebas para utilizar sobre la base de los riesgos identificados (Jorgensen, 2013). Las condiciones de pruebas están detalladas en el documento IEEE STD 829-1998 denominado "Norma para la documentación de pruebas de software."

Diseño de pruebas: Durante el diseño de pruebas, al crear casos de prueba para una o más condiciones de pruebas dadas, pueden ser aplicadas diversas técnicas de diseño (para producir casos de prueba de alto nivel o casos de prueba de bajo nivel). Los valores de entrada y condiciones previas de ejecución deberán ser producidas para definir los casos de prueba. Los resultados esperados deberían de ser también creados como parte de la especificación de un caso de prueba e incluir resultados, cambios en los datos y estados, y cualquier otra consecuencia de las pruebas (oráculo de pruebas). La "Norma para la documentación de pruebas de software" (IEEE STD 829-1998) describe las especificaciones de los casos de prueba. Un caso de prueba es un conjunto de valores de entrada, condiciones previas de ejecución, resultados esperados y condiciones posteriores de ejecución, desarrollados para un objetivo particular o condición de prueba, tales como ejecutar una ruta específica de programa o para verificar el cumplimiento de un requisito específico (Jorgensen, 2013).

Implementación de las pruebas: Durante la especificación de las pruebas, los casos de pruebas lógicos son transformados en casos de pruebas concretos. El siguiente paso es priorizar y organizar los casos de prueba en grupos de forma sensata para su ejecución y especificar los pasos secuenciales necesarios de realizar para la ejecución. Un ejemplo de procedimiento de prueba automatizado es especificar en un guion de pruebas la secuencia de acciones necesarias para ejecutar una prueba mediante una herramienta de ejecución de pruebas. Los diversos procedimientos de pruebas y los scripts de prueba automatizados son posteriormente agrupados en un esquema de ejecución de pruebas que define el orden en el que los diferentes procedimientos de prueba y, posiblemente los scripts de prueba automatizados, son ejecutados (cuándo un script debería ser ejecutado y por quién). La especificación de procedimiento de prueba es un documento que especifica una secuencia de acciones para la ejecución de una prueba (Jorgensen, 2013).

Categorías de técnicas de diseño de pruebas

El propósito de una técnica de diseño de prueba es identificar las condiciones de prueba, los casos de prueba y los datos de prueba. Haré referencia a tres tipos de técnicas de diseño de pruebas:

1. **Técnicas basadas en la especificación (caja negra)**: Son comúnmente conocidas como pruebas de caja negra o técnicas de prueba de entrada/salida porque consideran el software como una caja negra con entradas y salidas. Estas técnicas no tienen en cuenta la estructura interna del objeto de prueba: los casos de prueba se derivan de la especificación. Una prueba con todas las posibles combinaciones de entrada de datos sería una prueba completa, pero esto no es realista si consideramos el gran número de

combinaciones. Por consiguiente, las técnicas de diseño de pruebas ayudan a hacer una selección razonable de todos los posibles casos de prueba.
Estas técnicas se aplican tanto a las pruebas funcionales como a las no funcionales. Las pruebas funcionales están relacionadas con lo que el sistema hace, sus características o funciones. Las pruebas no funcionales están relacionadas con examinar cómo de bien hace algo el sistema, en lugar de centrarse en el qué hace. El problema a resolver, el software o sus componentes, se especifica a partir de modelos, formales o informales. Los casos de prueba pueden obtenerse sistemáticamente a partir de estos modelos (Patton, 2001).

2. **Técnicas basadas en la estructura (caja blanca)**: Son también conocidas como pruebas de caja blanca o pruebas basadas en el código porque requieren saber cómo se implementa el software. Estas técnicas usan la estructura interna del objeto bajo prueba para obtener los casos de prueba, independientemente de la funcionalidad del software. La idea general de las técnicas de caja blanca es ejecutar al menos una vez cada parte del código del objeto de la prueba. Se identifican casos de prueba orientados al flujo, analizando la lógica del programa y después ejecutándolos.
   Características comunes de las técnicas basadas en la estructura son:
   - Se debe recoger información de cómo debe utilizarse el software para los casos de prueba existentes, y puede obtenerse otros casos de manera sistemática para aumentar la cobertura.
   - Puede medirse el alcance de la cobertura del software para los casos de prueba existentes, y pueden obtenerse otros casos de manera sistemática para aumentar la cobertura.

   El objetivo principal de la técnica es entonces lograr una cobertura previamente definida de los estados durante la prueba, por ejemplo, para ejecutar todos los estados posibles en el programa. El enfoque del estudio de una técnica de caja blanca puede ser, por ejemplo, las declaraciones del objeto de prueba (Jorgensen, 2013).

3. **Técnicas basadas en la experiencia**: Las técnicas basadas en la experiencia confían en el conocimiento y la experiencia del personal para obtener los casos de prueba. El conocimiento de los probadores, desarrolladores, usuarios y otros implicados sobre el software, su uso y su entorno son una fuente de información. El conocimiento y la experiencia del personal técnico y de negocio es importante, porque dan diferentes perspectivas del análisis de las pruebas y del proceso de diseño. Otra fuente de información es el conocimiento de los posibles defectos y su distribución. Las pruebas de caja negra y caja blanca pueden ser combinadas con técnicas basadas en la experiencia para aprovechar la experiencia de los desarrolladores, probadores y usuarios para determinar qué debería ser probado (Jorgensen, 2013).

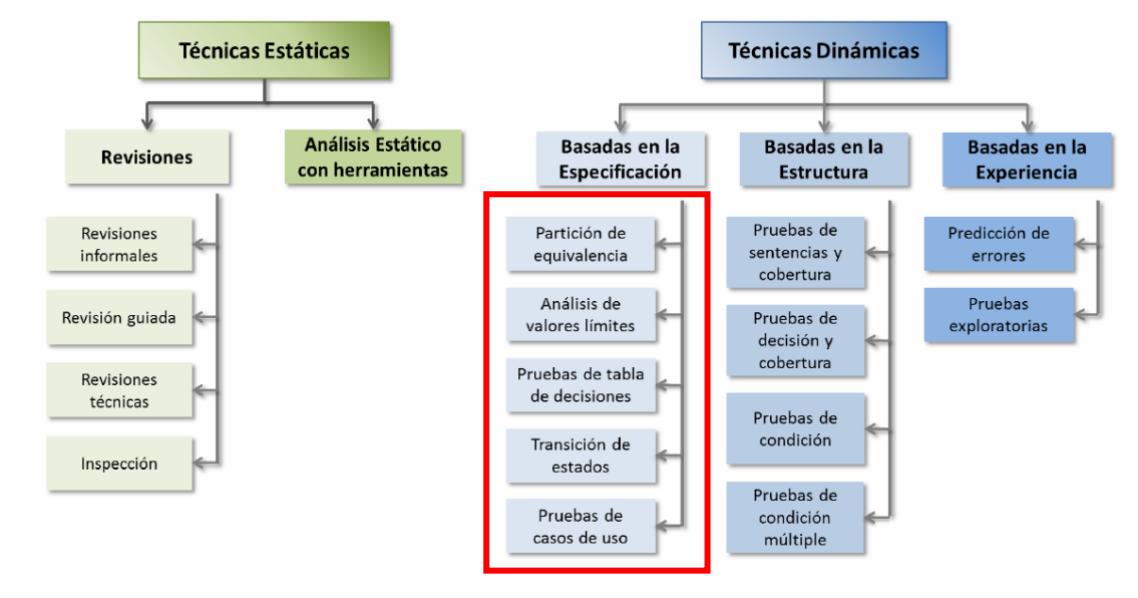


Figura 13: Cuadro resumen de las diferentes técnicas de diseño de pruebas, tanto estáticas, como dinámicas.

Partición de equivalencia

En la técnica de partición de equivalencia, el dominio de un posible valor de entrada es diferente para cada elemento de los datos de entrada y se dividen en clases o particiones de equivalencia. Una clase de equivalencia es un grupo de valores de entrada o salida del software que se espera que tengan un comportamiento similar, que sean procesados de la misma manera. La prueba de un representante de la clase de equivalencia se considera suficiente porque se asume que para cualquier otro valor de entrada para la misma clase de equivalencia, el objeto de la prueba no mostrará una reacción o comportamiento diferente;

- Si la prueba por un representante de la clase de equivalencia no funciona, suponemos que ninguno de los valores de esa clase de equivalencia funcionará.
- Si la prueba por un representante de la clase de equivalencia funciona, suponemos que todos los valores de esa clase de equivalencia funcionarán.

Las clases de equivalencia para un valor de entrada correcto son llamadas clases de equivalencia válidas. Además de las clases de equivalencia para los valores de entrada correctos, también han de ser probados para valores de entrada incorrectos (denominados clases de equivalencia inválidas) y deben ser ejecutados los casos de pruebas con representantes de estas clases.

¿Cómo pueden obtenerse sistemáticamente los casos de prueba? Por cada elemento de dato de entrada que debería ser probado, el dominio de todos los posibles datos de entrada debe ser determinado. Este dominio es segmentado en clases de equivalencia (tanto válidas como inválidas). Cada clase tiene que ser probada. Si la especificación de los objetos de prueba indica que algunas de las clases de equivalencia son procesadas de forma diferente, estos deben de ser asignados a una nueva subclase de equivalencia. Para completar los casos de prueba, el probador debe definir las condiciones previas y los resultados esperados para cada caso de prueba. Particionar en clases de equivalencia y seleccionar los valores representativos debe de realizarse cuidadosamente, ya que la probabilidad de generar fallos es más alta dependiendo de la calidad de la partición, como también es importante qué caso de prueba es ejecutado (Whittaker, 2002).

Poniendo un ejercicio a modo de ejemplo, el siguiente párrafo es una simulación de las especificaciones de los requisitos de un programa que calcula el bonus de navidad de los empleados dependiendo de su vinculación a la empresa:

"*Los empleados reciben un bonus igual al 50 por ciento de su ingreso mensual si han estado trabajando para la compañía más de tres años, un 75 por ciento si lo han hecho más de cinco años, y un 100 por ciento si es más de 8 años"* Los resultados de la partición de equivalencia pueden ser los siguientes:

Partición de equivalencia (años)	Valor representativo
0≤x≤3	2
3<x≤5	4
5<x≤8	6
x>8	10

Tabla 1: Partición de equivalencia válido y un valor representativo

Partición de equivalencia (años)	Valor representativo
x<0	2
3<x≤5	4

Tabla 2: Partición de equivalencia inválida y un valor representativo

Por lo general, el objeto de la prueba tiene más de un parámetro de entrada. La clase de equivalencia técnica da como resultado al menos dos clases de equivalencia (una válida y otra invalida) para cada uno de estos parámetros del objeto de prueba. Por lo tanto, hay por lo menos dos valores representativos que se deben de utilizar como entrada de prueba para cada parámetro. Para especificar un caso de prueba, a cada parámetro se le debe asignar un valor de entrada. Los valores representativos de todas las clases de equivalencia válidos deben ser combinados en los casos de prueba. Significa que todas las posibles combinaciones válidas de clases de equivalencia serán cubiertas. Cualquiera de estas combinaciones construye lo que se llama un "caso de prueba válido" o un "caso de prueba positivo" El valor representativo de una clase de equivalencia inválida puede ser combinado con representantes de otras clases de equivalencia válidas. Por lo tanto, cada clase de equivalencia inválida conduce a una suma adicional de un "caso de prueba inválido" o "caso de prueba negativo".

El número de casos de prueba válidos es el producto de clases de equivalencia válidos por parámetro. Debido a esta combinación multiplicativa, incluso unos pocos parámetros pueden generar cientos de casos de prueba "válidos" (reglas para reducir los casos de prueba). Las clases de equivalencia inválidas no son combinadas de forma multiplicativa. Un valor incorrecto debería de ser solo combinado con los correctos, porque un valor de parámetro incorrecto normalmente desencadena una gestión de excepciones de otra forma habría un riesgo de enmascarar un defecto. Una partición de equivalencia para un dato de salida tiene el mismo principio de división en clases de equivalencia, pero es usado para los datos de salida (lo que incrementa su complejidad).

Un criterio de completitud de unas pruebas para la partición de clase de equivalencia puede ser definido como el porcentaje de las clases de equivalencia ejecutadas en comparación con el número total de las clases de equivalencia especificadas (Jorgensen, 2013):

$$PE - cobertura = \left(\frac{nº\ PE\ ejecutadas}{nº\ total\ de\ PE}\right) * 100\%$$

Ecuación 1: Cobertura de las particiones de clases de equivalencia

Análisis de valores límite

Un análisis de valores límite (AVL) aporta valor a los casos de prueba que han sido identificados en las clases de equivalencia. El análisis de valores límite debería realizarse junto con la partición de clases de equivalencia porque los errores son hallados más a menudo en los límites de las clases de equivalencia que dentro de ellas. Unas pruebas que utilizan análisis de valores límite suelen detectar a menudo defectos. La técnica puede ser solo aplicada si el conjunto de los datos de una clase de equivalencia ha identificado los límites. Se comprueba la frontera de la clase de equivalencia seleccionada. En cada frontera, se prueba el valor del límite exacto, un valor por debajo del límite y un valor por encima del límite. De esta manera, el mínimo incremento posible en ambas direcciones es probado. Por lo tanto, se realizan tres casos de prueba para cada límite. Si el límite superior de una clase de equivalencia coincide con el límite inferior de la clase de equivalencia adyacente, entonces las respectivas pruebas coinciden también. En muchos casos no existe un valor de límite real porque éste pertenece a otra clase de equivalencia. En tales casos, puede ser suficiente con probar el límite con los valores que están justo dentro de esa clase de equivalencia, y los que están justo fuera.

Análogamente a la determinación de casos de prueba en la partición de clases de equivalencia, los límites válidos dentro de una clase de equivalencia pueden combinarse como casos de prueba. Los límites inválidos deben de ser verificados por separado y no pueden ser combinados con otros límites inválidos. Los valores intermedios de la clase de equivalencia no son necesarios si los dos valores del límite se prueban. Los análisis de valores límite pueden ser también aplicados como resultado de una clase de equivalencia. Para clases de equivalencia inválidas, el análisis de los valores de los límites es solamente útil para el tratamiento de excepciones para el objeto de prueba esperado dependiendo de un límite de clase de equivalencia. A modo de ejemplo, contamos con este ejercicio muy sencillo:

La capacidad de contenedores es de 5000 kilos en paquetes exactos de 1 kg

Primero identificamos las clases válidas:

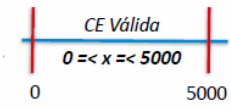


Figura 14: Clases de equivalencia válidas

A continuación identificamos el límite inferior y superior, y sus valores menores y mayores:

Figura 15: Valores límite inferior (izquierda) y superior (derecha) y sus valores menores y mayores

Después solo nos quedaría crear un caso de prueba para cada valor e identificar el representante:

CdP	Clase equivalencia	Tipo	Representante	CdP_01	CdP_02	CdP_03	CdP_04	CdP_05	CdP_06
Capacidad (CE1)	iCE11; **x < 0**	inválido	-1	*					
	vCE12; **x = 0**	válido	0		*				
	vCE13; **x > 0**	válido	1			*			
	vCE14; **x < 5000**	válido	4999				*		
	vCE15; **x = 5000**	válido	5000					*	
	iCE16; **x > 5000**	inválido	5001						*

Figura 16: Caso de prueba para cada valor

Si optimizamos cada uno de los casos de prueba identificados, podemos darnos cuenta de que los casos “CdP_03” y “CdP_04” se encuentran dentro de la frontera, sin ser límites, con lo cual no serían necesarios, quedando la tabla final de la siguiente manera:

CdP	Clase equivalencia	Tipo	Representante	CdP_01	CdP_02	CdP_05	CdP_06
Capacidad (CE1)	iCE11; **x < 0**	inválido	-1	*			
	vCE12; **x = 0**	válido	0		*		
	vCE15; **x = 5000**	válido	5000			*	
	iCE16; **x > 5000**	inválido	5001				*

Figura 17: Caso de prueba para cada valor, en su forma optimizada.

Análogamente al criterio de completitud para una partición de clase de equivalencia, la cobertura de los valores de los límites (*boundary values*) puede ser también predefinida y calculada después de la ejecución de las pruebas:

$$BV - cobertura = \left(\frac{nº\ de\ BV\ probados}{nº\ total\ de\ BV}\right) \cdot 100\%$$

Ecuación 2: Cobertura de los valores de los límites.

Hay que poner atención en que los valores del límite, como los valores adyacentes superior e inferior del límite, deben tenerse en cuenta. Sin embargo, solo los valores desiguales son usados para el cálculo. Los valores superpuestos de las clases de equivalencia adyacentes son contados como un valor del límite, porque únicamente un caso de prueba con ese valor es posible (Whittaker, 2002).

Pruebas de tabla de decisión

La partición de equivalencia y el análisis de valores límite consideran los diferentes valores de entrada como independientes, y cada uno de estos valores se consideran por separado para generar los casos de prueba. Las tablas de decisión constituyen una buena manera de reflejar los requisitos del sistema que contienen condiciones lógicas y de documentar el diseño del sistema interno.

Se basan en la gráfica causa-efecto que muestra las relaciones lógicas entre las distintas causas y sus efectos en los resultados para cada diseño de casos de prueba. Debe ser posible para encontrar las causas y los efectos a partir de la especificación. Toda causa es descrita como una decisión, compuesta de condiciones de entrada (o combinaciones de ellas). Las condiciones están conectadas con operadores lógicos (Patton, 2001).

Una tabla de decisión tiene dos partes: en la parte superior se listan las causas (condiciones). En la parte inferior se listan los efectos (acciones). Cada columna es un caso de prueba. Es decir, la combinación de condiciones y de efectos esperados para esta combinación. En el caso menos optimizado, cada causa de combinación es considerada un caso de prueba: la tabla de decisión asociada es denominada tabla de decisión completa. Para n condiciones, hay 2^n posibles combinaciones. Por lo tanto la tabla de decisión completa tiene 2^n columnas. Las tablas de decisiones completas se pueden simplificar. Puede ocurrir que no todas las combinaciones tengan sentido y se puedan eliminar algunas. También si el valor de una o mas condiciones específicas no afectan a las acciones de dos o más combinaciones, estas se podrían combinar.

El gráfico debe de transformarse en una tabla de decisión desde la cual se pueden tomar casos de prueba. Pasos para transformar un gráfico en una tabla de decisión:

1. Elegir un efecto.
2. Mirando el gráfico, encontrar combinaciones de causas que tienen este efecto y combinaciones que no lo tengan.
3. Añadir una columna en la tabla para cada una de estas combinaciones de causa y los estados de los efectos causados restantes.
4. Comprobar si las entradas de la tabla suceden varias veces y, en caso afirmativo, borrarlas.

A veces la tabla de decisión se deriva de la especificación sin usar los gráficos de la causa-efecto. Las pruebas de tablas de decisión tienen el objetivo de diseñar las combinaciones de las condiciones capaces de conseguir fallo (activar acciones inválidas). Vamos a poner un ejercicio a modo de ejemplo: *Para sacar dinero de un cajero, las siguientes condiciones deben ser satisfechas:*

- *La tarjeta bancaria es válida (E1).*
- *Introducir el pin correctamente (E2).*
- *El número máximo de intentos para introducir el PIN correctamente es 3 (E3).*
- *Hay dinero en el cajero, y en la cuenta (E4).*

Las siguientes acciones son posibles:

- *Rechazar la tarjeta (A1).*
- *Preguntar por otro pin (A2).*
- *"Tragarse" la tarjeta (A3).*
- *Pedir una cantidad de dinero alternativa (A4).*
- *Pagar la cantidad requerida (A5).*

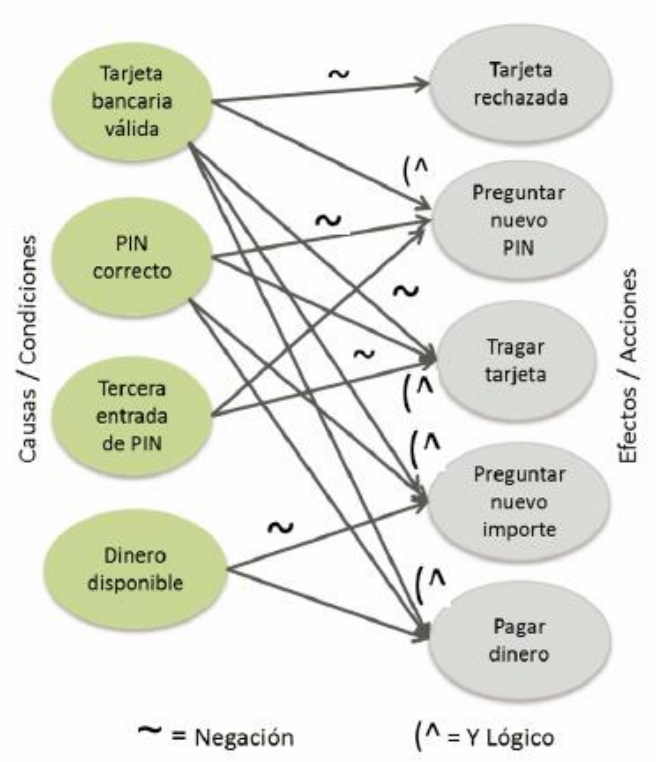


Figura 18: Gráfica representativa de las condiciones y acciones.

Ejemplo: Cajero

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	Condición / Causa																
E1	Tarjeta válida	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T
E2	PIN correcto	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T
E3	3 PINs incorrectos	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T
E4	Dinero disponible	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
	Efecto / Acción																
A1	Rechazar tarjeta	Y	Y	Y	Y	Y	Y	Y	Y								
A2	Pedir nuevo PIN									Y	Y						
A3	Tragar tarjeta											Y	Y				
A4	Pedir nuevo importe													Y		Y	
A5	Pagar dinero														Y		Y

Tabla de decisión: **Completa**

Figura 19: Tabla de decisión completa

Cada columna (también llamada “regla”) de la decisión es un caso de prueba. Las condiciones y dependencias para las entradas, los productos y sus resultados correspondientes a esta combinación de entradas se puede leer directamente en cada columna. Un requisito mínimo es

la ejecución de todas las columnas de la tabla de decisiones por al menos un caso de prueba. Esto verifica todas las combinaciones sensibles de las condiciones y efectos de éstas. Las gráficas de causa-efecto y las tablas de decisiones pueden crecer muy rápidamente y perder legibilidad cuando el número de condiciones y acciones dependientes aumenta. Sin el apoyo adecuado de las herramientas, la técnica no es fácilmente aplicable. El enfoque sistemático para definir una tabla de decisión con todas las posibles combinaciones puede revelar combinaciones que no están incluidas cuando son usadas otras técnicas de diseño de casos de prueba (Jorgensen, 2013).

Pruebas de transición de estado

En muchos casos, no solo el valor de entrada actual, sino también el histórico de ejecución o eventos o valores de entrada, influyen en los valores de salida y en cómo se comportan los objetos de prueba. Se utilizan diagramas para ilustrar las dependencias del histórico de estados. Y éstos, son la base del diseño de las pruebas de transición de estado. El objeto del sistema o de prueba se inicia a partir de un estado inicial, y a partir de éste se generan diferentes estados. Las transiciones de estado se activan por un evento que normalmente es una invocación de la función. Las transiciones de estado pueden involucrar acciones. Además del estado inicial, el otro estado especial es el estado final. Las máquinas de estados finitos, los diagramas de estado, o las tablas de transición de estado modelan este comportamiento.

La definición de una máquina de estado finito, según Bezier, es:

Una maquina abstracta (por ejemplo, un programa, un circuito lógico, la transmisión de un coche) para la cual el número de estados y símbolos de entrada son ambos finitos y fijos. Una máquina de estados finitos consiste en estados, transiciones, entradas y salidas.

Un diagrama de estado es un diagrama que representa los estados que un sistema o componente puede asumir, y muestra los eventos o circunstancias que causan o resultan de un cambio de estado a otro. Cualquier sistema en el que se obtiene un resultado diferente del mismo valor, dependiendo de qué ha pasado antes, es un sistema de estado finito. Por ejemplo; una pila o un procesador de textos. A modo de ejemplo, haremos el siguiente ejercicio:

Considera una pila que acepta cadenas (tipo "string"). Un posible caso de prueba con condiciones previas y posteriores es el siguiente:

- *Condición previa: la pila se inicializa "init"; el estado es vacío "empty".*
- *Valor de entrada: "push" de "hello".*
- *Resultado esperado: la pila contiene "hello".*
- *Condición posterior: el estado de la pila es relleno "filled".*

Para las pruebas de transición de estado, se pueden definir diferentes niveles de intensidad de la prueba (Whittaker, 2002). En el ejemplo de la pila, los estados son: *empty, filled y full.* Si se asume una altura máxima de 4, los tres estados serán alcanzados después de llamar a las siguientes funciones: "init" [*empty*], "push" [*filled*], "push" [*filled*], "push" [*filled*], "push" [*full*].

En la prueba anterior no se han utilizado todas las funciones de la pila. Otro requisito para la prueba sería invocar todas las funciones. Con la misma pila de antes, la siguiente secuencia de llamadas de funciones es suficiente para cumplir con este requisito: "init" [*empty*], "push" [*filled*], "top", "pop" [*empty*], "delete". Sin embargo, en esta secuencia tampoco se han alcanzado todos los estados. Una prueba de transición de estado debería ejecutar todas las funciones específicas de un estado concreto al menos una vez, comprobando así el cumplimiento entre

comportamiento especificado y el real del objeto de pruebas. Para identificar los casos de pruebas necesarios, la máquina de estados finitos es transformada en un árbol de transición:

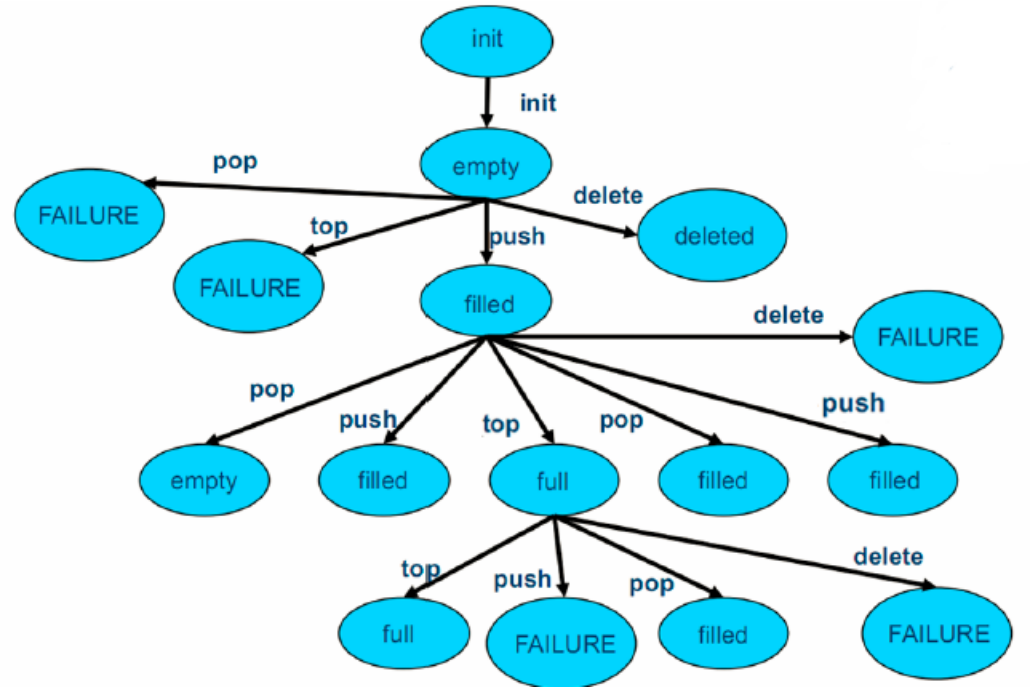


Figura 20: Árbol de transición de estados de una pila.

Otra posible representación del diagrama de transición de estados es la tabla de estado. Consideremos el siguiente diagrama de transición de estado:

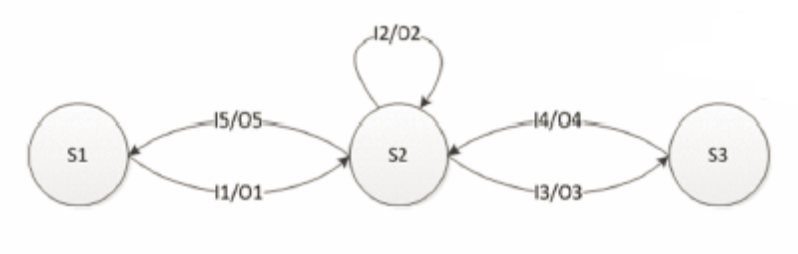


Figura 21: Diagrama de transición de estado a modo de ejemplo.

Con 3 estados (S1, S2, S3) y 5 transiciones, I_X/O_X (x =1, 2, 3, 4 y 5), representan las entradas y salidas para cada transición. La tabla de estado registra todos los estados de un lado de la tabla y todos los eventos que causan transiciones a lo largo de la parte superior (o viceversa). Entonces cada celda representa una pareja de estado-evento. El contenido de cada celda indica el estado en el que se moverá el sistema, cuando el evento correspondiente se produzca durante el estado asociado. Esto incluye los eventos que no se espera que sucedan en algunos estados (pruebas negativas).

		Transiciones				
		I1	I2	I3	I4	I5
Estados	S1	S2	-	-	-	-
	S2	-	S2	S3	-	S1
	S3	-	-	-	S2	-

Figura 22: Tabla de estado que representa al diagrama de transición de estado de la figura 21.

Una tabla de estado permite ver los números totales de combinaciones de estados y transiciones tanto válidos como inválidos. A través de una tabla de estado se pueden ver fácilmente las pruebas negativas. El objeto en las pruebas de transición de estado puede ser un sistema completo con diferentes estados del sistema, así como un sistema orientado a objetos con diferentes estados. Siempre que el histórico lleve a un comportamiento distinto, se debe aplicar una prueba de transición de estados. Estas pruebas son también una buena técnica para una prueba de sistema. Por ejemplo, la prueba de una interfaz gráfica de usuario puede ser diseñada de la siguiente forma: si los controles de pantalla y usuario son vistos como estados y las reacciones de entrada como transiciones de estado, entonces la prueba de una interfaz gráfica puede ser modelada como una maquina finita de estado.

No siempre es fácil de identificar los estados de un objeto de prueba. A menudo, el estado no está definido por una sola variable, sino que es el resultado de una constelación de valores de varias variables. Los criterios para la intensidad de la prueba y el grado de consecución también pueden ser definidos para las pruebas de transición de estado. Para aplicaciones muy críticas, pueden realizarse pruebas de transición de estado todavía más intensificadas para mejorar el grado de consecución. Conseguir cobertura suficiente a menudo no es posible, debido al gran número de casos de prueba necesarios. Por lo tanto se debe verificar un número limitado de combinaciones o secuencias (Jorgensen, 2013).

Pruebas de casos de uso

UML (*Unified Modeling Language* – Lenguaje unificado de modelado) define más de 10 notaciones gráficas las cuales pueden ser usadas en el desarrollo del software. Una de esas notaciones son los diagramas de casos de uso. Utilizar este tipo de pruebas nos ayuda a identificar los casos de prueba que ejecutan todo el sistema sobre una base de transacción por transacción de principio a fin. Un caso de uso es una descripción de un uso particular del sistema por un agente o usuario del sistema. Cada caso de uso describe las interacciones que el actor tiene con el sistema para conseguir una tarea específica (o, al menos, producir algo de valor al usuario). Los actores son, por lo general, las personas, pero también pueden ser otros sistemas. Los casos de uso son una secuencia de pasos que describen las interacciones entre el actor y el sistema. Los casos de uso se definen en términos del actor, no del sistema. Describen lo que el actor hace y lo que el actor ve en lugar de lo que espera el sistema (entradas/salidas).

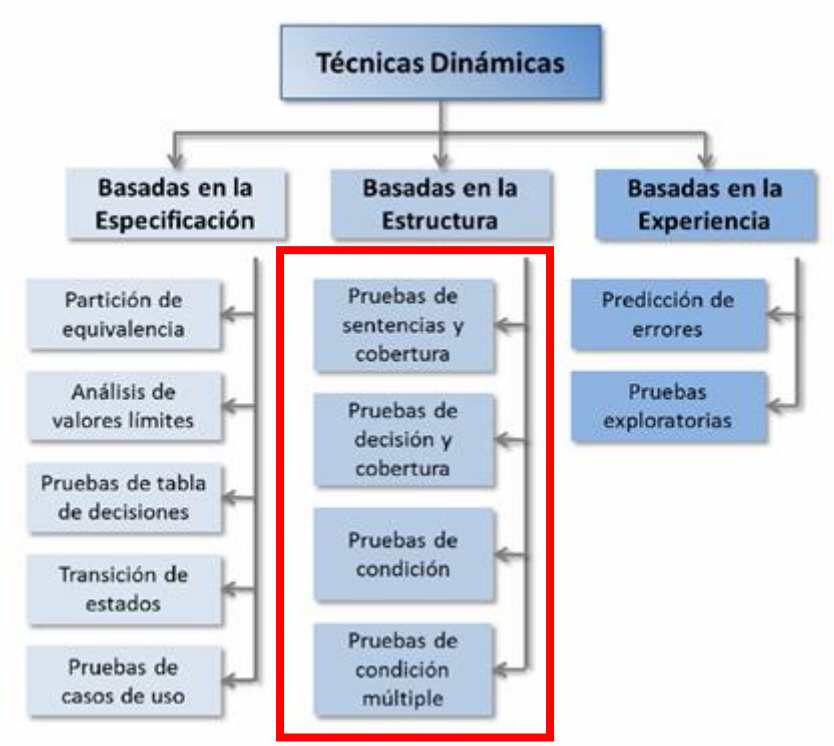
Un diagrama de casos de uso en su forma más simple es una representación de la interacción de un usuario con el sistema y representa las especificaciones de un caso de uso. Este tipo de diagrama que se utiliza típicamente en conjunción con el caso de uso textual y a menudo se acompaña de otros tipos de diagramas. A menudo utilizan el lenguaje y los términos de la empresa en lugar de términos técnicos, sobre todo cuando el actor es un usuario de negocios. Sirven como base para el desarrollo de casos de prueba, sobre todo a nivel de pruebas de sistemas y pruebas de aceptación. Los casos de uso también pueden descubrir defectos de integración. Cuando éstos son utilizados de manera que el actor puede ver las interfaces del

sistema como un enlace de comunicación o subsistema. Los casos de uso describen cómo el proceso fluye a través de un sistema basado en el uso más probable. Esto hace que los casos de prueba derivados de los casos de uso sean particularmente buenos para encontrar defectos en el uso real del sistema (los defectos que los usuarios tienen más probabilidad de encontrarse la primera vez que usan el sistema) (Whittaker, 2002).

Cada caso de uso, normalmente, tiene un escenario dominante (de mayor probabilidad) y a veces escenarios alternativos adicionales (cobertura, por ejemplo, casos especiales o condiciones excepcionales). Cada caso de uso debe especificar:

- Condición previa que es necesaria de cumplir para que funcione el caso de uso.
- Condiciones posteriores que son los resultados observables.
- Descripción del estado final del sistema después de que el caso de uso haya sido ejecutado satisfactoriamente.

Un criterio posible es que cada caso de uso y cada posible secuencia de casos de uso en el diagrama sea probado al menos una vez por cada caso de prueba. Puesto que las alternativas y extensiones son también casos de uso, este criterio requiere su ejecución.



Las técnicas de diseño basadas en la estructura son un buen camino para generar casos de prueba adicionales que difieren de las pruebas existentes. Para ayudar a asegurar una mayor amplitud de las pruebas, en el sentido de que los casos de prueba pueden conseguir el cien por cien de la cobertura. En cualquier medida se ejecutarán todas las partes del software desde el punto de vista de los elementos cubiertos. La cobertura de las pruebas mide de alguna forma específica la cantidad de pruebas realizadas por un grupo de pruebas:

$$Cobertura = \frac{Nº\ de\ elementos\ de\ cobertura\ ejecutados}{Nº\ total\ de\ elementos\ de\ cobertura} \cdot 100\%$$

Ecuación 3: Cobertura de las pruebas

La cobertura puede ser medida a diferentes niveles:

- A nivel de sistema o de aceptación. Los elementos de cobertura pueden ser requisitos, opciones del menú, pantallas o transacciones de negocio típicas.
- A nivel de integración. Se puede medir la cobertura de interfaz o interacciones específicas que han sido probadas. La cobertura de llamadas de módulo, objeto o llamadas a procedimientos pueden ser también medidas.
- A nivel de componente. Interesa la cobertura del código donde los elementos son códigos de elementos como sentencias y decisiones.

Las sentencias y decisiones son estructuras que pueden medir el código y son unas buenas herramientas de apoyo para estas medidas de cobertura. La cobertura de código normalmente se realiza a nivel de componente y a nivel de integración (Jorgensen, 2013).

Pruebas de sentencia y cobertura

Esta técnica se enfoca hacia cada sentencia del objeto de las pruebas. Los casos de prueba ejecutarán una cuota mínima predefinida o incluso todas las sentencias del objeto a prueba (cobertura se la sentencia completa). El primer paso consiste en traducir el código fuente en un gráfico de flujo de control. El gráfico hace que sea más fácil especificar en detalle los elementos de control que deben ser cubiertos. Las sentencias son representadas como nodos y el flujo de control entre las sentencias son representadas como aristas o conectores. Si las secuencias de los estados incondicionales aparecen en el fragmento del programa, entonces serán ilustradas como un solo nado, porque la ejecución de la primera sentencia garantiza que todas las siguientes sentencias serán ejecutadas. Las sentencias condicionales (*if, case*) y los bucles (*while, for*) tienen más de un conector para salir de ellos. Después de la ejecución de los casos de prueba se debe verificar cuáles de las sentencias se han ejecutado y cuándo se ha alcanzado el nivel de cobertura previamente establecido. Por lo general, todas las instrucciones deben llevarse a cabo, ya que es imposible determinar si las instrucciones que no se han realizado, son correctas o no. Los estudios y la experiencia en la industria indican que lo que no se considera razonable, a través de la prueba de la caja negra, en realidad puede lograr una cobertura de sentencia de un 60% a un 75%. Las pruebas típicas no sistemáticas es probable que tengan una cobertura de un 30% aproximadamente.

A modo de ejemplo, contamos con el siguiente ejercicio:

Considera el diagrama de flujo para un fragmento de un código que consiste en dos decisiones y un bucle. Cada sentencia es representada por un nodo en el diagrama de flujo. Cuál es la cobertura de sentencia.

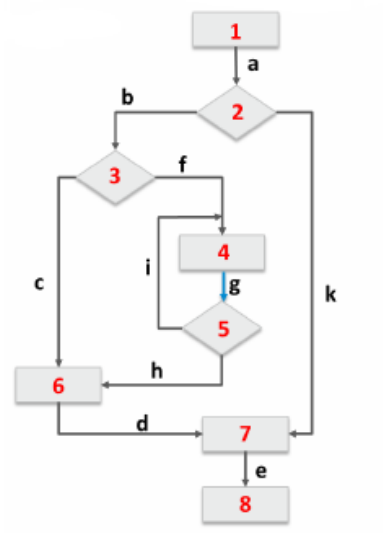


Figura 23: Diagrama de flujo del que hay que averiguar su cobertura de sentencia

La cobertura de sentencia es la cobertura de los nodos en el diagrama de flujo. Todos las sentencias (nodos) pueden ser alcanzados por un solo caso de prueba. En este caso, las aristas del gráfico deben ser recorridas en el siguiente orden: a, b, f, g, h, d, e. Con lo cual, en este ejemplo, un caso de prueba es el mínimo necesario para llegar a la cobertura de sentencia completa. Contamos con una cobertura del 100%.

Sin embargo, a veces el 100% de la cobertura de sentencia es difícil de alcanzar, por ejemplo, si aparecen las condiciones de excepción en el programa, las cuales pueden ser desencadenadas solo por un gran problema o no durante la ejecución de la prueba. Si la cobertura completa de todas las sentencias es requerida, y algunas de estas no se pueden ejecutar con ningún caso de prueba, entonces esto puede ser una indicación de código inalcanzable (código “muerto”). Es posible que falten sentencias después de un *ELSE* que no son detectadas por una prueba con la cobertura de sentencias (Whittaker, 2002).

Pruebas de decisión y cobertura

Un criterio más avanzado para las pruebas de caja blanca es la cobertura de rama (conectores) del diagrama de flujo. El objetivo es la ejecución de las decisiones. La ejecución de la sentencia está implícita en las decisiones y el resultado de la decisión determina qué sentencia se ejecuta posteriormente. Las pruebas deben asegurar que todas las decisiones son ejecutadas con los dos resultados posibles (verdadero y falso). A ésta técnica se le conoce como cobertura de decisión. Por ejemplo, si una sentencia *IF* no tiene la parte *ELSE* ésta debe ser considerada de todos modos.

La cobertura de decisión requiere la prueba de cada resultado de la decisión:

- Tanto *THEN* como *ELSE* en las sentencias *IF.*
- Todas las posibilidades de la sentencia *CASE* y los casos de error.
- Para los bucles, tanto en la no ejecución del cuerpo del bucle como en el flujo del cuerpo del bucle.

En el ejemplo de la figura 23, los siguientes tres casos de prueba adicionales son necesarios para que todas las ramas del grafo de flujo de control sean alcanzadas (cobertura de decisión)

- CP1: a b f g h d e

- CP2: a b c d e
- CP3: a b f g i g h d e
- CP4: a k e

Pero, ¿es cuatro el número mínimo de casos de prueba para lograr una cobertura complete de decisiones? El cálculo cuenta solo si una ejecución ha sido ejecutada. La frecuencia de ejecución no es relevante (en el ejemplo anterior, los extremos A y E son recorridos una vez para cada caso de prueba). Si ejecutamos solo los tres primeros casos de prueba, en el ejemplo anterior CP1, CP2 y CP3, la arista *k* no se ejecutaría. Esto nos da una cobertura de 9 decisiones ejecutadas de un total de diez. Con lo cual contaríamos con una cobertura del 90%.

La cobertura de decisión por lo general requiere la ejecución de un mayor número de casos de prueba que la cobertura de sentencias. En contraste con la cobertura de sentencias, la cobertura de decisión (rama) hace posible detectar los estados que faltan en las ramas vacías. El 100% de la cobertura de decisión garantiza el 100% de la cobertura de sentencias, pero no al revés. Por lo tanto, la cobertura de decisión es un criterio con más fuerza. Cada una de las ramas se considera por separado, sin requerir ninguna combinación particular de las ramas individuales. Para los sistemas orientados a objetos, ambas coberturas son inadecuadas ya que el flujo de control de las funciones en las clases suele ser corto y no muy complejo (Jorgensen, 2013).

Pruebas de condición y condición múltiple

Hay niveles más fuertes de cobertura estructural que la cobertura de decisión. La cobertura de condición y condición múltiple puede ser de varios tipos (Jorgensen, 2013):

- Simple: el objetivo de la prueba de condición es hacer que cada condición unitaria en la prueba adopte un valor *true* y uno *false*.
- Múltiple: el objetivo de la prueba es que se prueben todas las combinaciones posibles de los resultados de la condición en cada decisión. Cada parte atómica de condiciones es evaluada una vez para cada uno de los valores lógicos.
- Mínima múltiple: el objetivo es que la decisión se cumpla al menos una vez con las condiciones que contenga.

A modo de ejemplo, *consideremos la siguiente decisión: IF(x<5)OR(y<8):*

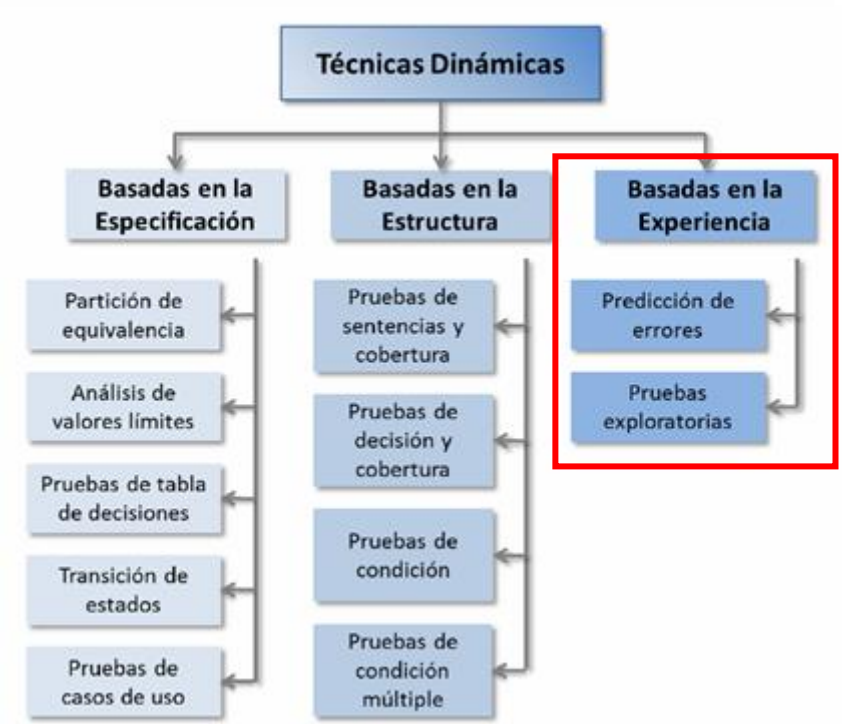
Condición 1 (x < 5)	Condición 2 (y < 8)	Resultado de la decisión	Caso de prueba
True	**False**	**True**	CP1 (x=4, y=9)
False	**True**	**True**	CP2 (x=6, y=7)
True	*True*	*True*	*CP3 (x=4, y=7)*
False	False	False	CP4 (x=6, y=9)

Figura 24: Tabla de decisión

Los casos de prueba necesarios para que se cumpla la cobertura de condiciones en todos sus tipos es:

- Simple (CP1, CP2).
- Múltiple (CP1, CP2, CP3, CP4).
- Mínimo múltiple (CP3).

En las pruebas de condición múltiples requieren que todas las combinaciones *true-false* de las condiciones atómicas se realicen al menos una vez. En el ejemplo anterior con 4 casos de prueba, la decisión da ambos valores lógicos como resultado. Por lo tanto, las pruebas de condición múltiple cumplen el criterio de sentencia y de cobertura de la decisión. Se trata de un criterio más amplio que también tiene en cuenta la complejidad de las condiciones compuestas. Sin embargo, es una técnica muy costosa debido al creciente número de condiciones atómicas, lo cual hace que el número posible de combinaciones aumente exponencialmente (del orden de 2_n con n condiciones atómicas).



A menudo se ha observado que algunas personas parecen ser especialmente hábiles cuando prueban un programa sin utilizar ninguna metodología en particular. Algunos defectos son difíciles de encontrar utilizando métodos sistemáticos, por lo que un probador puede ser muy creativo en la búsqueda de defectos esquivos y por eso es recomendable realizar pruebas adicionales de forma intuitiva.

La base de las técnicas basadas en la experiencia es la habilidad, experiencia y conocimiento del probador. El probador selecciona casos de prueba para descubrir problemas esperados y sus síntomas. Los casos de prueba se basan en la experiencia de donde han ocurrido errores en el pasado o el supuesto del probador en donde se pueden producir errores en el futuro. Es requerido conocimiento en el desarrollo de aplicaciones similares y el uso de tecnologías similares para el diseño de casos de prueba, además de experiencia realizando pruebas. La predicción de errores y las pruebas exploratorias son dos técnicas para las pruebas basadas en la experiencia (Patton, 2001).

Predicción de errores

La predicción de errores es una técnica que siempre debe utilizarse como complemento de otras técnicas más formales. La efectividad de la predicción del error es muy dependiente de la habilidad del probador. Algunas personas parecen ser naturalmente más hábiles en pruebas y otros son probadores porque tienen mucha experiencia ya sea como un probador o como trabajador en un sistema en particular, y son capaces de sacar a relucir sus debilidades. En el

uso de técnicas más formales, el probador es capaz de lograr una mejor comprensión del sistema, lo que hace y cómo funciona.

No hay reglas para predecir el error. El probador debe pensar en situaciones en las que el software puede tener defectos. Un enfoque estructurado a la técnica de predicción de error se llama ataque de falta. Se construye una lista de defectos y fallos desde la base de la propia experiencia del probador u otras personas, desde los defectos disponibles y de los datos del fracaso, y desde el conocimiento común acerca de por qué el software falla. Las pruebas estarán diseñadas para atacar a estos defectos o fallos (Jorgensen, 2013).

Pruebas exploratorias

Si los documentos que constituyen la base para el diseño de pruebas son de muy baja calidad, obsoletos o no existen en absoluto (quizás solo existe el software), la técnica de pruebas exploratorias puede ayudar. Esta técnica también es aplicable cuando el tiempo es muy limitado, ya que utiliza mucho menos tiempo que otras técnicas. El enfoque se basa principalmente en la intuición y la experiencia del probador. La prueba exploratoria es el diseño y ejecución de pruebas al mismo tiempo.

Las tareas y funciones del objeto de prueba se “exploran”. Después de esta exploración previa se decide que partes del software se pondrán a prueba. Unos pocos casos de prueba son ejecutados y sus resultados son analizados. Después de ejecutarlos, el comportamiento “desconocido” del objeto de prueba será analizado más profundamente. Cualquier cosa considerada singular, así como también cualquier otra información, es utilizada para determinar los siguientes casos de prueba. De esta forma se mejora el conocimiento del objeto de las pruebas. Hace que cada vez sea más claro lo que hace el objeto de prueba, cómo funciona ese objeto de prueba, qué problemas de calidad podría haber y qué expectativas con respecto al programa deben tenerse.

A veces, tiene sentido restringir las pruebas exploratorias a elementos específicos del programa (ciertas tareas o funciones). Los elementos se desglosan en un contrato de prueba. En estos casos es conveniente utilizar el contrato de prueba como una declaración de pruebas y posibles ideas de cómo probar. La prueba de un contrato no debería llevar más de una o dos horas de tiempo ininterrumpido. Al realizar el contrato de prueba, las siguientes preguntas son de interés:

- ¿Por qué?, ¿Cuál es el objetivo de la realización de la prueba?
- ¿Qué se va a probar?
- ¿Cómo?, ¿Qué método de prueba debería ser utilizado?
- ¿Qué problemas deberían ser encontrados?

Contrariamente a las técnicas sistemáticas, un criterio para la terminación de la prueba intuitiva no puede ser determinado previamente. La predicción de errores y las pruebas exploratorias a menudo se pueden utilizar con buenos resultados. Sin embargo, solo se deben utilizar como complemento de las técnicas sistemáticas. La efectividad de estos métodos depende en gran medida de la habilidad, la intuición y la experiencia previa del probador con aplicaciones como la del objeto de prueba y las tecnologías utilizadas. Tales enfoques también pueden contribuir a encontrar huecos y errores en el análisis de riesgo. Si la prueba intuitiva se ejecuta, como complemento a las pruebas sistemáticas, se puede detectar inconsistencia en las especificaciones de la prueba (Patton, 2001).

3. Arquitectura de las pruebas.

Supongamos que nos encontramos en un proyecto de automatización, en el que tenemos que analizar, diseñar y ejecutar las pruebas de una aplicación aún en desarrollo. En el actual proyecto presentaré dos ejemplos simples. Uno para testear cualquier página web y otro para testear cualquier API REST. El método más frecuente para distribuir aplicaciones a los clientes es el llamado integración continua/distribución continua, o por sus siglas en inglés, CI/CD (*Continuous integration / Continuous delivery*). A medida que los desarrolladores aumentan las versiones de la aplicación, los probadores aumentarán el número de pruebas ejecutadas con el mismo incremento que los desarrolladores amplían su código. La CI/CD es una solución para los problemas que puede generar la integración del código nuevo a los equipos de desarrollo y operaciones. Para poder simular un entorno lo más real posible, se ha utilizado el servidor de automatización *Jenkins*. En él, podremos ejecutar todas las pruebas de forma remota y en el dispositivo que nosotros queramos.

Las pruebas se ejecutarán remotamente, con lo cual necesitaré unos agentes activos y siempre disponibles para ese servidor de automatización. Para ello, utilizo los servicios que me proporciona el gestor de contenedores *Docker*. Gracias a él, tendré siempre activo uno o cuantos agentes queramos para poder ejecutar nuestras pruebas. Para poder controlar nuestro código, he elegido el controlador de versiones *Git.* Es el más utilizado y, hoy en día, la mayoría de los programadores saben utilizarlo. El código de nuestra automatización lo alojo en *Bitbucket.* Es un servicio de alojamiento web para los proyectos que utilizan el sistema de control de versiones *Mercurial* y *Git*. Hablando en un nivel muy bajo, utilizaré siempre el *framework* de *Serenity* para los reportes de cada ejecución. La estrategia de desarrollo que seguiré es BDD (*Behavior Driven Development,* desarrollo dirigido por comportamiento). Lo que plantea es definir un lenguaje común para el negocio y para los programadores, y utilizar eso como parte inicial del desarrollo y las pruebas. BDD es una evolución del TDD (*Test Driven Development,* desarrollo dirigido por test). Mientras que TDD se enfoca en la prueba unitaria, BDD se centra en escribir pruebas que verifiquen que el comportamiento del código es correcto desde el punto de vista de negocio. Como lenguaje de negocio utilizo *Cucumber*, ya que tiene librerías propias en *Java*, que es el lenguaje elegido para desarrollar las pruebas de ejemplo. Para facilitarme las tareas de automatización, utilizo los siguientes frameworks para testear una API REST y una página web: *Rest-Assured* y *Selenium*, respectivamente. Pasaré a describir cada una de las herramientas y características a continuación.

Jenkins

Jenkins es un servidor de integración continua, gratuito, open-source y actualmente uno de los más empleados para esta función. Esta herramienta proviene de otra similar llamada *Hudson*, ideada por Kohsuke Kawaguchi mientras trabajaba en *Sun Microsystems.* Unos años después de que Oracle comprara esta empresa, la comunidad de *Hudson* decidió renombrar el proyecto a Jenkins, migrar el código a GitHub y continuar el trabajo desde ahí.

La base de Jenkins son las tareas, en las cuales indicamos qué es lo que hay que hacer en cada construcción (en inglés, *build*). Por ejemplo, podríamos programar una tarea en la que se compruebe el repositorio de control de versiones cada cierto tiempo, y cuando un desarrollador quiera subir su código al control de versiones, este se compile y se ejecuten las pruebas. Si el resultado no es el esperado o nos hemos encontrado con algún error, Jenkins notificará al desarrollador y al equipo de pruebas. Si el build es correcto, podremos indicar a Jenkins que intente integrar el código y subirlo al repositorio de control de versiones. Yo voy a obviar toda la parte de desarrollo, para centrarme en las tareas que me proporciona Jenkins para las pruebas.

Para este proyecto, he instalado una imagen Jenkins directamente en un servidor de Windows. La versión que he instalado es la "2.164.3". A parte de los plugins que se instalan por defecto, he añadido el plugin de serenity y de groovy ya que los necesitaré para poder visualizar los reportes de las pruebas y poder compilar y ejecutar el proyecto. También he instalado un plugin llamado "*HTML Publisher plugin*" que nos servirá para publicar los reportes en formato HTML en nuestro Jenkins, y el plugin "*Cucumber reports*" que nos servirá para publicar los reportes del lenguaje de negocio que hayamos escogido, en este caso cucumber. Podemos observar la interfaz de la herramienta en la figura 25.

En la imagen podemos apreciar cuatro partes bien diferenciadas. En la primera de ellas tenemos toda la configuración necesaria para poder realizar una nueva tarea, la gestión de usuarios de la herramienta, historial de trabajos, administración de la herramienta y el manejo de credenciales. Esta última de todas sirve para poder utilizar contraseñas o claves que no queremos compartir con nadie ni que aparezcan en ningún historial de pruebas ni reportes. En mi caso, en este apartado he introducido mis credenciales de bitbucket, ya que los proyectos de automatización son privados y se necesitan credenciales de acceso. En la segunda de las partes, podemos apreciar los trabajos que hay en cola actualmente. En la tercera parte, contamos con el estado del ejecutor de construcciones, contando ahora mismo con un agente principal (servidor que está ejecutando Jenkins) y un agente remoto llamado "Agente", que será quien ejecute finalmente nuestras pruebas. Este agente es un contenedor de *Docker*, y se describirá en el apartado correspondiente. La última de las partes es la vista que tenemos de nuestras tareas. Podemos observar dos tareas en la vista, creadas a modo de ejemplo (Jenkins Docs, 2018).

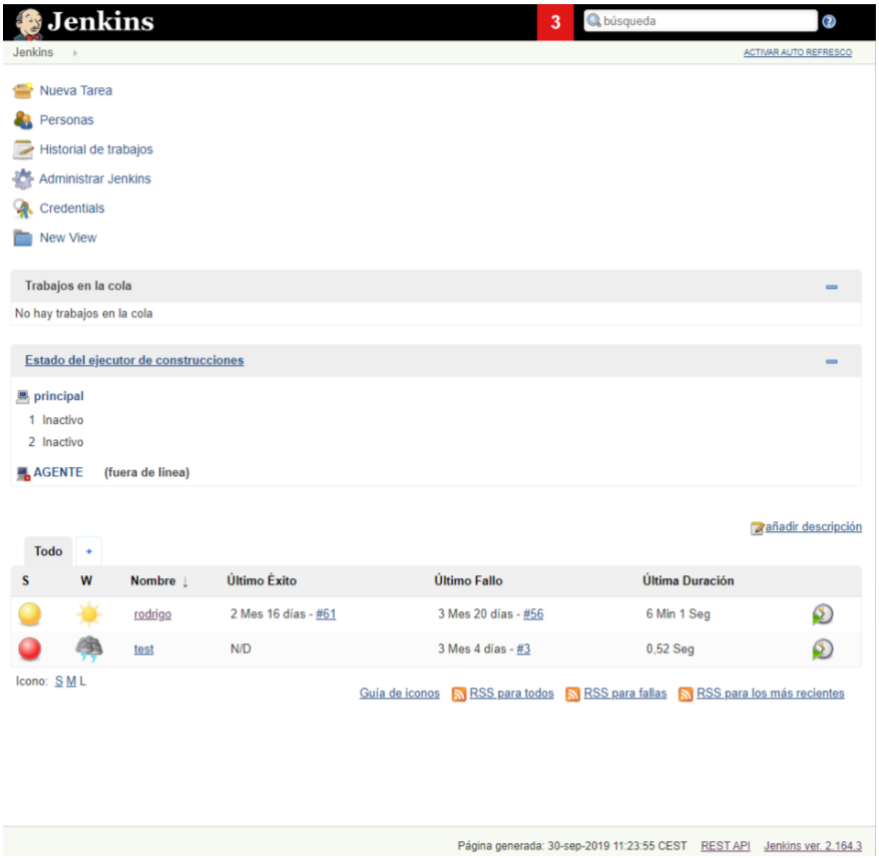


Figura 25: Interfaz de usuario del servidor de integración continua Jenkins instalado en el servidor.

Docker

Docker es una plataforma para desarrolladores y administradores de sistemas que se utiliza para desarrollar, desplegar y ejecutar aplicaciones con contenedores. Llamamos contenerizacion al uso de contenedores para desplegar aplicaciones. El uso de contenedores no es una práctica nueva, pero su uso facilita mucho la tarea de despliegue de cualquier tipo de servicio o aplicación. La contenerización es cada vez más popular ya que los contenedores tienen las siguientes características:

- Flexible: incluso las aplicaciones más complejas pueden ser contenerizadas.
- Ligero: Los contenedores aprovechan y comparten el núcleo de la máquina que los aloja.
- Intercambiable. Puedes desplegar mejoras y actualizaciones sobre la marcha.
- Portable: Puedes crear localmente, implementar en la nube y ejecutar en cualquier lugar.
- Escalable: Puedes incrementar y distribuir automáticamente las réplicas de los contenedores.
- Apilable: Puedes apilar servicios verticalmente y sobre la marcha.

Un contenedor se lanza ejecutando una imagen. Una imagen es un paquete ejecutable que incluye todo lo necesario para montar una aplicación (el código, librerías, variables de entorno y ficheros de configuración). Un contenedor es una instancia de ejecución de una imagen.

Los contenedores son nativos en Linux y comparten el núcleo de la máquina principal con otros contenedores. No necesita más memoria que cualquier otro ejecutable, lo que los hace ligeros.

Por el contrario, una máquina virtual ejecuta un sistema operativo completo con acceso virtual a los recursos de la máquina. En general, las máquinas virtuales proporcionan un entorno con más recursos de los que necesitan la mayoría de las aplicaciones.

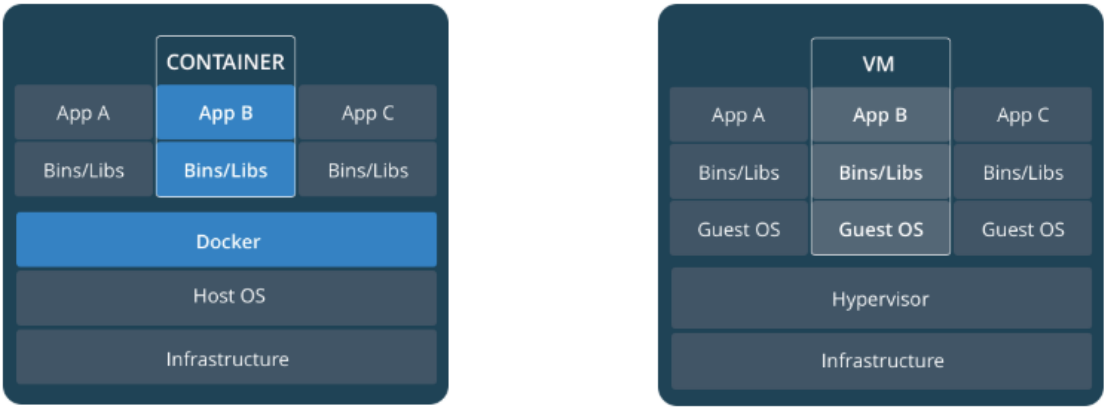


Figura 26: Comparativa de la infraestructura entre un sistema dockerizado y una máquina virtual.

Para el presente trabajo he instalado la versión de *docker desktop* en la raíz de un servidor con Windows 10. La versión que he instalado la podemos ver en la consola de comandos mediante el comando ***docker --version:***

```
PS C:\Users\rodri> docker --version
Docker version 18.09.2, build 6247962
```

Este comando nos indica la versión de docker que está en nuestra máquina actualmente. Como podemos ver en la salida de la consola de comandos, cuento con la versión estable 18.09.2. Para tener más información acerca de la instalación de docker, solo tenemos que ejecutar el comando ***docker info:***

```
PS C:\Windows\system32> docker info
Containers: 18
 Running: 18
 Paused: 0
 Stopped: 0
Images: 25
Server Version: 18.09.2
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
```

```
 Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
Swarm: inactive
Runtimes: runc
Default Runtime: runc
Init Binary: docker-init
containerd version: 9754871865f7fe2f4e74d43e2fc7ccd237edcbce
runc version: 09c8266bf2fcf9519a651b04ae54c967b9ab86ec
init version: fec3683
Security Options:
 seccomp
  Profile: default
Kernel Version: 4.9.125-linuxkit
Operating System: Docker for Windows
OSType: linux
Architecture: x86_64
CPUs: 2
Total Memory: 1.934GiB
Name: linuxkit-00155d544f09
ID: XOE2:C3DL:QE6N:73J4:DCG5:5C22:FBAN:4F6F:KYZO:727N:GVXU:BSE5
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): true
 File Descriptors: 119
 Goroutines: 123
 System Time: 2019-06-25T17:17:08.8282805Z
 EventsListeners: 1
Registry: https://index.docker.io/v1/
Labels:
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
Product License: Community Engine
```

Con esto podemos saber todas las características de nuestra herramienta. Todas ellas se fijan por defecto en nuestro servidor. *Docker* lo utilizamos para desplegar la imagen del nodo que ejecutará las pruebas en nuestro servidor de automatización Jenkins. La imagen del nodo se reúne en un fichero llamado *Dockerfile*, en el que incorporamos todas las características de nuestro contenedor, y una carpeta con los recursos necesarios para desplegar la imagen. En la figura 27 podemos observar la estructura del proyecto:

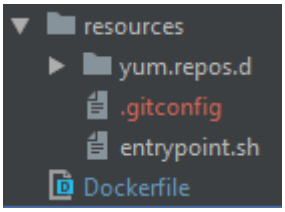


Figura 27: Estructura de ficheros del proyecto de infraestructura

El fichero *Dockerfile* tiene todas las especificaciones y características de la imagen que va a ser construida por el contenedor. El código que contiene es el siguiente:

```
#Sistema operativo de partida
FROM primetoninc/jdk:1.8

LABEL maintainer="Rodrigo Sánchez Alonso - UNIVERSIDAD DE VALLADOLDID - rodrigosanchezalonso@gmail.com"

# Esclavo de jenkins – Variables de entorno
ENV SLAVE_WORKDIR=/opt/jenkins \
    SLAVE_RUNNABLE=/usr/share/jenkins/slave.jar \
    JAVA_VM_MEM_MIN=512 \
    JAVA_VM_MEM_MAX=4096

# Opciones de las variables de entorno de Gradle
ENV GRADLE_HOME=/opt/gradle-4.10.2
ENV PATH="${PATH}:${GRADLE_HOME}/bin"

# Carpeta con los repositorios de todas las dependencias necesarias
ADD resources/yum.repos.d/epel.repo /etc/yum.repos.d/epel.repo

# SOFTWARE que vamos a incorporar a nuestra imagen
RUN curl --fail --location --retry 3 \
        http://opensource.wandisco.com/centos/7/git/x86_64/wandisco-git-release-7-2.noarch.rpm \
        -o /tmp/git-release-7-2.noarch.rpm \
    && rpm -ivh /tmp/git-release-7-2.noarch.rpm \
    && \rm -f /tmp/git-release-7-2.noarch.rpm \
    && yum clean all \
    && yum -y install unzip wget git bash curl zip py-pip openssh unix2dos libuuid which



# Descarga e instalación del software GRADLE
RUN curl --fail --location --retry 3 \
        http://services.gradle.org/distributions/gradle-4.10.2-bin.zip \
        -o /tmp/gradle.zip \
    && unzip /tmp/gradle.zip -d /opt/ \
```

```
    && \rm -f /tmp/gradle.zip


# Creación del esclavo de jenkins
RUN mkdir -p /usr/share/jenkins \
    && curl --create-dirs -sSLo ${SLAVE_RUNNABLE} \
    https://repo.jenkins-ci.org/public/org/jenkins-ci/main/remoting/3.26/remoting-
3.26.jar \
    && chmod 755 /usr/share/jenkins \
    && chmod 644 ${SLAVE_RUNNABLE}


# Configuración del entrypoint
ADD resources/entrypoint.sh /opt/
RUN chmod +x /opt/entrypoint.sh


# Creacion del volumen para los trabajos de jenkins
VOLUME ${SLAVE_WORKDIR}


# Configuración de GIT
ADD resources/.gitconfig /root/.gitconfig


ENTRYPOINT [ "/opt/entrypoint.sh" ]
```

La imagen de partida, llamada *primetoninc/jdk:1.8* no es más que una imagen basada en el sistema operativo centOS a la que se le ha incorporado el software *java development kit* de Oracle en su versión 1.8 (necesario para ejecutar los test). A continuación, nos encontramos con la etiqueta del mantenedor de la imagen actual y la incorporación de las variables de entorno necesarias para ejecutar el agente de Jenkins. Éstos son: el directorio de trabajo, el lugar donde se encuentra el ejecutable del agente y el rango de memoria de acceso aleatorio que puede utilizar. Lo mismo ocurre en el caso de las variables de entorno del gestor de dependencias Gradle. Después añadimos las carpetas con los enlaces a las dependencias necesarias para añadir los programas del apartado Software. En este mismo apartado instalamos programas como pueden ser Git, Zip y Unzip, Bash, etc… A continuación instalamos la versión de gradle 4.10.2 y el esclavo de Jenkins. Por último, añadimos el ejecutable llamado *entrypoint.sh* que se encuentra en la carpeta recursos, creamos el volumen de los trabajos de jenkins, configuramos el gestor de versiones Git y ejecutamos el fichero entrypoint.sh. Este mismo fichero se asegura que hemos fijado las variables de entorno necesarias, y el código es el siguiente:

```
#!/bin/bash


BASE_DIR=$(cd `dirname $0`; pwd)


if [ -z "${SLAVE_RUNNABLE}" ]; then
  SLAVE_RUNNABLE=/usr/share/jenkins/slave.jar
fi
if [ ! -f ${SLAVE_RUNNABLE} ]; then
  echo "${SLAVE_RUNNABLE} not found."
  exit 1
```

```
fi


# WORKSPACE
if [ -z ${SLAVE_WORKDIR} ]; then
  SLAVE_WORKDIR=${BASE_DIR}/jenkins
fi
if [ ! -d ${SLAVE_WORKDIR} ]; then
  mkdir ${SLAVE_WORKDIR}
fi
if [ -z "${JAVA_VM_MEM_MIN}" ]; then
  JVM_MIN_MEM=512
fi
if [ -z "${JAVA_VM_MEM_MAX}" ]; then
  JVM_MAX_MEM=4096
fi
if [ ${JAVA_VM_MEM_MIN} -gt ${JAVA_VM_MEM_MAX} ]; then
  echo "[`date`] [WARN ] JAVA_VM_MEM_MIN is bigger than JAVA_VM_MEM_MAX"
  JAVA_VM_MEM_MAX=${JAVA_VM_MEM_MIN}
fi

JAVA_OPTS="${JAVA_OPTS} -Dfile.encoding=utf-8 -Duser.timezone=Europe/Berlin"
JAVA_OPTS="${JAVA_OPTS} -Xmx${JAVA_VM_MEM_MAX}m -Xms${JAVA_VM_MEM_MIN}m"

echo "java ${JAVA_OPTS} -jar ${SLAVE_RUNNABLE} $@"

java ${JAVA_OPTS} -jar ${SLAVE_RUNNABLE} -workDir ${SLAVE_WORKDIR} "$@"
```

Una vez que tenemos todas las características añadidas, solo nos queda instalar nuestra imagen, y ejecutarla para convertirla en un agente de nuestro servidor de automatización Jenkins. Para ello tenemos que ubicarnos en el directorio de nuestro proyecto con la consola de comandos de PowerShell y utilizar el comando `docker build`:

```
PS C:\> cd .\Users\xxxx\Proyectos\TFG\
PS C:\Users\xxxx\Proyectos\TFG> docker build -t esclavo_minion .
Sending build context to Docker daemon   89.6kB
Step 1/14 : FROM primetoninc/jdk:1.8
1.8: Pulling from primetoninc/jdk
af4b0a2388c6: Pull complete
7af5e2bf94ec: Pull complete
b36ca92263b4: Pull complete
Digest: sha256:f2e7a6fb89dce5a99a1fcf3b4a6d802417b4b926b086fd7fa11cf02173cd74de
Status: Downloaded newer image for primetoninc/jdk:1.8
 ---> f4b4fccc65bb
Step 2/14 : LABEL maintainer="Rodrigo Sánchez Alonso - UNIVERSIDAD DE VALLADOLDID -
rodrigosanchezalonso@gmail.com"
 ---> Running in ad8093b34b57
```

```
Removing intermediate container ad8093b34b57

 ---> ba14e58614a7

Step 3/14 : ENV SLAVE_WORKDIR=/opt/jenkins    SLAVE_RUNNABLE=/usr/share/jenkins/slave.jar
JAVA_VM_MEM_MIN=512     JAVA_VM_MEM_MAX=4096

 ---> Running in b0d420668065

Removing intermediate container b0d420668065

 ---> 7ff53d6e6e1e

Step 4/14 : ENV GRADLE_HOME=/opt/gradle-4.10.2

 ---> Running in ccbfb5274f07

Removing intermediate container ccbfb5274f07

 ---> 1dd844debf40

Step 5/14 : ENV PATH="${PATH}:${GRADLE_HOME}/bin"

 ---> Running in 56b794b27a1f

Removing intermediate container 56b794b27a1f

 ---> 16a29286b1e4

Step 6/14 : ADD resources/yum.repos.d/epel.repo /etc/yum.repos.d/epel.repo

 ---> a69f9851a3d1

Step      7/14      :      RUN      curl      --fail      --location      --retry      3
http://opensource.wandisco.com/centos/7/git/x86_64/wandisco-git-release-7-2.noarch.rpm
-o /tmp/git-release-7-2.noarch.rpm     && rpm -ivh /tmp/git-release-7-2.noarch.rpm     &&
\rm -f /tmp/git-release-7-2.noarch.rpm     && yum clean all     && yum -y install unzip
wget git bash curl zip py-pip openssh unix2dos libuuid which

 ---> Running in f8b25fc33641

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current

                                 Dload  Upload   Total   Spent    Left  Speed

100  4464  100  4464    0     0  12630      0 --:--:-- --:--:-- --:--:-- 12645

warning: /tmp/git-release-7-2.noarch.rpm: Header V4 DSA/SHA1 Signature, key ID 3bbf077a:
NOKEY

Preparing...                          ########################################

Updating / installing...

wandisco-git-release-7-2              ########################################

Loaded plugins: fastestmirror, ovl

Cleaning repos: WANdisco-git base epel extras updates

Cleaning up everything

Maybe you want: rm -rf /var/cache/yum, to also free up space taken by orphaned data from
disabled or removed repos

Cleaning up list of fastest mirrors

Loaded plugins: fastestmirror, ovl

Determining fastest mirrors

 * base: repo.nixval.com

 * extras: mirror.gadix.com

 * updates: repo.nixval.com

Package zip-3.0-11.el7.x86_64 already installed and latest version

No package py-pip available.

Resolving Dependencies

--> Running transaction check

---> Package bash.x86_64 0:4.2.46-29.el7_4 will be updated

---> Package bash.x86_64 0:4.2.46-31.el7 will be an update
```

```
---> Package curl.x86_64 0:7.29.0-42.el7_4.1 will be updated
---> Package curl.x86_64 0:7.29.0-51.el7 will be an update
--> Processing Dependency: libcurl = 7.29.0-51.el7 for package: curl-7.29.0-51.el7.x86_64
---> Package dos2unix.x86_64 0:6.0.3-7.el7 will be installed
---> Package git.x86_64 0:2.18.0-1.WANdisco.402 will be installed
--> Processing Dependency: perl-Git = 2.18.0-1.WANdisco.402 for package: git-2.18.0-1.WANdisco.402.x86_64
--> Processing Dependency: perl >= 5.008 for package: git-2.18.0-1.WANdisco.402.x86_64
--> Processing Dependency: perl(Getopt::Long) for package: git-2.18.0-1.WANdisco.402.x86_64

[...]

---> Package perl-HTTP-Tiny.noarch 0:0.033-3.el7 will be installed
---> Package perl-parent.noarch 1:0.225-244.el7 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package                 Arch    Version                     Repository     Size
================================================================================
Installing:
 dos2unix                x86_64  6.0.3-7.el7                 base           74 k
 git                     x86_64  2.18.0-1.WANdisco.402       WANdisco-git  7.1 M
 openssh                 x86_64  7.4p1-16.el7                base          510 k
 which                   x86_64  2.20-7.el7                  base           41 k
Updating:
 bash                    x86_64  4.2.46-31.el7               base          1.0 M
 curl                    x86_64  7.29.0-51.el7               base          269 k
 libuuid                 x86_64  2.23.2-59.el7_6.1           updates        82 k
 unzip                   x86_64  6.0-19.el7                  base          170 k
 wget                    x86_64  1.14-18.el7_6.1             updates       547 k
Installing for dependencies:
 fipscheck               x86_64  1.4.1-6.el7                 base           21 k
 fipscheck-lib           x86_64  1.4.1-6.el7                 base           11 k
 groff-base              x86_64  1.22.2-8.el7                base          942 k
 less                    x86_64  458-9.el7                   base          120 k
 libedit                 x86_64  3.0-12.20121213cvs.el7      base           92 k
 libsmartcols            x86_64  2.23.2-59.el7_6.1           updates       140 k
 openssh-clients         x86_64  7.4p1-16.el7                base          655 k
 perl                    x86_64  4:5.16.3-294.el7_6          updates       8.0 M
 perl-Carp               noarch  1.26-244.el7                base           19 k
 perl-Encode             x86_64  2.51-7.el7                  base          1.5 M
 perl-Error              noarch  1:0.17020-2.el7             base           32 k
 perl-Exporter           noarch  5.68-3.el7                  base           28 k
```

```
 perl-File-Path          noarch  2.09-2.el7                base           26 k
 perl-File-Temp          noarch  0.23.01-3.el7             base           56 k
 perl-Filter             x86_64  1.49-3.el7                base           76 k
 perl-Getopt-Long        noarch  2.40-3.el7                base           56 k
 perl-Git                noarch  2.18.0-1.WANdisco.402      WANdisco-git   23 k
 perl-HTTP-Tiny          noarch  0.033-3.el7               base           38 k
 perl-PathTools          x86_64  3.40-5.el7                base           82 k
 perl-Pod-Escapes        noarch  1:1.04-294.el7_6          updates        51 k
 perl-Pod-Perldoc        noarch  3.20-4.el7                base           87 k
 perl-Pod-Simple         noarch  1:3.28-4.el7              base          216 k
 perl-Pod-Usage          noarch  1.63-3.el7                base           27 k
 perl-Scalar-List-Utils  x86_64  1.27-248.el7              base           36 k
 perl-Socket             x86_64  2.010-4.el7               base           49 k
 perl-Storable           x86_64  2.45-3.el7                base           77 k
 perl-Text-ParseWords    noarch  3.29-4.el7                base           14 k
 perl-Time-HiRes         x86_64  4:1.9725-3.el7            base           45 k
 perl-Time-Local         noarch  1.2300-2.el7              base           24 k
 perl-constant           noarch  1.27-2.el7                base           19 k
 perl-libs               x86_64  4:5.16.3-294.el7_6        updates       688 k
 perl-macros             x86_64  4:5.16.3-294.el7_6        updates        44 k
 perl-parent             noarch  1:0.225-244.el7           base           12 k
 perl-podlators          noarch  2.5.1-3.el7               base          112 k
 perl-threads            x86_64  1.87-4.el7                base           49 k
 perl-threads-shared     x86_64  1.43-6.el7                base           39 k
 rsync                   x86_64  3.1.2-6.el7_6.1           updates       404 k
Updating for dependencies:
 libblkid                x86_64  2.23.2-59.el7_6.1         updates       181 k
 libcurl                 x86_64  7.29.0-51.el7             base          221 k
 libmount                x86_64  2.23.2-59.el7_6.1         updates       182 k
 nspr                    x86_64  4.19.0-1.el7_5            base          127 k
 nss                     x86_64  3.36.0-7.1.el7_6          updates       835 k
 nss-pem                 x86_64  1.0.3-5.el7_6.1           updates        74 k
 nss-softokn             x86_64  3.36.0-5.el7_5            base          315 k
 nss-softokn-freebl      x86_64  3.36.0-5.el7_5            base          222 k
 nss-sysinit             x86_64  3.36.0-7.1.el7_6          updates        62 k
 nss-tools               x86_64  3.36.0-7.1.el7_6          updates       515 k
 nss-util                x86_64  3.36.0-1.1.el7_6          updates        78 k
 util-linux              x86_64  2.23.2-59.el7_6.1         updates       2.0 M

Transaction Summary
================================================================================
Install  4 Packages (+37 Dependent packages)
Upgrade  5 Packages (+12 Dependent packages)

Total download size: 28 M
```

```
Downloading packages:
Delta RPMs disabled because /usr/bin/applydeltarpm not installed.
warning:                  /var/cache/yum/x86_64/7/WANdisco-git/packages/perl-Git-2.18.0-
1.WANdisco.402.noarch.rpm: Header V4 DSA/SHA1 Signature, key ID 3bbf077a: NOKEY
Public key for perl-Git-2.18.0-1.WANdisco.402.noarch.rpm is not installed
--------------------------------------------------------------------------------
Total                                              4.9 MB/s |  28 MB  00:05
Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-WANdisco
Importing GPG key 0x3BBF077A:
 Userid      : "WANdisco (http://WANdisco.com - We Make Software Happen...) <software-
key@wandisco.com>"
 Fingerprint: 69c1 be83 da54 cbed 6889 72f8 e9f0 e922 3bbf 077a
 From       : /etc/pki/rpm-gpg/RPM-GPG-KEY-WANdisco
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Warning: RPMDB altered outside of yum.
  Updating   : bash-4.2.46-31.el7.x86_64                                    1/75
  Updating   : nspr-4.19.0-1.el7_5.x86_64                                   2/75
                                [...]
  Verifying  : nss-softokn-freebl-3.28.3-8.el7_4.x86_64                    75/75

Installed:
  dos2unix.x86_64 0:6.0.3-7.el7        git.x86_64 0:2.18.0-1.WANdisco.402
  openssh.x86_64 0:7.4p1-16.el7        which.x86_64 0:2.20-7.el7

Dependency Installed:
  fipscheck.x86_64 0:1.4.1-6.el7
  fipscheck-lib.x86_64 0:1.4.1-6.el7
  groff-base.x86_64 0:1.22.2-8.el7
  less.x86_64 0:458-9.el7
  libedit.x86_64 0:3.0-12.20121213cvs.el7
  libsmartcols.x86_64 0:2.23.2-59.el7_6.1
  openssh-clients.x86_64 0:7.4p1-16.el7
  perl.x86_64 4:5.16.3-294.el7_6
  perl-Carp.noarch 0:1.26-244.el7
  perl-Encode.x86_64 0:2.51-7.el7
  perl-Error.noarch 1:0.17020-2.el7
  perl-Exporter.noarch 0:5.68-3.el7
  perl-File-Path.noarch 0:2.09-2.el7
  perl-File-Temp.noarch 0:0.23.01-3.el7
  perl-Filter.x86_64 0:1.49-3.el7
  perl-Getopt-Long.noarch 0:2.40-3.el7
  perl-Git.noarch 0:2.18.0-1.WANdisco.402
  perl-HTTP-Tiny.noarch 0:0.033-3.el7
```

```
  perl-PathTools.x86_64 0:3.40-5.el7
  perl-Pod-Escapes.noarch 1:1.04-294.el7_6
  perl-Pod-Perldoc.noarch 0:3.20-4.el7
  perl-Pod-Simple.noarch 1:3.28-4.el7
  perl-Pod-Usage.noarch 0:1.63-3.el7
  perl-Scalar-List-Utils.x86_64 0:1.27-248.el7
  perl-Socket.x86_64 0:2.010-4.el7
  perl-Storable.x86_64 0:2.45-3.el7
  perl-Text-ParseWords.noarch 0:3.29-4.el7
  perl-Time-HiRes.x86_64 4:1.9725-3.el7
  perl-Time-Local.noarch 0:1.2300-2.el7
  perl-constant.noarch 0:1.27-2.el7
  perl-libs.x86_64 4:5.16.3-294.el7_6
  perl-macros.x86_64 4:5.16.3-294.el7_6
  perl-parent.noarch 1:0.225-244.el7
  perl-podlators.noarch 0:2.5.1-3.el7
  perl-threads.x86_64 0:1.87-4.el7
  perl-threads-shared.x86_64 0:1.43-6.el7
  rsync.x86_64 0:3.1.2-6.el7_6.1

Updated:
  bash.x86_64 0:4.2.46-31.el7                curl.x86_64 0:7.29.0-51.el7
  libuuid.x86_64 0:2.23.2-59.el7_6.1         unzip.x86_64 0:6.0-19.el7
  wget.x86_64 0:1.14-18.el7_6.1

Dependency Updated:
  libblkid.x86_64 0:2.23.2-59.el7_6.1
  libcurl.x86_64 0:7.29.0-51.el7
  libmount.x86_64 0:2.23.2-59.el7_6.1
  nspr.x86_64 0:4.19.0-1.el7_5
  nss.x86_64 0:3.36.0-7.1.el7_6
  nss-pem.x86_64 0:1.0.3-5.el7_6.1
  nss-softokn.x86_64 0:3.36.0-5.el7_5
  nss-softokn-freebl.x86_64 0:3.36.0-5.el7_5
  nss-sysinit.x86_64 0:3.36.0-7.1.el7_6
  nss-tools.x86_64 0:3.36.0-7.1.el7_6
  nss-util.x86_64 0:3.36.0-1.1.el7_6
  util-linux.x86_64 0:2.23.2-59.el7_6.1

Complete!
Removing intermediate container f8b25fc33641
 ---> 575d165bfb71
Step 8/14 : RUN curl --fail --location --retry 3
http://services.gradle.org/distributions/gradle-4.10.2-bin.zip -o /tmp/gradle.zip
&& unzip /tmp/gradle.zip -d /opt/ && \rm -f /tmp/gradle.zip
 ---> Running in 50ee0df2259f
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 74.7M  100 74.7M    0     0  4440k      0  0:00:17  0:00:17 --:--:-- 4839k
Archive:  /tmp/gradle.zip
   creating: /opt/gradle-4.10.2/
  inflating: /opt/gradle-4.10.2/getting-started.html
  inflating: /opt/gradle-4.10.2/LICENSE
   creating: /opt/gradle-4.10.2/init.d/
  inflating: /opt/gradle-4.10.2/init.d/readme.txt
  inflating: /opt/gradle-4.10.2/NOTICE
   [...]
  inflating: /opt/gradle-4.10.2/lib/plugins/plexus-interpolation-1.14.jar
  inflating: /opt/gradle-4.10.2/lib/plugins/plexus-component-annotations-1.5.5.jar
  inflating: /opt/gradle-4.10.2/lib/plugins/opentest4j-1.0.0.jar
  inflating: /opt/gradle-4.10.2/lib/plugins/plexus-cipher-1.7.jar
Removing intermediate container 50ee0df2259f
 ---> 1ad61efeb09a
Step 9/14 : RUN mkdir -p /usr/share/jenkins          && curl --create-dirs -sSLo
${SLAVE_RUNNABLE}                          https://repo.jenkins-ci.org/public/org/jenkins-
ci/main/remoting/3.26/remoting-3.26.jar      && chmod 755 /usr/share/jenkins      && chmod
644 ${SLAVE_RUNNABLE}
 ---> Running in 9b5285a4531f
Removing intermediate container 9b5285a4531f
 ---> 042e45d9669e
Step 10/14 : ADD resources/entrypoint.sh /opt/
 ---> 028b42b55cd9
Step 11/14 : RUN chmod +x /opt/entrypoint.sh
 ---> Running in 0cc6b78af2e1
Removing intermediate container 0cc6b78af2e1
 ---> b15a09e0b444
Step 12/14 : VOLUME ${SLAVE_WORKDIR}
 ---> Running in c8f02aefa2d4
Removing intermediate container c8f02aefa2d4
 ---> 09c786d3b3ca
Step 13/14 : ADD resources/.gitconfig /root/.gitconfig
 ---> f8d4af474dd9
Step 14/14 : ENTRYPOINT [ "/opt/entrypoint.sh" ]
 ---> Running in f376749a89ef
Removing intermediate container f376749a89ef
 ---> 8eaf42b06024
Successfully built 8eaf42b06024
Successfully tagged esclavo_minion:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker
host. All files and directories added to build context will have '-rwxr-xr-x' permissions.
It is recommended to double check and reset permissions for sensitive files and
directories.
```

He omitido algunos fragmentos de la salida por pantalla debido a su extensión. En este momento tenemos la imagen en nuestro software Docker bajo el nombre “esclavo_minion”. Una vez que hemos creado y configurado nuestro agente de Jenkins, tenemos que ejecutar el siguiente comando:

```
docker run -d --name esclavo_minion --rm esclavo_minion -jnlpUrl
http://192.168.1.7:8080/computer/AGENTE/slave-agent.jnlp -workDir "/home/jenkins/agent"
```

Una vez lanzado, el contenedor estará corriendo dentro de nuestro Docker y se conectará con el servidor de automatización Jenkins, como podemos ver en la figura 28:

Figura 28: Agente conectado en Jenkins frente a agente no conectado en Jenkins

En estos momentos, tenemos un agente con todas las características necesarias para ejecutar un proyecto de pruebas (Docker Docs, 2018)

Bitbucket y Git

Bitbucket es un servicio de alojamiento basado en web, para los proyectos que utilizan el sistema de control de versiones Mercurial y Git. Es este servicio el que utilizaré para alojar los proyectos tanto de testing con los ejemplos correspondientes, como el proyecto de infraestructura con la imagen del agente del servidor de automatización. El sistema de control de versiones utilizado es *Git*, y necesitamos un control de versiones puesto que estamos simulando un entorno real en el que se encuentra un jefe de proyecto y varios programadores. Para poder trabajar todos a la vez en el mismo proyecto, cada programador trabaja en una rama del proyecto, y una vez terminado suben el código a la rama principal, bajo una previa revisión del jefe de proyecto. También usaremos git para clonar el proyecto en el agente del servidor de automatización (Bitbucket Docs, 2020). Los comandos de git más frecuentes son los siguientes (Git Documentation, 2020):

- git config: Se utiliza para establecer una configuración específica del usuario, como sería el caso del email, nombre de usuario, tipo de formato, etc…
- git init: Este comando se utiliza para crear un repertorio de Git.
- git add: Se utiliza para agregar archivos al directorio local del índice.
- git clone: Este comando se utiliza para clonar determinado repositorio localmente. En nuestro caso, clonamos nuestro proyecto mediante el siguiente comando

```
git clone https://sanalor@bitbucket.org/sanalor/tfgrodrigo.git
```

- git commit: Este comando se utiliza para guardar los cambios en la cabecera. Ten en cuenta que cualquier cambio comprometido no afectará al repertorio remoto.
- git status: Este comando muestra la lista de los archivos que se han cambiado junto con los archivos que están por ser añadidos o comprometidos.
- git push: Este comando es uno de los más básicos. Un simple push envía los cambios que se han hecho en la rama principal de los repertorios remotos que están asociados con el directorio que está trabajando.
- git checkout: Se puede utilizar para crear ramas o cambiar entre ellas. Por ejemplo, éste comando crea una rama y se cambia a ella:

```
command git checkout -b <branch-name>
```

- git remote: Se utiliza para conectar a un repositorio remoto. El siguiente comando muestra los repositorios remotos que están configurados actualmente:

```
git remote -v
```

- git branch: Este comando se utiliza para listar, crear o borrar ramas.
- git pull: Para poder fusionar todos los cambios que se han hecho en el directorio local en el que se está trabajando. Se traen los cambios que estén en el repositorio remoto, y se puede hacer por ramas específicas.
- git merge: Este comando se utiliza para fusionar una rama con otra rama activa.
- git diff: Este comando se utiliza para hacer una lista de conflictos.
- git fetch: Este comando le permite al usuario buscar todos los objetos de un repositorio remoto que actualmente no reside en el directorio local en el que se está trabajando.

Serenity

Serenity, según la definición que nos proporciona su documentación oficial, es una librería de código abierto, que nos ayuda a escribir test de aceptación y regresión mejores, más efectivos y de forma más rápida, y utiliza esos tests para producir reportes y documentación en vivo. Ésta información no solo muestra los resultados de las pruebas, sino que también evalúa las características que se han probado.

A modo de ejemplo, podemos observar la apariencia de un reporte real en la figura 29:

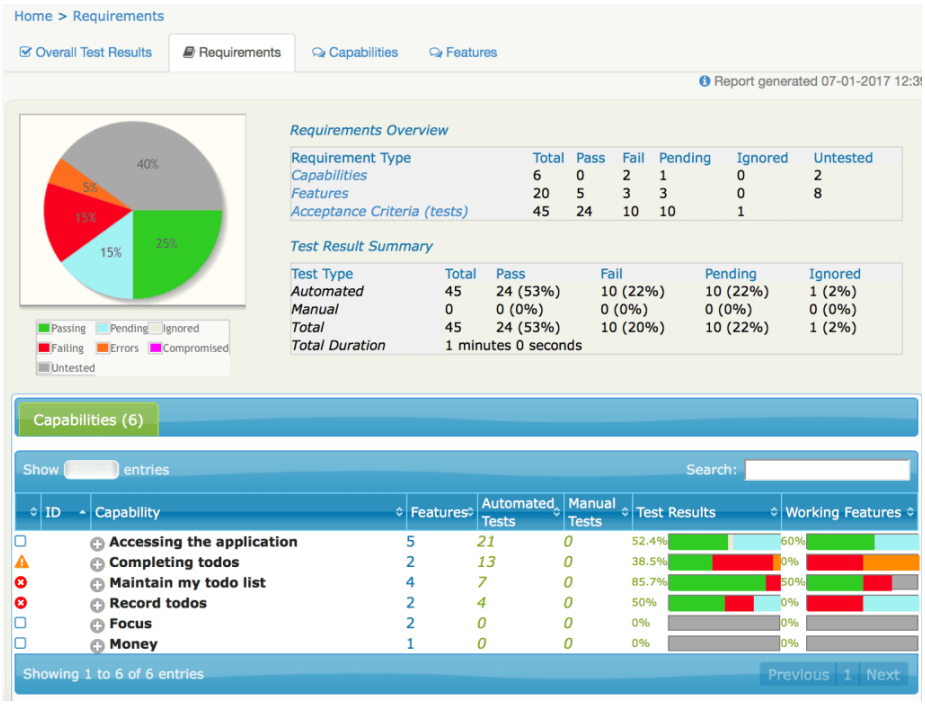


Figura 29: Apariencia de los reportes de la automatización de las pruebas de una aplicación de negocio.

Antes de entrar en detalles, voy a presentar el concepto de Desarrollo Basado en el Comportamiento (en inglés, Behaviour Driven Development BDD). Este concepto encaja bien en las metodologías ágiles, ya que generalmente en ellas se especifican los requerimientos como historias de usuario. Estas historias de usuario deberán tener sus criterios de aceptación, y de ahí se desprenden pruebas de aceptación, las cuales pueden ser escritas directamente en lenguaje "*Gherkin*". *Gherkin* es un lenguaje común, que lo puede escribir alguien sin conocimientos en programación, pero que lo puede comprender también un programa, de forma tal de utilizarlo como especificación de pruebas.

Típicamente, estas pruebas se van a guardar en archivos ".feature", los cuales deberían estar versionados junto al código fuente del sistema que se está probando. A continuación, se presenta, a modo de ejemplo, un escenario simple tomado de Cucumber, en este mismo proyecto:

```
Feature: Pruebas sobre la aplicacion gratuita PetStore


  Scenario: Añadir una nueva mascota a la tienda virtual
    When Yo creo la mascota con las siguientes caracteristicas
      | campo        | valor     |
      | id           | 456000    |
      | categoryId   | 456001    |
      | categoryName | UVA       |
      | name         | Doggie    |
      | photoUrls    | prueba    |
      | tagId        | 455999    |
      | tagName      | prueba    |
      | status       | available |
    Then El codigo de respuesta es "200"
```

```
  Scenario: Comprobar que hemos añadido la mascota correctamente
    When Yo consulto los datos de la mascota con id "456000"
    Then El codigo de respuesta es "200"
    And El nombre de la mascota corresponde con "Doggie"
```

En estos archivos se especifica:

- *Feature*: nombre de la funcionalidad que vamos a probar, el título de la prueba.
- *Scenario*: habrá uno por cada prueba que se quiera especificar para esta funcionalidad.
- *Given*: precondiciones de las pruebas. En este caso no utilizo ninguna precondición.
- *When*: se especifican las acciones que se van a ejecutar.
- *Then*: se especifica el resultado de la prueba.

Puede haber un *feature* por archivo y este contendrá distintos escenarios de prueba. Gherkin soporta muchos idiomas. Si bien lo más común es verlo en inglés, se puede hacer en español también como en el ejemplo mostrado anteriormente.

Éste lenguaje es simplemente texto con algunas palabras clave y algo de estructura. Hay herramientas como Cucumber (la que utilizo en el presente proyecto) y JBehave (la que utilizaría el lenguaje Python) que permiten implementar una capa de conexión entre esa especificación de prueba y la interfaz del sistema que se quiere probar, pudiendo así utilizar eso como los pasos de una prueba automatizada (Serenity, 2019).

Selenoid

Selenoid es una implementación en *Go* del código original de Selenium. *Go* es un lenguaje de programación de código libre (Toledo, 2017). Selenium es un proyecto general que encapsula una gran variedad de herramientas y librerías que nos permiten desarrollar automatizaciones web. Para ser más específico, provee infraestructura para la especificación W3C *WebDriver specification*. Ésta es una plataforma e interfaz de código neutral compatible con la mayoría de navegadores web. La fuente de código de Selenium cumple la licencia Apache 2.0.

Selenoid utiliza Docker para lanzar los navegadores. Primero tenemos que crear un fichero de configuración para los diferentes navegadores.

```
{
  "chrome": {
    "default": "75.0",
    "versions": {
      "75.0": {
        "image": "selenoid/vnc:chrome_75.0",
        "port": "4444",
        "path": "/"
      },
      "76.0": {
        "image": "selenoid/vnc:chrome_76.0",
        "port": "4444",
```

```
        "path": "/"
      }
    }
  }
}
```

Éste es el archivo *browsers.config* que tengo ubicado en la siguiente carpeta en mi ordenador C:\Users\rodri\selenoid. A la hora de iniciar Selenoid, necesitaré apuntar a esta configuración para que podamos utilizar los navegadores y versiones deseados. Mediante los siguientes comandos, podemos cargar las imágenes de esos navegadores en nuestro repositorio de imágenes locales de Docker:

```
docker pull selenoid/vnc:chrome_74.0

docker pull selenoid/vnc:chrome_75.0
```

Una vez que tengo todas las imágenes cargadas en mi repositorio local, es hora de iniciar Selenoid mediante el siguiente comando, el cual explicaré a continuación:

```
PS C:\Users\rodri\selenoid> docker run -d --name selenoid -p 4444:4444 -v
//var/run/docker.sock:/var/run/docker.sock -v /c/Users/rodri/selenoid:/etc/selenoid:ro
aerokube/selenoid:latest
```

El puerto que utilizo para el servicio de Selenoid es el 4444, indico la carpeta en la que se encuentra mi configuración y le indico que utilice la última imagen disponible de Selenoid.

Ya tenemos instalado Selenoid en un contenedor de Docker llamado “selenoid”. Ahora voy a implementar una interfaz para poder ver nuestras automatizaciones. En este caso me he descargado un fichero ejecutable del repositorio oficial de selenoid (https://github.com/aerokube/selenoid-ui/releases). Una vez tenemos el fichero en nuestra carpeta, tenemos que dirigirnos a esa ruta en nuestra consola y ejecutar el siguiente comando:

```
C:\Users\rodri\selenoid> ./selenoid-ui -selenoid-uri http://localhost:4444 -listen
localhost:8090
```

Le estamos indicando a la interfaz en qué dirección y puerto tenemos instalado nuestro Selenoid y exportamos la interfaz a la misma dirección, pero forzamos el puerto 8090. Esto es importante ya que el puerto que utiliza por defecto la interfaz de selenoid es el puerto 8080, y es dónde tenemos instalado Jenkins previamente.

Ahora podemos abrir la siguiente dirección web en cualquier navegador para poder ver la interfaz de selenoid: http://localhost:8090. En la siguiente imagen podemos ver un ejemplo de mi interfaz:

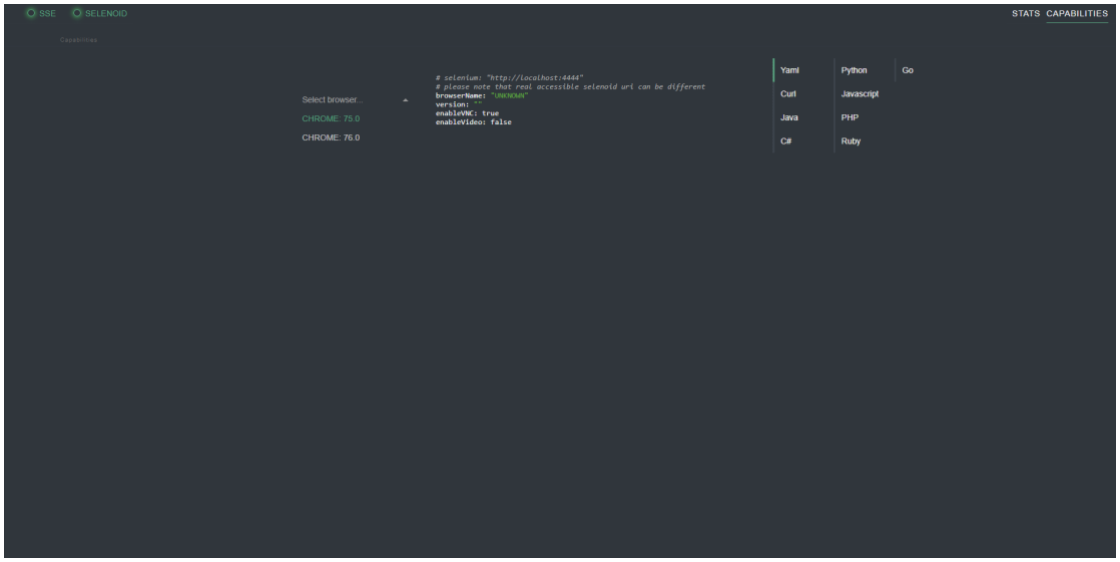


Figura 30: Interfaz de Selenoid donde podemos apreciar las diferentes versiones instaladas de los navegadores de nuestro archivo de configuración.

A lo largo del presente proyecto se presentarán imágenes de pruebas de automatización web a modo de ejemplo (Andryashin, Krutov, Merkushev & Aerokube community, 2020).

Rest-Assured

Rest-Assured es una librería que utilizo en Java para poder testear y validar servicios *REST*. En el lenguaje de programación Java es más difícil utilizar servicios *REST* que en otros lenguajes dinámicos como Ruby o Groovy. Esta librería nos trae la simplicidad de utilizar esos lenguajes en el dominio de Java. Por ejemplo, si tu servidor devuelve el siguiente fichero *JSON* cuando hacemos una petición a la dirección http://localhost:8080/lotto/{id}:

```
{
   "lotto":{
      "lottoId":5,
      "winning-numbers":[2,45,34,23,7,5,3],
      "winners":[
         {
            "winnerId":23,
            "numbers":[2,45,34,23,3,5]
         },
         {
            "winnerId":54,
            "numbers":[52,3,12,11,18,22]
         }
      ]
   }
}
```

Podemos utilizar esta librería para validar cosas interesantes de la respuesta, como, por ejemplo:

```
@Test public void
lotto_resource_returns_200_with_expected_id_and_winners() {


    when().
            get("/lotto/{id}", 5).
    then().
            statusCode(200).
            body("lotto.lottoId", equalTo(5),
                 "lotto.winners.winnerId", hasItems(23, 54));


}
```

De esta manera estamos comprobando que el código de la respuesta es 200 (*OK*), que el ID recibido es igual a 5, y que los ID de los ganadores tiene los ítems 23 y 54 (Rest-Assured, 2019) (Johan Haleby, 2020).

4. Pruebas a servicios REST

El servicio sobre el que he creado automatizaciones se llama *Petstore.* Es un servicio gratuito cuya descripción se encuentra en el siguiente enlace: https://petstore.swagger.io/. Las pruebas las presento en el presente documento en el lenguaje de negocio. El repositorio del proyecto de pruebas a servicios es el siguiente: https://sanalor@bitbucket.org/sanalor/tfgrodrigo.git.

La estructura la podemos observar en la siguiente imagen:

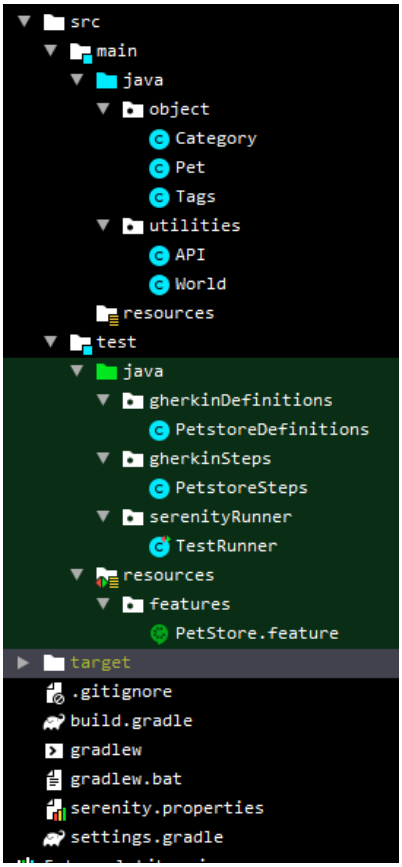


Figura 31: Estructura del proyecto de pruebas a servicios REST

En el primer nivel nos encontramos con los siguientes ficheros:

- .gitignore: Se utiliza para añadir reglas de excepción de ficheros y carpetas en GIT.
- build.gradle: Se utiliza para cargar todas las dependencias del proyecto, diferentes aspectos de configuración de gradle y las diferentes tareas que vamos a realizar.
- gradlew y gradlew.bat: Archivos creados automáticamente por gradle a la hora de empezar un proyecto desde cero.
- serenity.properties: Archivo de configuración de serenity en el que podemos configurar el nombre del proyecto que aparecerá en los reportes o los colores de la consola.
- En la carpeta fuente nos encontramos con todas las clases de Java y los diferentes archivos de definición de los escenarios en cucumber.

El lenguaje de negocio utilizado es llamado *Cucumber*. Nos presenta una visión a muy alto nivel del tipo de pruebas que se van a realizar. Presento a continuación las pruebas creadas en este proyecto, incluidas en el fichero “PetStore.feature”:

```
Feature: Pruebas sobre la aplicacion gratuita PetStore


  Scenario: Añadir una nueva mascota a la tienda virtual
    When Yo creo la mascota con las siguientes caracteristicas
      | campo        | valor     |
      | id           | 456000    |
      | categoryId   | 456001    |
      | categoryName | UVA       |
      | name         | Doggie    |
      | photoUrls    | prueba    |
      | tagId        | 455999    |
      | tagName      | prueba    |
      | status       | available |
    Then El codigo de respuesta es "200"


  Scenario: Comprobar que hemos añadido la mascota correctamente
    When Yo consulto los datos de la mascota con id "456000"
    Then El codigo de respuesta es "200"
    And El nombre de la mascota corresponde con "Doggie"


  Scenario: Actualizar alguna carácterística de la mascota
    When Yo consulto los datos de la mascota con id "456000"
    Then El codigo de respuesta es "200"
    And Yo modifico el parametro "name" a "Rodolfo" de la mascota con id "456000"
    And El codigo de respuesta es "200"


  Scenario: Compruebo el nuevo parámetro de la mascota
    When Yo consulto los datos de la mascota con id "456000"
    Then El codigo de respuesta es "200"
    And Yo consulto el parametro "name" y tiene que ser "Rodolfo"


  Scenario: Borrar la mascota
    When Yo borro los datos de la mascota con id "456000"
    Then El codigo de respuesta es "200"


  Scenario: Comprobar que la mascota no se está disponible
    When Yo consulto los datos de la mascota con id "456000"
    Then El codigo de respuesta es "404"
```

Para poder ejecutar este proyecto tenemos que contar con los siguientes requerimientos mínimos:

- Gradle en su quinta versión: https://gradle.org/
- Java SE Development Kit 8: https://www.oracle.com/java/technologies/javase-jdk8-downloads.html
- Conexión pública a internet.

Descargamos el Proyecto en la ruta que deseemos, y nos dirigimos a la ruta fuente del proyecto con la consola de comandos, y ejecutamos el siguiente comando:

```
> gradle clean test aggregate
```

Comenzarán a ejecutarse las diferentes pruebas y podemos ver un reporte rápido de la ejecución en la consola. Una vez han terminado de ejecutarse las pruebas, podemos ver el reporte generado por serenity en la siguiente ruta: target/site/serenity/index.html. Lo abrimos con nuestro explorador favorito y tendremos acceso a todos los reportes (SmartBear, 2017) (Johan Haleby, 2020) (Gradle, 2020). Incluyo un ejemplo de estos en la figura 32:

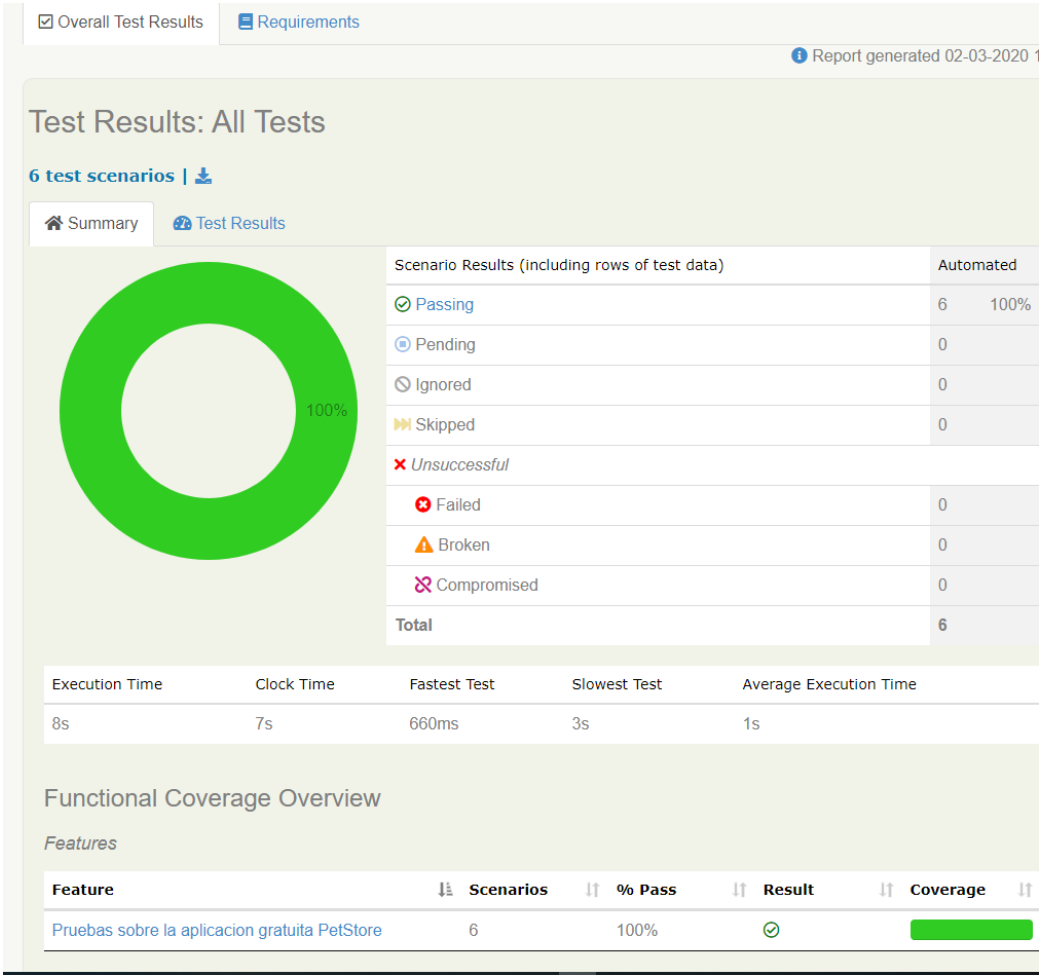


Figura 32: Ejemplo de los reportes generados por serenity en el proyecto de pruebas a servicios REST.

5. Pruebas a servicios WEB

Para la parte de pruebas a servicios web el contenido es muy limitado debido a aspectos legales que me impiden crear automatizaciones en muchas páginas web. En esta ocasión he generado una única automatización en la página web de la biblioteca de la Universidad de Valladolid. He realizado una búsqueda de un libro y he comprobado que se encuentra en la base de datos de la Universidad y que se encuentra disponible en alguno de los diferentes centros. La estructura del proyecto se puede visitar en el siguiente repositorio: https://sanalor@bitbucket.org/sanalor/tfgrodrigo-fe.git.

La estructura del proyecto en esta ocasión la podemos ver en la figura 32:

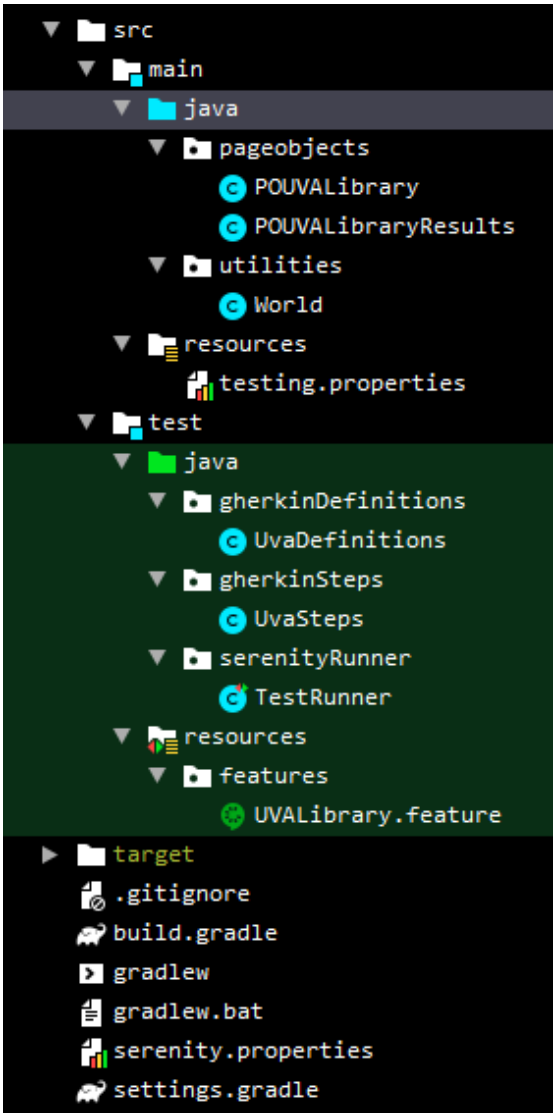


Figura 33: Estructura del proyecto de pruebas a servicios web.

Voy a describir las diferencias que encontramos con el proyecto de pruebas a servicios REST en términos de estructura:

- El archivo UVALibrary.feature contine la definición de las pruebas en el lenguaje de negocio Cucumber.

- El archivo serenity.properties incluye la configuración necesaria para apuntar a los drivers de exploradores que tenemos montado en el servidor. Utilizando Selenoid, previamente descrito.

Os presento a continuación la única prueba automatizada que incluye el proyecto de automatización de pruebas de servicios web. Es el contenido del fichero UVALibrary.feature:

```
Feature: Pruebas sobre la página web de la biblioteca de la Universidad de Valladolid


  Scenario Outline: Comprobar que un libro se encuentra en la Biblioteca
    When Yo estoy en la pagina web de la biblioteca de la Universidad de Valladolid
    And Yo escribo "<titulo>" en el catalogo de libros y le doy click en buscar
    Then Yo compruebo que obtengo "<titulo>" en la pagina de resultados
    And Yo compruebo que el libro esta disponible en cualquier Biblioteca
    Examples:
      | titulo     |
      | Cenicienta |
```

Para la ejecución de este proyecto contamos con dos opciones. La primera de ellas es ejecutar las pruebas con un webdriver ubicado en nuestro ordenador. En mi caso he descargado el ultimo webdriver estable de Chrome en esta ubicación: https://chromedriver.chromium.org/downloads. El archivo serenity.properties tiene el siguiente aspecto:

```
# Aparece en la primera parte de los reportes
serenity.project.name = UVA_TFGRodrigo_FE
serenity.console.colors = true


serenity.maintain.session = true


# Ejecucion local
webdriver.driver = chrome
chrome.switches = --start-maximized
# Web driver paths
webdriver.chrome.driver = C:\\tools\\chromedriver.exe
webdriver.ie.driver = C:\\tools\\IEDriverServer.exe


# Ejecucion remota (Selenoid)
#webdriver.remote.driver = chrome
#webdriver.remote.url = http://192.168.0.16:4444/wd/hub
#chrome.switches=--start-maximized,--incognito
#serenity.driver.capabilities=enableVNC:true;name:UVA_TFGRodrigo_FE


# Opciones de Serenity
#                          ->                          http://thucydides.info/docs/serenity-
staging/#_serenity_system_properties_and_configuration
serenity.restart.browser.for.each = scenario
#   Screenshots   options=(FOR_EACH_ACTION,   BEFORE_AND_AFTER_EACH_STEP,   FOR_FAILURES,
DISABLED)
```

```
#                          ->                          http://thucydides.info/docs/serenity-staging/#_configuring_when_screenshots_are_taken
serenity.take.screenshots=FOR_EACH_ACTION
```

Como podemos observer en el código, el webdriver de Chrome lo he ubicado en la ruta C:/tools/chromedriver.exe, y desde este archivo de configuración apunto a ese ejecutable. Para poder ejecutar el proyecto localmente tenemos que tener los siguientes requerimientos mínimos:

- Gradle en su quinta versión: https://gradle.org/
- Java SE Development Kit 8: https://www.oracle.com/java/technologies/javase-jdk8-downloads.html
- Conexión pública a internet.
- WebDriver de vuestro explorador web favorito e indicar su ruta en el fichero serenity.properties.

En este punto, nos ubicamos en la ruta raíz del proyecto con la consola de comandos y ejecutamos el siguiente comando:

```
> gradle clean test aggregate -DdataProperties=testing.properties
```

Esta vez contamos con un fichero de propiedades y hay que incluirlo en las opciones de gradle a la hora de ejecutar el proyecto.

Por el otro lado, si queremos ejecutar el proyecto contra un webdriver ubicado y configurado en otro servidor tenemos que comentar las líneas referentes a las propiedades locales y descomentar las líneas referentes a las propiedades remotas, en el archivo serenity.properties, de la siguiente manera:

```
# Aparece en la primera parte de los reportes
serenity.project.name = UVA_TFGRodrigo_FE
serenity.console.colors = true


serenity.maintain.session = true


# Ejecucion local
#webdriver.driver = chrome
#chrome.switches = --start-maximized


#Web driver paths
#webdriver.chrome.driver = C:\\tools\\chromedriver.exe
#webdriver.ie.driver = C:\\tools\\IEDriverServer.exe


# Ejecucion remota (Selenoid)
webdriver.remote.driver = chrome
webdriver.remote.url = http://192.168.0.16:4444/wd/hub
chrome.switches=--start-maximized,--incognito
serenity.driver.capabilities=enableVNC:true;name:UVA_TFGRodrigo_FE


# Opciones de Serenity
```

```
#                          ->                          http://thucydides.info/docs/serenity-staging/#_serenity_system_properties_and_configuration

serenity.restart.browser.for.each = scenario

#  Screenshots  options=(FOR_EACH_ACTION,  BEFORE_AND_AFTER_EACH_STEP,  FOR_FAILURES, DISABLED)

#                          ->                          http://thucydides.info/docs/serenity-staging/#_configuring_when_screenshots_are_taken

serenity.take.screenshots=FOR_EACH_ACTION
```

Como podemos observar, ahora apuntamos nuestra ejecución a un webdriver remoto ubicado en la dirección local 192.168.0.16 en el puerto 4444. Aunque incluya una dirección local, funcionaría de la misma manera utilizando mi dirección IP pública 84.127.158.128, o cualquier dirección en la que tengamos instalado el servicio de Selenoid o Selenium Hub. Una vez contemos con los requerimientos mínimos de ejecución, y hayamos cambiado el aspecto del fichero serenity.properties, podemos ejecutar las pruebas ejecutando el mismo comando:

```
> gradle clean test aggregate -DdataProperties=testing.properties
```

Comenzará a ejecutarse la prueba y podemos ver un reporte rápido de la ejecución en la consola. Una vez han terminado de ejecutarse las pruebas, podemos ver el reporte generado por serenity en la siguiente ruta: target/site/serenity/index.html. Lo abrimos con nuestro explorador favorito y tendremos acceso a todos los reportes (Andryashin et al., 2020). Incluyo un ejemplo de estos en la figura 33:

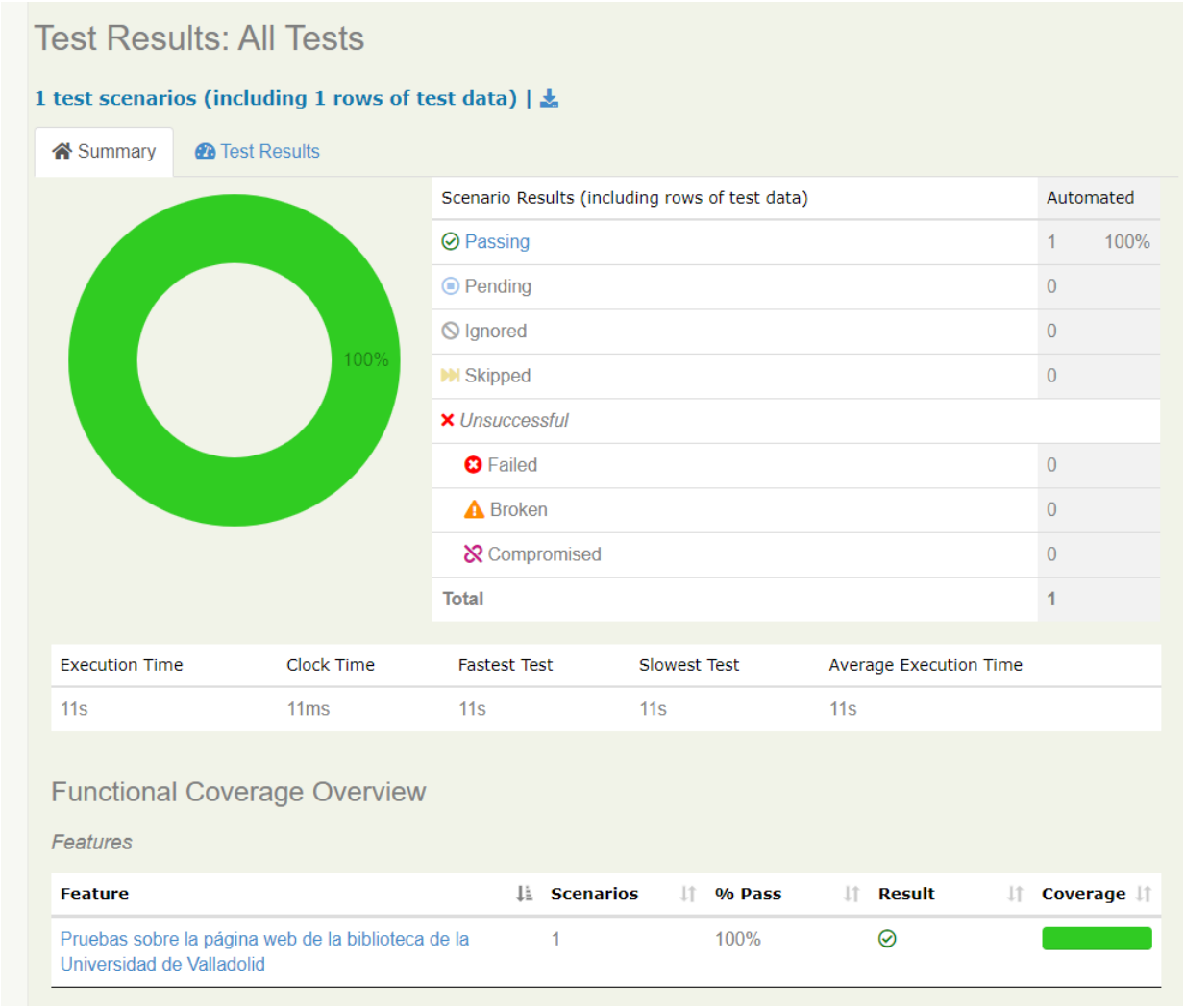


Figura 34: Ejemplo de los reportes generados por serenity en el proyecto de pruebas a servicios web.

6. Coste Económico

En el presente apartado se presenta un coste aproximado de la ejecución del proyecto. Los costes se basarán tanto en elementos hardware y software como en horas trabajadas. La cantidad final dependerá de la cantidad de tests integrados en el proyecto y de su complejidad. El oráculo de pruebas de cualquier aplicación es infinito así que hay que seleccionar las pruebas que aporten mayor valor a la calidad de la aplicación.

A continuación, podemos ver una tabla con el desglose del presupuesto:

Concepto	Detalles	Costes
Horas trabajadas	12€ por hora 200 horas	2400 €
Costes oficina y gastos generales	Alquiler + Equipamiento + Luz + Agua + Internet	500 €
Software	IntelliJ IDEA Git Docker Bitbucket Jenkins	0 €
	Total	**2900 €**

Desglose detallado del presupuesto realizado:

- Salario del programador: Se ha establecido un coste de 12 euros por hora trabajada. El tiempo empleado para la realización del actual proyecto es de 200 horas.
- Costes oficina y gastos generales: Estos son el coste del alquiler del puesto de trabajo, el equipamiento (ordenador, teclado, ratón, pantalla), la luz, el agua y la conexión a Internet. Con un contrato de trabajo de jornada completa corresponderían a 25 días laborables.
- Programas software necesarios para el desarrollo de la arquitectura y el proyecto de testing: Todos los programas utilizados son gratuitos. Algunos de ellos incluyen versión de pago, pero no se necesita para la finalidad del proyecto.

7. Conclusiones y líneas futuras

En este capítulo se presentan una serie de conclusiones que se han ido recogiendo a lo largo del presente documento, y no solo durante la realización de éste, puesto que incluyo experiencia profesional paralela a su realización. La mayoría de ellas las he adquirido en el mundo profesional o hablando con terceras personas referentes a esta ciencia de estudio.

Para finalizar, expongo una serie de líneas futuras o diferentes líneas de investigación. Puesto que las pruebas y la calidad es un campo en desarrollo, muchas de las ideas han sido compartidas con el gremio y han sido extraídas de la experiencia. La realización del proyecto me ha ayudado a aclarar muchos aspectos relacionados con la Ingeniería de pruebas y de ahí he extraído alguna idea que expondré en el apartado final.

Conclusiones

Durante la realización del proyecto he tenido una maduración en este campo de estudio que me permite extraer la primera de las conclusiones, para mí una de las más importantes. Cuando un ingeniero de pruebas encuentra un fallo en la funcionalidad, en el seguimiento de los datos o realizando pruebas no funcionales, como pueden ser unas simples pruebas de rendimiento, no ha inventado algo que no estuviera ya en el código del producto a probar. El principal objetivo de un *tester* es el intercambio de información con sus compañeros de proyecto (de información que ya estaba incluida en el código). Es por eso por lo que la relación con los desarrolladores tiene que ser impecable y la comunicación tiene que ser lo más asertiva posible.

Otro aspecto importante es el contexto que puede proporcionar un desarrollador de automatización de pruebas a un analista de negocio, o cualquier actor involucrado en el proyecto que no tiene conocimiento acerca de lenguajes de programación. Las pruebas escritas en el primer nivel, el nivel del negocio cuyo nombre técnico es *gherkin*, deben poder ser comprendidas por cualquier actor involucrado en el proyecto. Cuando entra un nuevo miembro en el equipo, éste primer nivel de negocio sirve para entender en profundidad la herramienta a desarrollar o probar (según tu rol en el equipo) en términos de funcionalidad.

Es imposible concretar una medida numérica de la calidad del software. Un desarrollador de automatización de pruebas tiene que ser capaz de encontrar una cantidad de defectos de cualquier tipo (véase tipos de defectos en “Tipos de prueba” dentro del segundo apartado “Proceso de pruebas de Software”) pero no debe ser la última barrera para decidir si la aplicación debe salir a producción. Existen roles para esa decisión en todos los proyectos.

Los dos ejemplos de automatización de pruebas se pueden ejecutar en cualquier sistema que cuente con los requerimientos mínimos, lo que nos permite ampliar toda la funcionalidad a todo el proceso de desarrollo de un software o un producto hardware. Todo esto se expondrá en el apartado referente a las líneas futuras de investigación.

Toda la teoría que nos presenta la ingeniería de pruebas nos ofrece ideas sobre cómo desarrollar nuestra estrategia de pruebas. No existe ningún estándar en cuanto a qué probar. Cada ingeniero abstrae cada una de las ideas adquiridas en su experiencia y en la interacción con las diferentes herramientas o actores, y las utilizará en su siguiente estrategia de pruebas. Esto no quiere decir que realizar un alto número de pruebas en un producto sea el comportamiento adecuado. Siempre se tiene que cubrir la mayor parte del producto posible con las pruebas más simples. Lo más complejo no siempre es lo que más rinde.

Estimar el trabajo de un ingeniero de pruebas en base a fallos encontrados es una mala práctica. Siempre que se pueda abstraer el trabajo de otros actores como los desarrolladores será una buena práctica para estimar. Es por eso por lo que se utiliza el tanto por ciento de producto cubierto con nuestras pruebas, ya sean funcionales o no funcionales. Como queremos generar un entorno de integración continua, cada una de nuestras pruebas se ejecutarán antes de desplegar una nueva funcionalidad del producto de desarrollo, para evitar regresiones de código de desarrollo o para prevenir futuros fallos.

Líneas futuras

Cualquier trabajo de investigación en el que se ha aportado un mínimo de entusiasmo contribuye a despejar muchas incógnitas sobre el tema tratado, como ya he aclarado en el anterior capítulo, pero de forma paralela genera nuevas preguntas y líneas de trabajo.

Se puede trabajar con otros lenguajes como puede ser Python. A mí me parece muy interesante debido a su fácil usabilidad ya que es un lenguaje de muy alto nivel. Si se quisiera implementar pruebas con este lenguaje, el lenguaje de negocio utilizado se llama *Behave*. Sustituye al lenguaje *gherkin* que he utilizado en la realización del presente proyecto: *Cucumber*. En cuanto a evaluación personal, prefiero utilizar Python para tareas de *scripting* para proveer a los diferentes actores de información del producto de forma diferente. La seguridad que nos ofrece JAVA es una de las características más importantes a seguir en mi línea de trabajo.

Una de las mejores implementaciones con que podemos contar es un administrador de tareas del estilo de Jira, Trello o Asana. Con ella podemos administrar y gestionar nuestros tests, permitiéndonos una serie de posibilidades como:

- Gestionar cada una de nuestras pruebas con etiquetas o con diferentes cajas para que nos sea más fácil su mantenimiento y actualización.
- Con las etiquetas podemos ejecutar un conjunto de pruebas dependiendo de lo que se quiera probar. Si se va a integrar un nuevo componente a nuestro producto se ejecutarán las pruebas relacionadas con los componentes a integrar. Una vez que se ha comprobado la integración, podemos ejecutar todas las pruebas al producto ya integrado para comprobar que no se han transmitido errores a otros componentes, o por la misma integración.
- Podemos relacionar nuestras entradas de pruebas con cada una de las entradas de desarrollo, para poder ver qué pruebas están asignadas a una determinada parte de desarrollo en un simple vistazo.

Un desarrollador de automatización de pruebas debe estar en continuo aprendizaje acerca de las tecnologías existentes en el mercado y las actualizaciones de las ya conocidas. Las empresas que no externalizan el *testing* empiezan a apostar por incluir a los desarrolladores de pruebas en cuestiones relacionadas con desarrollo de pruebas cuando el marco de pruebas está muy avanzado. El conocimiento que se obtiene sobre la aplicación te permite aportar nuevas ideas a los actores de negocio y a los mismos desarrolladores de software.

Bibliografía:

- BDD, Cucumber y Gherkin. Desarrollo dirigido por comportamiento. Raúl Hernandez (2018). Obtenido de https://www.genbeta.com/desarrollo/bdd-cucumber-y-gherkin-desarrollo-dirigido-por-comportamiento. Fecha de última consulta: Julio de 2020.

- Crispin, L. & Gregory, J. (2008). *Agile testing: a practical guide for testers and agile teams.* Indiana: Pearson Education. ISBN: 978-0321534460.

- Docker Docs. Orientation and setup (2019). Obtenido de https://docs.docker.com/get-started/. Fecha de última consulta: Julio de 2020.

- Elfriede Dustin, Thom Garrett, Bernie Gauf. (2009). *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Estados Unidos: Addison-Wesley Professional. ISBN: 9780321619594.

- Europa Press (2019). *Veinte años sin noticias de la Mars Polar Lander.* Recuperado de https://www.europapress.es/ciencia/misiones-espaciales/noticia-veinte-anos-noticias-mars-polar-lander-20191203140417.html. Fecha de última consulta: Julio de 2020.

- Garzas, J. ¿Qué es Jenkins? Explicado en menos de 10 min para quienes no lo conocen de nada (Mayo 2014). Obtenido de https://www.javiergarzas.com/2014/05/jenkins.html. Fecha de última consulta: Julio de 2020.

- Git Documentation (2020). Obtenido de https://git-scm.com/docs. Fecha de última consulta: Julio de 2020.

- Gradle Build Tool. Gradle Docs (2020). Obtenido de https://gradle.org/. Fecha de última consulta: Julio de 2020.

- International Software Testing Qualifications Board, ISTQB (2020*). Foundation Level Syllabus for Certified Tester*. url: https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html. Fecha de última consulta: Julio de 2020.Certified Tester. Foundation Level Syllabus (2018). Obtenido de https://www.istqb.org/downloads/send/51-ctfl2018/208-ctfl-2018-syllabus.html

- Jenkins Docs. Jenkins User Documentation (2018). Obtenido de https://jenkins.io/doc/. Fecha de última consulta: Julio de 2020.

- Patton, Ron (2001). *Software Testing.* Washington: Sams. ISBN: 9780672319839.

- Paul C. Jorgensen (2013). *Software Testing: A Craftsman's Approach, Fourth Edition*. Reino Unido: Auerbach Publications. ISBN: 9781315360409.

- Proyecto de automatización de pruebas a servicios REST. Rodrigo Sánchez (Mayo 2020). Obtenido de https://sanalor@bitbucket.org/sanalor/tfgrodrigo.git. Fecha de última consulta: Julio de 2020.

- Proyecto de automatización de pruebas a servicios WEB. Rodrigo Sánchez (Marzo de 2020). Obtenido de https://sanalor@bitbucket.org/sanalor/tfgrodrigo-fe.git. Fecha de última consulta: Julio de 2020.

- Rest-Assured Docs (2019). Obtenido de http://rest-assured.io/. Fecha de última consulta: Julio de 2020.

- Rest-Assured Usage. Johan Haleby (Marzo 2020). Obtenido de https://github.com/rest-assured/rest-assured/wiki/Usage. Fecha de última consulta: Marzo de 2020.

- Rus, Cristian (2020). *El día que la NASA estrelló una sonda en Marte porque alguien se olvidó de usar el sistema métrico*. Recuperado de https://www.xataka.com/espacio/dia-que-nasa-estrello-sonda-marte-porque-alguien-se-olvido-usar-sistema-metrico. Fecha de última consulta: Julio de 2020.

- Samuel, Henry (2009). *Air France Airbus A380 superjumbo forced to return to New York after technical hitch* .Recuperado de: https://www.telegraph.co.uk/news/worldnews/europe/france/6693877/Air-France-Airbus-A380-superjumbo-forced-to-return-to-New-York-after-technical-hitch.html. Fecha de última consulta: Julio de 2020.

- Selenoid. Alexander Andryashin, Ivan Krutov, Kirill Merkushev and the Aerokube community (2020). Obtenido de: https://aerokube.com/selenoid/latest/. Fecha de última consulta: Junio de 2020.

- Swagger Petstore. Supported by SmartBear (2017). Obtenido de https://petstore.swagger.io/. Fecha de última consulta: Julio de 2020.

- The code collaboration platform for modern software teams. Bitbucket Docs (2020). Obtenido de https://bitbucket.org/product/features. Fecha de última consulta: Julio de 2020.

- The Serenity Reference Manual. Serenity Docs (2019). Obtenido de http://thucydides.info/docs/serenity-staging/. Fecha de última consulta: Julio de 2020.

- Toledo, F. (2017). ¿Qué es BDD?. Obtenido de https://www.federico-toledo.com/que-es-bdd/. Fecha de última consulta: Julio de 2020.

- VENTAJAS DE LA INTEGRACIÓN CONTINUA / DISTRIBUCIÓN CONTINUA (CI/CD). ¿Qué son la integración / distribución continuas (CI/CD)? (2019). Obtenido de https://www.redhat.com/es/topics/devops/what-is-ci-cd. Fecha de última consulta: Julio de 2020.

- Whittaker, J.A. (2002). *How to Break Software: A Practical Guide to Testing.* Michigan: Addison Wesley. ISBN: 9780201796193.