



TALLER DE TESTING Y CALIDAD DE SOFTWARE

U2. Desarrollo y Ejecución de casos de prueba

Semana 5





ESCUELA DE CONSTRUCCIÓN E INGENIERIA

Director: Marcelo Lucero

ELABORACIÓN

Experto disciplinar: Aída Villamar Gallardo.

Diseño instruccional: Carla Silva Alvarado.

VALIDACIÓN

Experto disciplinar: Andrés del Alcázar

Jefa de Diseño Instruccional: Alejandra San Juan Reyes.

EQUIPO DE DESARROLLO

AIEP

AÑO

2021



Tabla de contenidos

| | |
|---|----|
| 1. Antecedentes históricos asociados al surgimiento de pruebas de sistema y testeo de software..... | 4 |
| 1.1.- Antecedentes históricos del surgimiento de las pruebas de sistema..... | 4 |
| 2. Diferencias de pruebas estáticas, dinámicas, manuales y automáticas en proyectos de desarrollo de software..... | 7 |
| 2.1.- Pruebas estáticas y dinámicas..... | 7 |
| 2.2.-Tipos de pruebas según su ejecución: manuales y automáticas | 9 |
| 3. Pruebas de compatibilidad, regresión e integración, de caja negra y de caja blanca, en proyectos de desarrollo de software. | 12 |
| 3.2.-Pruebas de regresión: | 14 |
| 3.3.-Pruebas de integración:..... | 17 |
| 3.4.-Enfoques de pruebas de sistema: caja negra, caja blanca o testing aleatorio | 20 |
| 4. Pruebas de sistema en diferentes niveles, considerando proyectos de desarrollo de software. | 26 |
| 4.1.- -Niveles de pruebas: unitarias, integración, sistema y aceptación | 26 |
| Conclusiones..... | 31 |
| Referencias bibliográficas | 31 |



Aprendizaje esperado de la semana

Aplican diferentes tipos de pruebas de sistema para la mejora de calidad de proyectos de desarrollo de software.

Introducción

- ¿Qué es una prueba manual?
- ¿Qué es una prueba automatizada?
- ¿Cuál es la diferencia entre las pruebas manuales y las pruebas automatizadas?

Es posible que las respuestas a estas preguntas en lo referente a teoría ya estemos en condiciones de responderlas, pues las comenzamos a revisar en semanas anteriores. Ahora bien, importante es saber las respuestas pero, más importante aún, es saber aplicar cada una de estas pruebas en la etapa y en la forma adecuada.

Por lo tanto, esta semana vamos a profundizar en cada una de ellas y estudiaremos la forma de incorporarlas en nuestros proyectos, pudiendo obtener resultados concretos, que permitirán evaluar la calidad del trabajo realizado.

1. Antecedentes históricos asociados al surgimiento de pruebas de sistema y testeo de software.


1.1.- Antecedentes históricos del surgimiento de las pruebas de sistema

“Las pruebas de software son el proceso de ejecutar un programa con la intención de encontrar errores” — Glenford J. Myers.

La etapa de pruebas hoy en día es uno de los puntos más importantes en el desarrollo de software, cabe destacar que no solo en esta área sino en general se ha convertido en una de las etapas más importantes, tanto en el desarrollo de sistemas como en la fabricación de distintos productos, lo que se busca es garantizar la calidad que están recibiendo nuestros clientes y para ello se considera la aplicación estricta y rigurosa de diversas pruebas.

La finalidad en esta etapa es poder garantizar la calidad que va a recibir el cliente, minimizando la cantidad de fallas evitando así algo tan importante como es la mala imagen o una pérdida en la confianza que estamos entregando.

Ahora bien, si vemos el origen de esta necesidad, encontraremos que es netamente un tema económico, el mercado del desarrollo de software



comenzó a crecer y corregir los fallos en etapas avanzadas y esto suponía un costo muy elevado, en cambio, la detección en etapas tempranas resulta muy rentable.

Según Fuente: (Hernandez, 2019):

Aunque términos como “testing” o “pruebas de calidad” puedan parecer novedosos, las pruebas de software se han venido realizando y evolucionando desde los inicios de la computación, diferenciando cinco importantes periodos según los modelos de pruebas más influyentes:

Periodo de debugging (1947–1956)

En 1947, se acuñan los términos “bug” y “debugging”. Grace Murray, una científica de la universidad de Harvard que trabajaba con la computadora Mark II, detectó que una polilla se había quedado pegada en un relé provocando que éste no hiciera contacto. Detalló el incidente en la bitácora de trabajo, pegando la polilla con cinta adhesiva como evidencia y refiriéndose a la polilla como el “bug” (bicho) causante del error, y a la acción de eliminar el error como “debugging”.

Por entonces, las pruebas que se realizaban estaban enfocadas en el hardware debido a que no estaba tan desarrollado como hoy en día y su fiabilidad era imprescindible para el correcto funcionamiento del software.

El término debugging estaba asociado a la aplicación de un parche para un determinado fallo como fase dentro de la etapa de desarrollo del software, y es por ello, que las pruebas que se realizaban eran únicamente de índole correctiva tomando ciertas medidas con el propósito de hacer funcionar el programa.

Es en 1949, cuando Alan Turing escribe su primer artículo acerca de realizar comprobaciones sobre un programa y después en 1950, en el artículo “Turing test”, expone la situación de cómo un software debe adaptarse a los requisitos de un proyecto y el comportamiento de una máquina o un sistema de referencia (lógica humana) debe ser indistinguible.

Periodo de demostración (1957–1978)

En 1957, Charles Baker expone la necesidad de desarrollar pruebas que garanticen que el software satisface los requisitos prediseñados (testing) además de la funcionalidad del programa (debugging).

El desarrollo de pruebas adquirió mayor importancia debido a que más aplicaciones, más caras y complejas estaban siendo desarrolladas, y el coste de solucionar todas esas deficiencias afectaba y suponía un claro riesgo para la rentabilidad del proyecto. Además, que la experiencia en este nuevo sector estaba aumentando.

Se puso especial foco en el aumento del número y la calidad de pruebas, y por primera vez se empezó a relacionar la calidad de un producto con el estado de la fase de testing.



El objetivo era demostrar que el programa hacía lo que anteriormente se había dicho que debía hacer, usando parámetros esperados y reconocibles.

Periodo de destrucción (1979–1982)

En 1979, Glenford J. Myers, con la definición entregada al comienzo, cambia radicalmente el procedimiento para la detección de fallos en el programa. La preocupación de Myers era que al perseguir la meta de demostrar que un programa no tiene fallos, uno podría seleccionar subconscientemente datos de prueba que tienen una baja probabilidad de causar fallos en el programa, en cambio, si el objetivo es demostrar que un programa tiene fallos, nuestros datos de prueba tendrán una mayor probabilidad de detectarlos y tendremos más éxito en las pruebas y por ende en la calidad del software.

“Las pruebas de software son el proceso de ejecutar un programa con la intención de encontrar errores” — Glenford J. Myers.

Desde este momento, las pruebas intentarán demostrar que un programa no funciona como debería, al contrario de como se venía haciendo hasta entonces, conduciendo a nuevas técnicas de testing y análisis.

Periodo de evaluación (1983-1987)

En 1983 se propone una metodología que integra actividades de análisis, revisión y prueba durante el ciclo de vida del software con el fin de obtener una evaluación del producto según avanza el desarrollo.

La fase de testing se reconoce como una fase íntegra en la elaboración de un producto, adquiriendo especial importancia por la aparición de herramientas para el desarrollo de pruebas automatizadas, lo cual mejoró notablemente la eficiencia.

Periodo de prevención (1988 — actualidad)

En 1988, William Hetzel publica “The Growth of Software Testing” donde se redefine el concepto de prueba como la planificación, diseño, construcción, mantenimiento y ejecución de pruebas y entornos de prueba. Esta etapa, principalmente se reflejó en la aparición de la fase de testing, en la fase más temprana de la elaboración de un producto, la etapa de planificación.

Podemos seguir la evolución de la ingeniería de pruebas de software examinando los cambios en el modelo de procesamiento de pruebas y el nivel de profesionalismo a lo largo de los años. La definición actual de una buena práctica de prueba de software implica una metodología preventiva.”
— William C. Hetzel

Es importante visualizar como la etapa de prueba de software ha evolucionado y ha ido emergiendo desde su ausencia hasta su presencia continua en todo el ciclo de vida.



Si imagináramos la fase completa del desarrollo como una línea finita, donde el principio es la planificación y el final es la monitorización del producto vendido, veríamos como la fase de pruebas se ha ido desplazando hacia la izquierda. Apareció siendo una etapa posterior a la fabricación, más tarde fue una etapa previa a la venta y actualmente se encuentra en toda la fase completa. Esta práctica es conocida como Shift-Left, y está ocurriendo en muchos otros campos.

Fuente: (Hernandez, 2019)

2. Diferencias de pruebas estáticas, dinámicas, manuales y automáticas en proyectos de desarrollo de software.

2.1.- Pruebas estáticas y dinámicas

Las pruebas estáticas y las pruebas dinámicas son dos tipos de técnicas de prueba que se complementan entre sí. Las pruebas estáticas se realizan en las etapas iniciales del desarrollo del software, mientras que las pruebas dinámicas se llevan a cabo después de la finalización del proceso de desarrollo. Las pruebas dinámicas apuntan a la validación del software, en cambio, las pruebas estáticas apuntan a la verificación del software.

Podemos considerar a las pruebas como la unión de diversas actividades dentro del ciclo de vida del software, se encuentran vinculadas con la etapa de planificación, con el diseño y la posterior evaluación del software con el objetivo de detectar defectos y establecer si el sistema cumple o no con los requerimientos definidos inicialmente.



Cuadro comparativo

| Base de comparación | Pruebas estáticas | Prueba dinámica |
|--------------------------------|--------------------------|---|
| Básico | No ejecuta el software. | Es necesaria la ejecución del software. |
| Costo | Bajo | Alto |
| Cobertura de estados de cuenta | 100% | 50% |
| Consumo de tiempo | Menos | Más |
| Descubre | Gran variedad de errores | Tipos limitados de errores |
| Realizado | Antes de la compilación | Solo cuando los ejecutables están disponibles |

Tabla 1: Cuadro comparativo pruebas estáticas & pruebas dinámicas

Fuente: (LIVING-IN-BELGIUM, 2021)

Como ya hemos visto, las pruebas estáticas son una manera de analizar código fuente, podemos considerar documentos con los requerimientos, diseño, todo aquello que no es necesario ejecutar, se realizan en una etapa de verificación y son consideradas como una técnica preventiva. A diferencia de las pruebas dinámicas quienes toman datos como entrada y posteriormente se valida contrastando con el resultado esperado, podemos decir que son comparables con las pruebas de validación.

Algo a considerar, al momento de escoger a la persona que va a realizar las pruebas estáticas, es que idealmente no haya participado en el desarrollo del sistema, o escritura de código.

| Herramientas de prueba estática | Herramientas de prueba dinámica |
|--|--|
| Las pruebas estáticas siguen un enfoque simbólico de las pruebas, por lo que no requieren entradas y salidas reales. | Las pruebas dinámicas utilizan los datos en vivo, ya que sabemos que prueban el sistema de software en el momento de la ejecución. |
| 1. Analizadores de flujo: Analiza el flujo de datos de entrada a salida, sea coherente o no. | 1. Conductor de prueba: Se utiliza para insertar los datos en un MUT (módulo en prueba). |
| 2. Pruebas de ruta: Descubre el código ambiguo, no utilizado e | 2. Bancos de prueba: Muestra el programa en ejecución junto con el |

| inalcanzable. | código fuente. |
|---|---|
| 3. Analizadores de cobertura: Confirma que se han probado todas las rutas lógicas que inciden en el software. | 3. Emuladores: Ayuda a emular las partes del sistema de software que se van a desarrollar. |
| 4. Analizadores de interfaz: Inspecciona los efectos de pasar datos entre los módulos. | 4. Analizadores de mutaciones: Esto realiza la prueba para verificar la tolerancia a fallas del sistema para el cual los errores se ingresan intencionalmente en el código. |

Tabla 2: Herramientas pruebas estáticas & Herramientas pruebas dinámicas
Fuente: (LIVING-IN-BELGIUM, 2021)

2.2.-Tipos de pruebas según su ejecución: manuales y automáticas

Dentro del tipo de pruebas según su ejecución, encontramos aquellas que son manuales y aquellas que son automáticas, en ambas existe una persona que cumple el rol de probador, pero que realiza distintas actividades en cada uno de los tipos de prueba.

En las estáticas, encontramos que el analista de control de calidad prueba manualmente el software, a diferencia de las pruebas automatizadas donde el probador escribe scripts de prueba con la finalidad de automatizar la ejecución de la prueba y utiliza herramientas de automatización para desarrollar esos scripts de prueba y validar el software.



Figura 1: Pruebas automatizadas de software
Fuente: (Frias, 2018)

Por otro lado, tenemos que las pruebas manuales, permiten un análisis acabado por parte de un humano, lo que implica un beneficio mayor cuando lo que se busca es mejorar la experiencia de usuario, pero si lo que se necesita es ejecutar de forma repetitiva y por un largo período de tiempo realizar diversos test cases, la opción más viable pasan a ser las pruebas automatizadas.



Figura 2: Pruebas manuales de software
Fuente: (Pngwing, s.f.)

En las pruebas manuales, un probador va aplicando los casos de prueba asociados a todos los escenarios de prueba de forma manual, es decir, sin la ayuda de una herramienta de prueba de automatización. Las pruebas automatizadas son más rápidas y precisas que las pruebas manuales.



| Prueba manual | Pruebas automatizadas |
|--|--|
| Requieren un probador humano para ejecutar los casos de prueba. | Requieren herramientas de automatización para ejecutar los casos de prueba. |
| Se prueba un software manualmente por analistas de control de calidad. | Un probador escribe scripts de prueba para automatizar la ejecución de la prueba. |
| La prueba manual ayuda a identificar los defectos en el software en desarrollo. | Se emplean herramientas de automatización para desarrollar scripts de prueba y poder validar el software. |
| El probador comprueba todas las características esenciales del software | Se ejecutan automáticamente para comparar el resultado obtenido con el resultado esperado de la ejecución. |
| Los probadores de software normalmente requieren experiencia para realizar las pruebas manuales. | Se requiere cierto esfuerzo manual y experiencia para crear scripts de prueba iniciales. |
| Las pruebas requieren mas tiempo para su ejecución y tienen un nivel de complejidad mayor. | La ejecución automatizada de pruebas es más fácil y requiere un tiempo mínimo en comparación con las pruebas manuales. |

Tabla 3: Pruebas manuales & Pruebas automatizadas

Fuente: (STREPHONSAYS, s.f.)

Cuadro comparativo

| Base de comparación | Prueba manual | Pruebas automatizadas |
|---------------------------|--|---|
| Básico | Las pruebas las realizan humanos manualmente. | Las pruebas se realizan a través del hardware / software automáticamente. |
| Precisión y confiabilidad | Moderado y susceptible a errores. | Alto |
| Consumo de tiempo | Más | Menos |
| Costo | Baja, de corta duración. | Alto, pero durante un período prolongado es rentable. |
| Usado para | Pruebas exploratorias, de usabilidad y ad-hoc. | Pruebas de regresión, rendimiento y carga. |
| Acción humana | Necesario | No involucrado |
| Experiencia de usuario | Mejora la experiencia del cliente | La experiencia positiva del cliente no está asegurada. |

Tabla 4: Cuadro comparativo Pruebas manuales & Pruebas automatizadas

Fuente: (LIVING-IN-BELGIUM, LIVING-IN-BELGIUM, 2021)

3. Pruebas de compatibilidad, regresión e integración, de caja negra y de caja blanca, en proyectos de desarrollo de software.

3.1.-Pruebas de compatibilidad:

Son aquellas pruebas por medio de las cuales se va a probar el sistema con distintas plataformas, sistemas operativos, navegadores, configuraciones de hardware y en general con todo aquello que pudiera interactuar el software. Se debe especificar un conjunto de configuraciones de computadores de clientes más frecuentes o posibles y en función de estas preparar las pruebas. En esta instancia podemos utilizar el estudio de factibilidad técnica en el que se define la configuración mínima y optima, considerando ambas en las pruebas que se deben realizar.

La compatibilidad se relaciona con la integración de los sistemas. Para que la información pueda ser compartida y para que los usuarios tengan acceso a sistemas semejantes con interfases comunes, los sistemas tienen que ser compatibles. En este contexto la compatibilidad significa compartir equipos, sistemas operativos y programas. Fuente: (project, s.f.)



Figura 3: Navegadores
Fuente: (Soluciones, 2012)

La doctrina de prueba de configuraciones impone la prueba del sistema en todos los entornos conocidos de hardware y software en los cuales vaya a funcionar. La prueba de compatibilidad asegura una interfaz funcionalmente consistente entre plataformas de software y hardware. Por ejemplo, la interfaz de ventanas puede ser visualmente distinta dependiendo del entorno de implementación, pero los mismos comportamientos básicos del usuario deberían dar lugar a los mismos resultados independientemente del estándar de interfaz de cliente. Fuente: (Roger S. Pressman, 2010)



Figura4: Sistemas operativos
Fuente: (Testing, s.f.)

La mayoría de los programas de escritorio, están pensados para un sistema operativo y una arquitectura en particular, teniendo que desarrollarse versiones paralelas para adaptarse a otras configuraciones. Un software cross-platform, es aquel que puede funcionar sin problemas en esa amplia variedad de soportes.

Con frecuencia, se dice que las aplicaciones y los sitios web son naturalmente cross-platform debido a que generalmente, pueden visualizarse desde todos los navegadores de uso común. Que un sitio web pueda verse en distintos navegadores, no garantiza una plena accesibilidad: cada web browser interpreta el código fuente a su manera (más allá de los estándares), dando lugar a diferencias de visualización y funcionamiento; y hay lenguajes web que aún no son plenamente soportados. De este problema surge una disciplina especialmente orientada al desarrollo web: el cross-browser testing. Se trata de evaluar un sitio o aplicación en distintas combinaciones de navegador, sistema operativo, resolución de pantalla y dispositivo. Como se observa, a pesar del nombre, el análisis incluye mucho más que diferencias entre navegadores (el examen limitado a esta variable se conoce como multi-browser testing). El cross-browser testing no pretende que un sitio web se vea exactamente igual bajo todas las configuraciones posibles. De hecho, esto es prácticamente imposible. En realidad, busca que «funcione de manera equivalente bajo distintas condiciones», más allá de su apariencia o comportamiento externo. Es decir, apunta más al funcionamiento que a la visualización: *un sitio web puede verse de manera similar en dos computadoras diferentes y sólo funcionar adecuadamente en una*. Por eso, si bien los desarrolladores deben procurar que no existan grandes diferencias visuales al acceder un sitio o aplicación web en distintas plataformas, lo fundamental es garantizar que en ambos casos el usuario pueda disfrutar plenamente de sus utilidades. Fuente: (Soluciones, 4r Soluciones, 2019)



Figura 5: Pruebas de compatibilidad de hardware, sistemas operativos y medios

Fuente: (TesterGame, s.f.)

3.2.-Pruebas de regresión:

Son aquellas pruebas cuyo objetivo es descubrir errores o los bugs que hemos visto anteriormente, requerimientos que no se hayan implementado de manera completa o diferencias entre el comportamiento obtenido y el esperado, causado por modificaciones o cambios en el software. En estas pruebas la automatización es lo ideal, también se pueden ejecutar pruebas de regresiones manuales, pero demandan mucho más tiempo y automatizarlas entrega un nivel de confiabilidad mayor. Y, hay que considerar que se ejecutan periódicamente. Además de lo anterior, estas pruebas encargan de verificar que estas modificaciones que se mencionan no hayan tenido un impacto inesperado o afectado de alguna manera al resto del sistema.

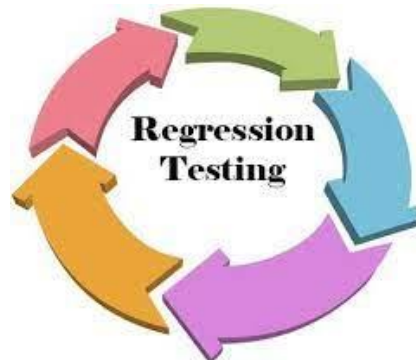


Figura 6: Pruebas de regresión

Fuente: (Fontalvo, 2021)

¿Por qué se llaman pruebas de regresión?

“En un principio pensaba que se refería a regresar a ejecutar las mismas pruebas, ya que se trata un poco de eso. Luego vi que el concepto en realidad está asociado a verificar que lo que estoy probando no tenga regresiones. El tema es que “no tener regresiones” imaginé que se refería a que no haya una regresión en su calidad, o en su funcionalidad, pero



escuché el rumor de que el concepto viene de la siguiente situación: si los usuarios tienen la versión N instalada, y le instalamos la N+1, y esta tiene fallos, nos veremos atormentados al tener que volver a la versión previa, hacer una regresión a la versión N. ¡Queremos evitar esas regresiones! Y por eso se realizan estas pruebas. Tampoco es válido pensar que las pruebas de regresión se limitan a verificar que se hayan arreglado los bugs que se habían reportado, pues es igual de importante ver que lo que antes andaba bien ahora siga funcionando” Fuente: (Rodríguez, 2014).

¿A qué se refiere que las pruebas de regresión se ejecuten frecuentemente?

Está asociado a que cada vez que se realicen modificaciones, ya sea actualizaciones o correcciones sobre nuestro sistema, es necesario volver a ejecutar todas las pruebas que fueron diseñadas para evaluar esas funcionalidades. Entonces, vemos que ejecutar las pruebas de regresión se refiere a *volver a ejecutar estas pruebas que se habían diseñado previamente*.

¿Cuáles son las técnicas comunes de prueba de regresión?



Prueba de regresión unitaria

Inmediatamente después de que se completen los cambios de codificación para una sola unidad, un probador, generalmente el desarrollador responsable del código, vuelve a ejecutar todas las pruebas unitarias aprobadas anteriormente. En los enfoques de desarrollo de prueba primero, como el desarrollo basado en pruebas (TDD) y en los entornos de desarrollo continuo, las pruebas unitarias automatizadas se integran en el código, lo que hace que las pruebas de regresión unitarias sean muy eficientes en comparación con otros tipos de pruebas.

Figura 7: Prueba de regresión unitaria

Fuente: (Ranorex, s.f.)



Prueba de humo

Las pruebas de humo, también llamadas pruebas de verificación de compilación, son una serie de pruebas de regresión de alta prioridad que se ejecutan después de fusionar los cambios de código y antes de cualquier otra prueba. El propósito de las pruebas de humo es garantizar rápidamente que la funcionalidad básica esté en su lugar antes de realizar pruebas adicionales, como la capacidad del AUT para iniciarse y permitir que los usuarios inicien sesión. Es una buena práctica realizar pruebas de humo automáticamente cada vez que se envían cambios de software al repositorio.

Figura 8: Prueba de humo

Fuente: (Ranorex, s.f.)



Prueba de cordura

Las pruebas de cordura son un subconjunto de las pruebas funcionales que examinan solo los módulos modificados. El objetivo de las pruebas de cordura es garantizar que las nuevas funciones funcionen según lo diseñado y que se hayan resuelto los defectos informados en las pruebas anteriores. Las pruebas de cordura tienen dos objetivos: primero, garantizar que los cambios de código tengan el efecto deseado, lo que puede realizarse mediante pruebas exploratorias manuales; en segundo lugar, para detectar posibles regresiones, normalmente utilizando casos de prueba automatizados o con guiones.

Figura 9: Prueba de cordura

Fuente: (Ranorex, s.f.)



Regresión completa

También conocida como técnica de "volver a probar todo", todos los casos de prueba de regresión se ejecutan en una regresión completa. Si bien una regresión completa puede ser tentadora para garantizar que la aplicación se haya probado a fondo, esto es, por definición, costoso y no siempre es práctico, especialmente para versiones menores. En general, puede ser necesario un conjunto de pruebas de regresión completo para versiones importantes con cambios de código importantes, después de cambios de configuración importantes, como un puerto a una nueva plataforma, o para asegurar la compatibilidad con un sistema operativo actualizado. También puede ser necesaria una regresión completa al adaptar una aplicación a otro idioma o cultura, lo que se denomina "localización". Las pruebas de regresión de localización verifican el contenido de la pantalla y la configuración de la interfaz de usuario, como las opciones de visualización de la fecha y la hora, los formatos de moneda y número de teléfono, los conjuntos de caracteres de soporte adecuados, mensajes de error y ayuda correctos, el diseño de los elementos de la GUI y los cuadros de diálogo, y más. Por último, también se puede ejecutar una regresión completa en los casos en que "las regresiones son difíciles de predecir y localizar, como [como regresiones CSS en las que un cambio puede afectar a varias páginas y estados de la interfaz de usuario](#) .

Figura 10: Prueba de regresión completa

Fuente: (Ranorex, s.f.)

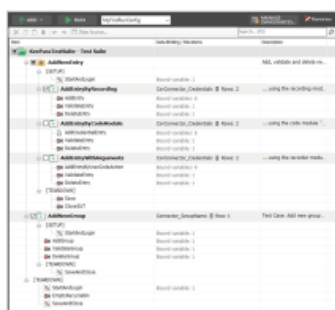


Regresión parcial

Como alternativa a una regresión completa, una estrategia de regresión parcial selecciona solo ciertas pruebas para su ejecución. Las pruebas pueden seleccionarse en función de la prioridad del caso de prueba, o pueden seleccionarse según los módulos particulares de la AUT que cubren. Por ejemplo, si un equipo publica un cambio en los tipos de pago aceptados para una tienda en línea, puede optar por realizar pruebas de regresión para el proceso de pago, pero excluir las pruebas de regresión para otras funciones, como buscar artículos y colocarlos en el carrito. Consulte a continuación las sugerencias sobre cómo priorizar los casos de prueba de regresión para administrar los recursos de prueba limitados.

Figura 11: Prueba de regresión parcial

Fuente: (Ranorex, s.f.)



Many test cases



Few resources

El desafío de las pruebas de regresión: cómo ejecutar suficientes casos de prueba con el tiempo, el personal y los recursos del sistema disponibles.

Figura 12: El desafío de las pruebas de regresión

Fuente: (Ranorex, s.f.)

3.3.-Pruebas de integración:

Verifican que las distintas unidades desarrolladas de un sistema funcionan en conjunto correctamente una vez que se han probado individualmente, esto con el objetivo de comprobar que su interacción es la correcta por medio de sus interfaces y se ajustan a los requerimientos no funcionales definidos.

¿Qué son las pruebas de integración?

Las pruebas de integración se realizan para comprobar las interacciones entre distintos componentes o sistemas tras su integración. Si bien las pruebas unitarias son responsabilidad del autor del código, las pruebas de integración se realizan habitualmente por equipos más especializados en



pruebas. Las pruebas de integración son tan imprescindibles como las unitarias. *El hecho de haber comprobado todos los componentes con resultado satisfactorio no asegura el correcto funcionamiento entre ellos.* Y ahí es donde radica la importancia de asegurar la integración. Para realizar este tipo de pruebas, se hace uso de objetos que simulen en muchos casos los datos que contenga la aplicación como, por ejemplo: Stubs, drivers, subsistemas, bases de datos, API's, microservicios...

Es importante también destacar, en este tipo de pruebas, la importancia de la descripción del error detectado, ya que estos defectos se localizan sobre elementos creados por varios desarrolladores, quienes únicamente conocerán a fondo la parte del código desarrollado por ellos mismos. Fuente: (Moreno, 2019)

La estrategia clásica para la prueba de software de computador comienza con «probar lo pequeño» y funciona hacia fuera haciendo «probar lo grande». Siguiendo la jerga de la prueba de software, se comienza con las pruebas de unidad, después se progresa hacia las pruebas de integración y se culmina con las pruebas de validación del sistema. En aplicaciones convencionales, las pruebas de unidad se centran en las unidades de programa compilables más pequeñas -1 subprograma (por ejemplo, módulo, subrutina, procedimiento, componente)-. Una vez que cada una de estas unidades han sido probadas individualmente, se integran en una estructura de programa, mientras que se ejecutan una serie de pruebas de regresión para descubrir errores, debidos a las interfaces entre los módulos y los efectos colaterales producidos por añadir nuevas unidades. Por último, el sistema se comprueba como un todo para asegurarse de que se descubren los errores en requisitos. Fuente: (Roger S. Pressman, 2010)



Figura 13: Pruebas de integración
Fuente: (Moreno, Oscar Moreno, 2019)

Niveles de pruebas de integración

Encontramos dos niveles dentro de las pruebas de integración.

- **Pruebas de integración de componentes:** Está enfocado en la interacción e interfaces entre las unidades integradas. Se deben efectuar después de las pruebas unitarias. Habitualmente son automatizadas, esto implica una ventaja en lo que a tiempo y dinero se refiere. Para los desarrollos ágiles, la integración de las diferentes unidades, por lo general están dentro de un proceso de integración continua.
- **Pruebas de integración de sistemas:** Estas se centran en la interacción e interfaces entre diferentes sistemas, paquetes o microservicios. En este caso se unirían las pruebas de interacciones con elementos exteriores a la organización, por ejemplo, servicios web (mail, almacenamiento cloud, redes sociales, etc...) Esto supone un reto mayor al no tener todo el control de estos elementos externos. Las pruebas de integración de sistemas se pueden ejecutar posteriormente a las pruebas de sistema, o bien, paralelamente a ellas. Fuente: (Moreno, Oscar Moreno, 2019)

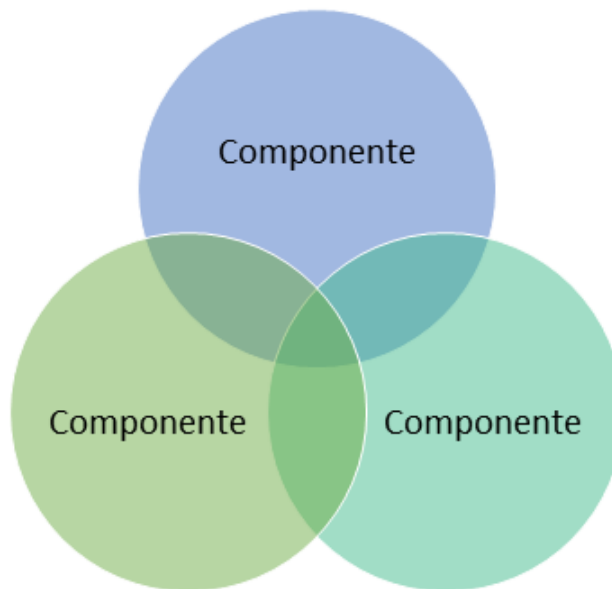


Figura 14: Integración entre componentes.

Fuente: (JCAMILORADA, 2014)

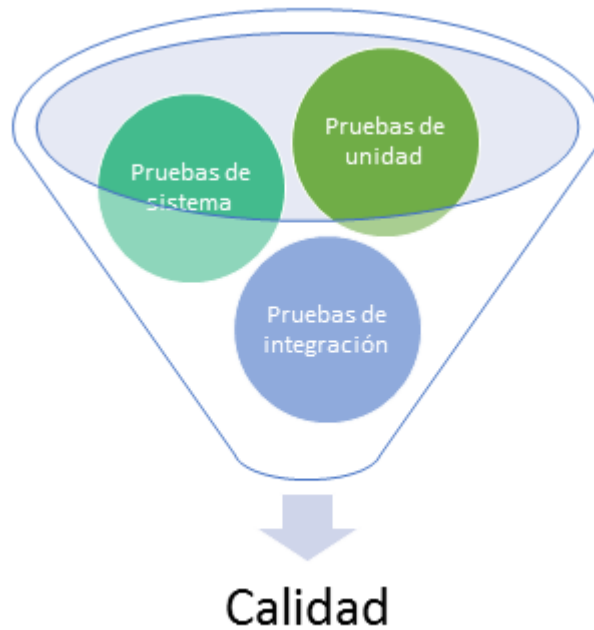


Figura 15: Utilizar las diferentes pruebas nos permitir realizar una evaluación integral para conseguir un producto de calidad.

Fuente: (JCAMILORADA, 2014)

3.4.-Enfoques de pruebas de sistema: caja negra, caja blanca o testing aleatorio

Como ya lo hemos visto en la semana anterior, ya probado todos los componentes del sistema y realizado la integración de ellos, se debe pasar a un nuevo nivel de pruebas en el que vamos a comprobar si el sistema cumple con los requisitos que se han especificado.

A. Caja Negra

Pruebas de caja negra: Técnica de pruebas en la que se verifica la funcionalidad, pero no se consideran, por ejemplo, la estructura interna del código, aspectos de implementación o elementos internos del software. En las pruebas de caja negra, el enfoque está centrado en las entradas y salidas del sistema, no es necesario conocer la estructura interna del sistema. Para la definición de las entradas y las salidas, es necesario basarse solamente en los requerimientos del software y las especificaciones funcionales.



Ejemplo 1: Campo de texto que solo acepta caracteres alfabéticos

Descripción del caso: Se tiene un campo de texto que solo acepta caracteres alfabéticos. La longitud del valor ingresado debe estar entre 6 y 10 caracteres.

Técnica de pruebas de caja negra: Partición de equivalencias¹.

Usando partición de equivalencias, se pueden establecer tres particiones, longitudes entre 0 y 5 caracteres (partición inválida), longitudes entre 6 y 10 caracteres (partición válida), y longitudes mayores a 10 caracteres (partición inválida). Además, el ingreso de caracteres no alfabéticos (por ejemplo, un número), se considera también dato inválido.

Caso 1.1: Datos de entrada: cadena de 5 caracteres. Resultado esperado (Salida): La aplicación no permite el ingreso del dato y muestra un mensaje de error.

Caso 1.2: Datos de entrada: cadena de 7 caracteres, incluyendo uno o más caracteres no alfabéticos. Resultado esperado (Salida): La aplicación no permite el ingreso del dato y muestra un mensaje de error.

Caso 1.3: Datos de entrada: cadena de 7 caracteres, solo de caracteres alfabéticos. Resultado esperado (Salida): La aplicación permite el ingreso del dato.

Caso 1.4: Datos de entrada: cadena de 11 caracteres. Resultado esperado (Salida): La aplicación no permite el ingreso del dato y muestra un mensaje de error. Fuente: (PMOinformatica.com, 2017)

Ejemplo 2: Ingreso de un campo fecha

Descripción de la situación: Se tiene una aplicación en la cual se registra una transacción administrativa o de contabilidad, que posee un campo fecha. Según la especificación funcional, el campo fecha solo acepta fecha iguales o anteriores al día actual. Es decir, el ingreso de fechas en el futuro no está permitido.

Técnica de pruebas de caja negra: Partición de equivalencias y análisis de valores borde².

¹ Este método intenta dividir el dominio de entrada de un programa en un número finito de clases de equivalencia. Una prueba realizada con un valor representativo de cada clase es equivalente a una prueba realizada con cualquier otro valor de dicha clase. Si el caso de prueba correspondiente a una clase de equivalencia detecta un error, el resto de los casos de prueba de dicha clase de equivalencia deben detectar el mismo error. Y viceversa. Fuente: (GAVIRIA, s.f.)

² Parte del principio que el comportamiento al borde de una partición de datos tiene mayores probabilidades de presentar errores (bugs). Los valores máximos y mínimos de una partición son sus valores borde. Aplican tanto para datos inválidos como válidos. Al incluirlas en el diseño de casos de prueba, se define una prueba



Caso 2.1: Datos de entrada: Fecha de hoy (Valor borde). Resultado esperado (Salida): Se permite el ingreso de la transacción (mensaje de éxito).

Caso 2.2: Datos de entrada: Fecha de hoy más un día (Fecha de mañana). Resultado esperado (Salida): No se permite el ingreso de la transacción y se muestra un mensaje de error.

Caso 2.3: Datos de entrada: Fecha del día de ayer. Resultado esperado (Salida): Se permite el ingreso de la transacción (mensaje de éxito). Fuente: (PMOinformatica.com, 2017)

B. Caja Blanca

Pruebas de caja blanca: Se basan en una estructura identificada del software o del sistema, según unos niveles específicos.

Nivel de componente: estructura de un componente software. Ejemplos: sentencias, decisiones, caminos distintos.

Nivel de integración: la estructura se basa en un árbol de llamadas, diagrama en el que un módulo llama a los otros módulos.

Nivel de sistema: la estructura puede ser por menús, ejemplo: proceso de negocio, páginas web.

Basadas en el flujo de control: se cubren todas las sentencias o bloques de sentencias en un programa, o combinaciones especificadas de ellas.

La adecuación de tales pruebas se mide en porcentajes; por ejemplo, se dice haber alcanzado una cobertura de sentencia del 100% cuando las sentencias han sido ejecutadas por lo menos una vez por las pruebas". Fuente: (Mera-Paz, Análisis del proceso de pruebas, 2016)

Existen tres pautas fundamentales para poder realizar con éxito una prueba de caja blanca.

Pruebas de cubrimiento: Se trata básicamente de ejecutar al menos una vez cada sentencia. Pero para cumplir con las pruebas de cubrimiento se necesitan varios casos de prueba:

- Determinar posibles «camino» independientes.
- Cada condición debe cumplirse en un caso y en otro no.
- Y puede ser imposible cubrir el 100%
- Código que nunca se ejecuta: condiciones imposibles

por cada valor borde. La capacidad de identificar defectos de esta técnica es alta, se pueden revisar las especificaciones funcionales para identificar datos interesantes. Fuente: (GAVIRIA, s.f.)



Pruebas de condiciones: Cumplir o no cada parte de cada condición. Se necesitan varios casos de prueba:

- Determinar expresiones simples en las condiciones
- Una por cada operando lógico o comparación
- Cada expresión simple debe cumplirse en un caso y en otro no, siendo decisiva en el resultado.

Pruebas de bucles: Se trata de verificar que los ciclos o bucles, tanto como «while» (repetir mientras) como el bucle «for» (repetir para), no sean infinitos o que funcionen correctamente:

- Conseguir números de repeticiones especiales
- Bucles simples:
 - Repetir cero, una y dos veces
 - Repetir un número medio (típico) de veces
 - Repetir el máximo-1, máximo y máximo +1
- Bucles anidados
 - Repetir un número medio (típico) los bucles internos, el mínimo los externos, y variar las repeticiones del bucle intermedio ensayado.
 - Ensayarlo con cada nivel de anidamiento.

Ahora bien, una vez comprendidos estos conceptos, vamos a ver un ejemplo básico de este tipo de prueba:

```
1 n = int(input("Ingresar N: "))
2 for i in range(n):
3     a = int(input("Ingresar A: "))
4     b = int(input("Ingresar B: "))
5     c = int(input("Ingresar C: "))
6     if a > 1 and b > 2 and c > 3:
7         s = (a + b) * (b + c) * (a + c)
8         if s > 0:
9             s = s / (a * b * c)
10        else:
11            s += 1
12
```

Figura 16: Ejemplo código Python
Fuente: (KIMBO, 2020)

Prueba de cubrimiento y condiciones:

Si se cumple que $A > 1$ y $B > 2$ y $C > 3$:

- A. Se puede ejecutar $S = (A+B) * (B+C) * (A+C)$
- B. Pero nunca va a cumplir $S = S+1$, ya que nunca S será menor a 0, ya que la primera condición pide que sean números mayores a 0.

C. Las pruebas de cubrimiento funcionarían únicamente en:

✓ $S = (A+B) * (B+C) * (A+C)$

✓ $S = S/(ABC)$

✗ $S = S+1$ (que es lo mismo que $S += 1$)

Si no se cumple que $A > 1$ y $B > 2$ y $C > 3$:

D. Nunca se ejecutaría las operaciones dentro de la condición (directamente no haría nada).

✗ $S = (A+B) * (B+C) * (A+C)$

✗ $S = S/ABC$

✗ $S = S+1$

Prueba de bucle:

Si $N > 0$:

E. Puede ejecutar las instrucciones que le siguen

Si $N \leq 0$:

F. Nunca se ejecutarían las instrucciones que le siguen

Fuente: (KIMBO, 2020)

C. Testing aleatorio

En inglés “random testing” o conocidas también como “monkey testing”

Es la técnica de diseño de pruebas de software de caja negra en la que se seleccionan casos de prueba, con la finalidad de simular un perfil operativo, en la que los sistemas son probados creando entradas independientes y aleatorias. Los resultados de la salida son comparados con los requerimientos del sistema para verificar si el resultado de la prueba es correcto o incorrecto.



Figura 17: Monkey Testing

Fuente: (Yunta, 2019)



Algunas de las fortalezas y debilidades de las pruebas aleatorias son:

Fortalezas de las pruebas aleatorias:

- Es económico de usar
- No tiene ningún sesgo
- Los errores se encuentran muy fácil y rápidamente.
- Si el software se usa correctamente, encontrará los errores.

Algunas debilidades de esta prueba son:

- Es capaz de encontrar solo errores básicos.
- Es preciso cuando las especificaciones son imprecisas.
- Esta técnica se compara mal con otras técnicas para encontrar errores.
- Esta técnica creará un problema para la integración continua si se seleccionan al azar diferentes entradas en cada prueba.
- Algunos piensan que la prueba de caja blanca es mejor que esta técnica de prueba aleatoria.

Características de las pruebas aleatorias:

- Se realiza cuando los defectos en una aplicación de software no se identifican en los intervalos regulares.
- La entrada aleatoria se utiliza para probar el rendimiento del sistema y su confiabilidad.
- Ahorra tiempo y esfuerzo que las pruebas reales.
- No se utilizan otros métodos de prueba.

Fuente: (Sr, 2019)

Pasos de prueba aleatorios:

- Las entradas aleatorias se identifican para evaluarlas con el sistema.
- Las entradas de prueba se seleccionan independientemente del dominio de prueba.
- Las pruebas se ejecutan utilizando esas entradas aleatorias.
- Registre los resultados y compárelos con los resultados esperados.
- Reproducir / replicar el problema y plantear defectos, corregir y volver a probar.

Fuente: (TutorialsPoint, 2021)

Ejemplo de prueba aleatoria:

Considerar en esta prueba números enteros aleatoriamente con la finalidad de probar una función del sistema que entrega sus resultados en base a esos números enteros, comparando los resultados obtenidos con los

resultados esperados, la diferencia es que en este caso las entradas son aleatorias por lo que ya no se cuenta con un conjunto específico o limitado para escoger, dispone de opciones infinitas que puede utilizar como entradas al sistema.

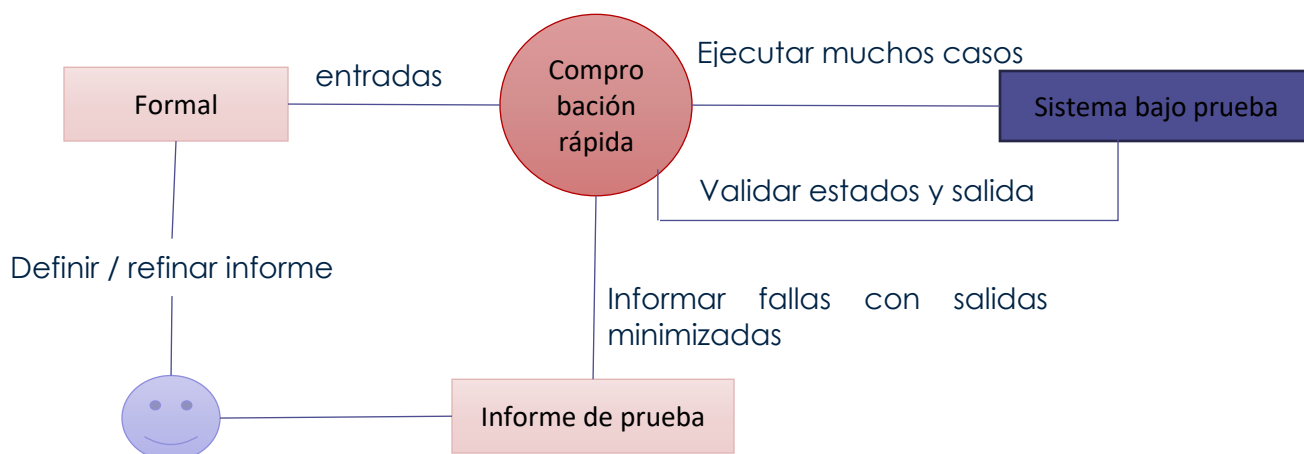


Figura 18: Pruebas aleatorias
Fuente: Elaboración propia

La mayoría de las observaciones negativas relacionadas con las pruebas aleatorias dicen relación con la incorrecta aplicación de la técnica, lo fundamental y punto clave en ese sentido es la generación aleatoria de las entradas que forman parte del dominio del sistema que está siendo probado.

Las pruebas aleatorias son ideales para poner a prueba, por ejemplo, comportamientos de falla, validar datos, cumplimiento de especificación de requisitos, concurrencia³ en el sistema.

4. Pruebas de sistema en diferentes niveles, considerando proyectos de desarrollo de software.

4.1.- Niveles de pruebas: unitarias, integración, sistema y aceptación

Podemos definir el concepto de niveles de pruebas como el conjunto de pruebas que se le aplican al software en las diversas fases del proceso de desarrollo, van desde la planificación de requisitos hasta las pruebas anteriores a la puesta en producción.

El procedimiento más utilizado para el desarrollo de pruebas de software es conocido como modelo en V y su característica esencial es el establecer tipos de prueba por cada fase del desarrollo.

³ La concurrencia en software implica la existencia de diversos flujos de control en un mismo programa colaborando para resolver un problema. Fuente: (valencia, s.f.)

Niveles de prueba

- Los niveles de pruebas definen los tipos de pruebas que se deben realizar y el orden en que se deben ejecutar

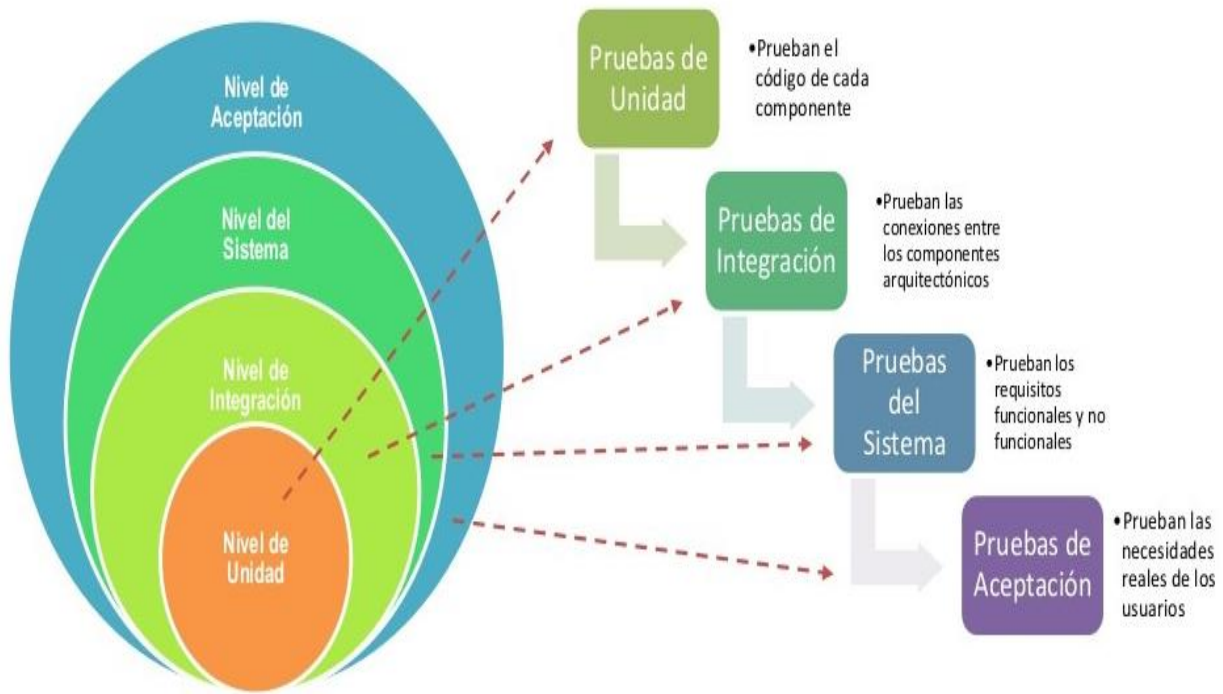


Figura 20: Niveles de pruebas
Fuente: (Valencia, 2016)

Prueba de unidad: Corresponden a la primera etapa de las pruebas dinámicas, se llevan a cabo independientemente en cada módulo del sistema. Tienen como finalidad demostrar que cada módulo está codificado de manera correcta.

Prueba de integración: Su objetivo es detectar errores generados por el ensamble de programas o componentes que han sido probados individualmente, se busca asegurar que la comunicación, relaciones y datos



compartidos se realicen debidamente. Se diseñan para evidenciar errores o completitud en las especificaciones de las interfaces.



Figura 21: Ejemplo pruebas unitarias y de integración

En la Figura 21 se aprecia claramente como la combinación de distintos componentes puede afectar a la funcionalidad individual de cada uno. En este ejemplo todas las pruebas unitarias tienen resultados correctos, los cajones abren y cierran correctamente, pero la falta de pruebas de integración hace que no se cumpla el sencillo requisito principal, puesto que una vez ensamblado el bloque de cajones, ninguno de ellos puede ser abierto como se esperaba. Fuente: (Hernandez, Medium, 2019)

Prueba de sistema: Su objetivo radica en verificar la integración adecuada de todos los elementos del sistema y que se ejecutan las operaciones correctas funcionando como un todo. Es parecido a la prueba de integración, pero con un alcance mucho más extenso.

Prueba de aceptación: Las realizan fundamentalmente los usuarios apoyados por el equipo del proyecto. Su objetivo es corroborar que el sistema está terminado, que cubre debidamente los requisitos de la organización y que es aceptado por los usuarios finales.

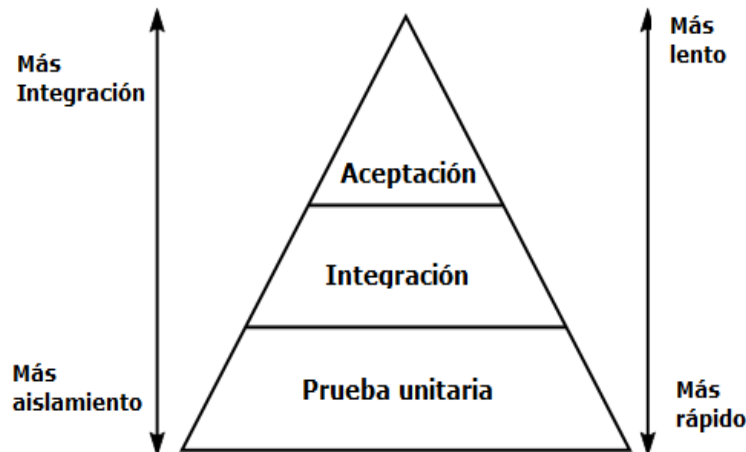


Figura 22: Ámbito de las pruebas
Fuente: Elaboración propia

La Figura 22 es una muy buena representación de la estructura y el ámbito de los distintos tipos de pruebas que se pueden hallar en la actualidad.

¿Cómo puedo estar seguro de que lo he hecho correctamente?

Aplicando las mismas prácticas SQA⁴ que se aplican en todos los procesos de ingeniería del software -las revisiones técnicas formales valoran los modelos de análisis y diseño-; las revisiones especializadas tienen en consideración la usabilidad y la comprobación se aplica para descubrir errores en el contenido, funcionalidad y compatibilidad. Fuente: (Roger S. Pressman, 2010)

⁴ Software Quality Assurance - Garantía de calidad del software SQA



Conclusiones

Hemos visto en esta semana las pruebas desde otro punto de vista, profundizamos en sus definiciones, pero conocimos un aspecto muy importante como es la aplicación de ellas, entendiendo que cada etapa del ciclo de vida tiene asociadas ciertas pruebas, que tienen características y objetivos diferentes.

Debemos tener presente en cada uno de nuestros desarrollos, el cumplimiento de los requerimientos que han sido definidos con el cliente al comienzo del proceso y un proceso que incorpore conceptos de calidad como son el diseño y aplicación de las pruebas según el objetivo propuesto.

Referencias bibliográficas

Roger S. Pressman, P. (2010). INGENIERÍA DEL SOFTWARE. UN ENFOQUE PRÁCTICO. McGraw-Hill Interamericana Editores, S.A. de c.v

Federico Toledo Rodríguez, INTRODUCCIÓN A LAS PRUEBAS DE SISTEMAS DE INFORMACIÓN. Abstracta, Montevideo, Uruguay, 2014

J. A. Mera-Paz, "ANÁLISIS DEL PROCESO DE PRUEBAS DE CALIDAD DE SOFTWARE", Ingeniería Solidaria, vol. 12, no. 20, pp. 163-176, oct. 2016.
doi: <https://repository.ucc.edu.co/handle/20.500.12494/962>