EL PROCESO DE PRUEBAS

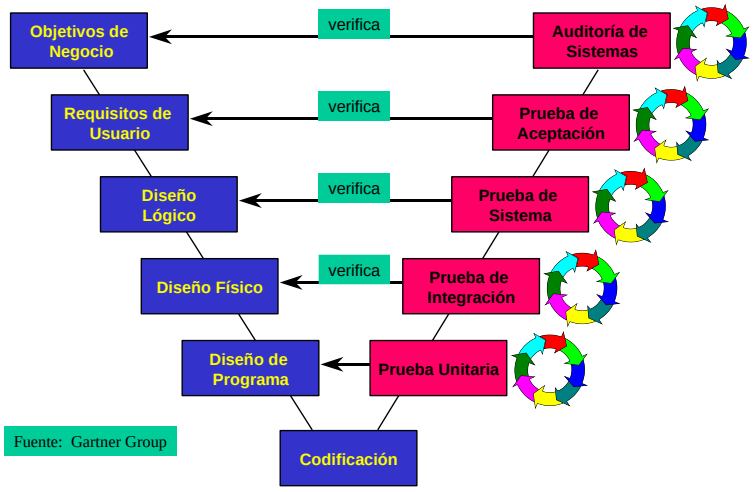
Enfoque

- El enfoque inicial de los equipos de TI hasta la década de 1980, cuando se enfrentaban a una calidad insuficiente, era intentar encontrar todos los fallos mediante pruebas y luego solucionarlos. Para concienciar a la gente sobre esta falacia, ayudó mucho el término "fase de fijación" [Weinberg 2008]. Este término tenía como objetivo concienciar a la gente de que las pruebas no pueden incluir calidad al final, ni un equipo puede solucionar ningún problema al final.
- En las décadas de 1980 y 1990 se descubrió que es imposible detectar todos los fallos en un sistema complejo: Nunca habrá un sistema perfecto si se prueba al final y se intenta encontrar y solucionar todos los fallos. Por lo tanto, es necesario un conjunto equilibrado de medidas de calidad a lo largo de todo el ciclo de vida para garantizar la calidad.

El proceso

- Excepto para programas pequeños, los sistemas no deberían probarse como un único elemento indivisible.
- Los sistemas grandes se construyen a partir de subsistemas que se construyen a partir de módulos (paquetes, ensamblados, componentes, …), compuestos de clases y métodos o funciones y procedimientos.
- El proceso de prueba debería trabajar por etapas, llevando a cabo la prueba de forma incremental a la vez que se realiza la implementación del sistema, siguiendo el modelo en V.

Proceso de pruebas: Ciclo en V

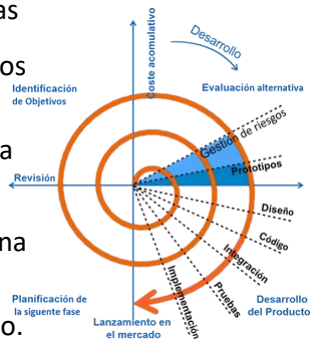


Ciclo en V

- El modelo en V establece una simetría entre las fases de desarrollo y las pruebas.
- Las principales consideraciones se basan en la inclusión de las actividades de planificación y ejecución de pruebas como parte del proyecto de desarrollo.
- Inicialmente, la ingeniería del sistema define el papel del software y conduce al análisis de los requisitos del software, donde se establece el campo de información, la función, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software. Al movernos hacia abajo, llegamos al diseño y, por último, a la codificación, el vértice de la V.
- Para desarrollar las pruebas seguimos el camino ascendente por la otra rama de la V. Partiendo de los elementos más básicos, probamos que funcionan como deben (lo que hacen, lo hacen bien). Los combinamos y probamos que siguen funcionando como deben. Para terminar probamos que hacen lo que deben (que hacen todo lo que tienen que hacer).

Desarrollo iterativo e incremental

- El desarrollo incremental implica establecer requisitos, diseñar, construir y probar un sistema en fragmentos, lo que significa que las prestaciones del software crecen de forma incremental. El tamaño de estos incrementos de prestaciones varía, ya que algunos métodos tienen piezas más grandes y otros más pequeñas. Los incrementos de prestaciones pueden ser tan pequeños como un simple cambio en una pantalla de la interfaz de usuario o una nueva opción en una consulta.
- El desarrollo iterativo se produce cuando se especifican, diseñan, construyen y prueban conjuntamente grupos de prestaciones en una serie de ciclos, a menudo, de una duración fija. Las iteraciones pueden implicar cambios en las prestaciones desarrolladas en iteraciones anteriores, junto con cambios en el alcance del proyecto. Cada iteración proporciona software operativo, que es un subconjunto creciente del conjunto general de prestaciones hasta que se entrega el software final o se detiene el desarrollo.



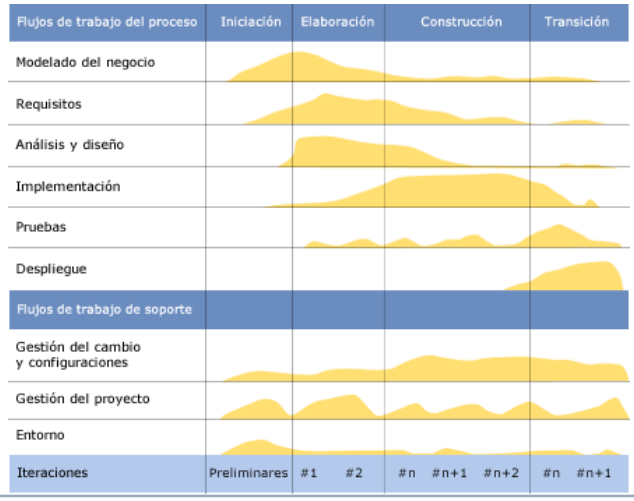
Desarrollo iterativo e incremental

- Rational Unified Process (RUP):
  - Cada iteración tiende a ser relativamente larga (por ejemplo, de dos a tres meses), y los incrementos de las prestaciones son proporcionalmente grandes, como por ejemplo dos o tres grupos de prestaciones relacionadas.
- Scrum:
  - Cada iteración tiende a ser relativamente corta (por ejemplo, horas, días o unas pocas semanas), y los incrementos de las prestaciones son proporcionalmente pequeños, como unas pocas mejoras y/o dos o tres prestaciones nuevas.
- Kanban:
  - Implementado con o sin iteraciones de longitud fija, que puede ofrecer una sola mejora o prestación una vez finalizada, o puede agrupar prestaciones para liberarlas de una sola vez.
- Espiral (o prototipado):
  - Implica la creación de incrementos experimentales, algunos de los cuales pueden ser reelaborados en profundidad o incluso abandonados en trabajos de desarrollo posteriores.

RUP

- El Proceso Unificado Rational o RUP es un proceso de desarrollo de software dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental. Complementa al Lenguaje Unificado de Modelado (UML) y constituye la metodología estándar más utilizada para el análisis, diseño, implementación y documentación de sistemas orientados a objetos.
- RUP divide el proceso en cuatro fases: Inicio, Elaboración, Construcción y Transición.
  - Durante la fase de inicio, las iteraciones hacen mayor énfasis en actividades de modelado del negocio y de requisitos.
  - En la fase de elaboración, las iteraciones se orientan al desarrollo de la baseline de la arquitectura, abarcan más los flujos de trabajo de requisitos, modelo de negocios (refinamiento), análisis, diseño y una parte de implementación orientado a la baseline de la arquitectura.
  - En la fase de construcción, se lleva a cabo la construcción del producto por medio de una serie de iteraciones. Para cada iteración se seleccionan algunos Casos de Uso, se refinan su análisis y diseño y se procede a su implementación y pruebas. Se realiza una pequeña cascada para cada ciclo. Se realizan iteraciones hasta que se termine la implementación de la nueva versión del producto.
  - En la fase de transición (pruebas finales de aceptación, puesta en producción, estabilización) se pretende garantizar que se tiene un producto preparado para su entrega a los usuarios.
- Las primeras iteraciones (en las fases de Inicio y Elaboración) se enfocan hacia la comprensión del problema y la tecnología, la delimitación del ámbito del proyecto, la eliminación de los riesgos críticos y al establecimiento de una baseline (línea base) de la arquitectura.
- En cada fase participan todas las disciplinas: captura de requisitos, modelado de negocio, análisis, diseño, implementación y pruebas, pero dependiendo de la fase el esfuerzo dedicado a una disciplina varía.

RUP



Pruebas

- Las pruebas consisten en actividades de verificación, validación y exploración que brindan información sobre la calidad y los riesgos relacionados, para establecer el nivel de confianza de que el objeto de prueba podrá entregar el valor comercial buscado en la base de la prueba.
- La verificación es la confirmación mediante examen y mediante la provisión de evidencia objetiva de que se han cumplido los requisitos especificados. Responde a la pregunta: ¿Estamos construyendo correctamente el sistema de TI?
- La validación es la confirmación mediante examen y mediante la provisión de evidencia objetiva de que se han cumplido las demandas para un uso específico previsto. Responde a la pregunta: ¿Estamos construyendo el sistema de TI adecuado?
- La exploración es la actividad de investigar y establecer la calidad y los riesgos del uso de un sistema de TI mediante examen, indagación y análisis. Responde a la pregunta: ¿Cómo se podría (mal)utilizar el sistema de TI?
- Las pruebas deben proporcionar diferentes niveles de información:
  - Detallado: para conocer la calidad de los objetos nuevos y modificados que entregaron e investigar anomalías en el sistema de TI y tomar medidas correctivas cuando sea necesario.
  - Intermedio: para realizar un seguimiento del estado y el progreso.
  - General: para respaldar las decisiones de seguir o no.

Criterios de evaluación

- Las pruebas consisten en evaluar la calidad basándose en criterios. En las pruebas utilizamos criterios para determinar si el objeto de prueba cumple con las expectativas sobre calidad y riesgos.
- Los **criterios de entrada** son los criterios que un objeto (por ejemplo, un documento base de prueba o un objeto de prueba) debe satisfacer para estar listo para usarse en una actividad específica.
- Los **criterios de salida** son los criterios que un objeto (por ejemplo, un documento base de prueba o un objeto de prueba) debe satisfacer para estar listo al final de una actividad o etapa específica del proyecto.
- Los **criterios de aceptación** son los criterios que debe cumplir un objeto de prueba para ser aceptado por un usuario, cliente u otra parte interesada.
- Los **criterios de finalización** son los criterios que un equipo debe satisfacer para haber completado una (grupo de) actividad(es).

Variedad de pruebas

- Las pruebas cubren una gran cantidad de escenarios por lo que tienen una gran variedad de clasificaciones en función de la metodología, enfoque, objetivos, criterios, …
  - Tipo: Estáticas y Dinámicas
  - Información: Pruebas de defectos y Pruebas estadísticas
  - Nivel: Unitarias, Integración, Sistema, Aceptación
  - Modo: Pruebas manuales o automatizadas
  - Características de calidad: Pruebas funcionales y no funcionales (rendimiento, estructurales, de mantenimiento, seguridad, …)
  - Enfoque: Basadas en la especificación, en la estructura o en la experiencia.
  - Estrategias: Pruebas de humo, progresión, regresión, aprendizaje, exploratorias, …

Pruebas estáticas

- Las pruebas estáticas, también conocidas como revisión, son pruebas que examinan productos (como especificaciones de requisitos, manuales o código fuente) sin que se ejecuten programas.
- Esto se puede hacer sin que el objeto de prueba se esté ejecutando, por ejemplo, revisando requisitos o documentos de diseño, código fuente, manuales de usuario, planes de proyecto y prueba y casos de prueba.
- En los modelos de entrega de TI de alto rendimiento, como Scrum y DevOps, una parte importante de las actividades de prueba estáticas se realiza durante el refinamiento de las historias de los usuarios, por lo que no se implementa como una actividad separada, pero aún requiere un enfoque específico. Además, se realiza un análisis estático del código para cumplir con los estándares de codificación.
- Las pruebas estáticas estarán respaldadas por herramientas tanto como sea posible. De hecho, hoy en día existen herramientas que no sólo hacen análisis estático sino que también son capaces de refactorizar el código basándose en patrones de refactorización estándar. Aún así, en general, las pruebas estáticas son una combinación de inteligencia humana y poder de herramientas.

Pruebas dinámicas

- Las pruebas dinámicas son pruebas mediante la ejecución del objeto de prueba, es decir, la ejecución de una aplicación.
- Cuando ejecutamos una prueba, en otras palabras, ejecutamos el objeto de la prueba (que puede variar desde un único módulo de programa hasta un proceso de negocio completo de extremo a extremo en múltiples organizaciones), eso es lo que llamamos prueba dinámica.
- Las pruebas dinámicas se pueden realizar manualmente, pero para muchas variedades de pruebas, se recomienda encarecidamente el soporte de herramientas de automatización de pruebas.

Prueba estadística

- La **prueba estadística** se puede utilizar para probar el rendimiento del programa y su confiabilidad.
- Las pruebas se diseñan para reflejar la frecuencia de entradas reales de usuario.
- Después de ejecutar las pruebas, se puede hacer una estimación de la confiabilidad operacional del sistema.
- El rendimiento del programa se puede juzgar midiendo la ejecución de las pruebas estadísticas.

Prueba de defectos

- La **prueba de defectos** intenta incluir áreas donde el programa no está de acuerdo con sus especificaciones.
- Las pruebas se diseñan para revelar la presencia de defectos en el sistema que se manifiestan en forma de fallos.
- Los fallos demuestran la existencia de defectos que han sido ocasionados por errores.
- Cuando se han encontrado defectos en un programa, éstos deben ser localizados y eliminados. A este proceso se le denomina **depuración**.

Niveles de pruebas

- **Pruebas Unitarias o de Componentes**: verifican la funcionalidad y estructura de cada componente individualmente, una vez que ha sido codificado.
- **Pruebas de Integración**: verifican el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente, con el fin de comprobar que interactúan correctamente a través de sus interfaces, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.
- **Pruebas de Regresión**: verifican que los cambios sobre un componente de un sistema de información no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- **Pruebas del Sistema**: ejercitan profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.
- **Pruebas de Aceptación**: validan que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema, que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.

Pruebas Unitarias

- Las pruebas unitarias tienen como objetivo verificar la funcionalidad y estructura de cada componente individualmente, una vez que ha sido codificado.
- Con las pruebas unitarias verificas el diseño de los programas, vigilando que no se producen errores y que el resultado de los programas es el esperado.
- Estas pruebas las efectúa normalmente la misma persona que codifica o modifica el componente y que, también normalmente, genera un juego de ensayo para probar y depurar las condiciones de prueba.
- Las pruebas unitarias constituyen la prueba inicial de un sistema y las demás pruebas deben apoyarse sobre ellas.

Pruebas Unitarias

- Existen dos **enfoques** principales para el diseño de casos de prueba:
  - **Enfoque estructural o de caja blanca**. Se verifica la estructura interna del componente con independencia de la funcionalidad establecida para el mismo.

    Por tanto, no se comprueba la corrección de los resultados, sólo si éstos se producen. Ejemplos de este tipo de pruebas pueden ser ejecutar todas las instrucciones del programa, localizar código no usado, comprobar los caminos lógicos del programa, etc.
  - **Enfoque funcional o de caja negra**. Se comprueba el correcto funcionamiento de los componentes del sistema de información, analizando las entradas y salidas y verificando que el resultado es el esperado. Se consideran exclusivamente las entradas y salidas del sistema sin preocuparse por la estructura interna del mismo.
- El enfoque que suele adoptarse para una prueba unitaria está claramente orientado al diseño de casos de caja blanca, aunque se complemente con caja negra.

Pruebas Unitarias

- Los **pasos necesarios** para llevar a cabo las pruebas unitarias son los siguientes:
  - **Ejecutar todos los casos de prueba** asociados a cada verificación establecida en el plan de pruebas, registrando su resultado. Los casos de prueba deben contemplar tanto las condiciones válidas y esperadas como las inválidas e inesperadas.
  - **Corregir los errores o defectos encontrados y repetir las pruebas que los detectaron**. Si se considera necesario, debido a su implicación o importancia, se repetirán otros casos de prueba ya realizados con anterioridad.

Pruebas Unitarias

- La prueba unitaria se da por finalizada cuando se hayan realizado todas las verificaciones establecidas y no se encuentre ningún defecto, o bien se determine su suspensión.
- Al finalizar las pruebas, obtienes las **métricas de calidad del componente** y las contrastas con las existentes antes de la modificación:
  - Número ciclomático.
  - Cobertura de código.
  - Porcentaje de comentarios.
  - Defectos hallados contra especificaciones o estándares.
  - Rendimientos.

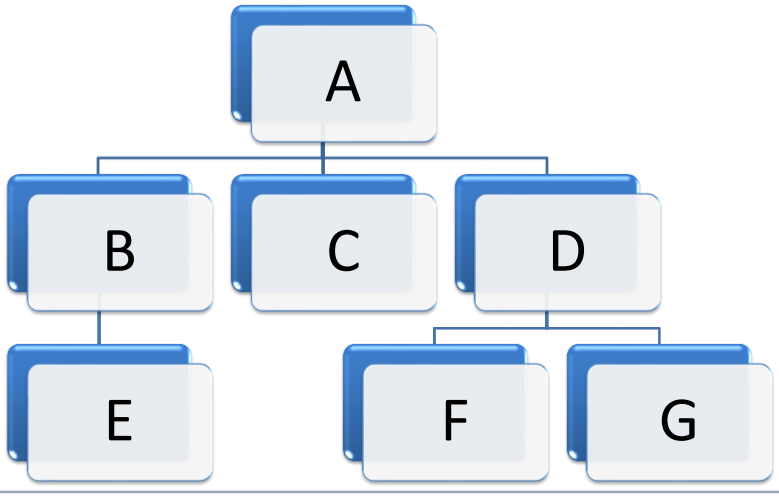
Pruebas de Integración

- Las pruebas de integración te permiten verificar el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente, con el fin de comprobar que interactúan correctamente a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.
- Se trata de probar los caminos lógicos del flujo de los datos y mensajes a través de un conjunto de componentes relacionados que definen una cierta funcionalidad.
- En las pruebas de integración examinas las interfaces entre grupos de componentes o subsistemas para asegurar que son llamados cuando es necesario y que los datos o mensajes que se transmiten son los requeridos.
- Debido a que en las pruebas unitarias es necesario crear módulos auxiliares que simulen las acciones de los componentes invocados por el que se está probando, y a que se han de crear componentes "conductores" para establecer las precondiciones necesarias, llamar al componente objeto de la prueba y examinar los resultados de la prueba, a menudo se combinan los tipos de prueba unitarias y de integración.

Pruebas de Integración

- Los **tipos fundamentales de integración** son los siguientes:
  - **Integración incremental:** combinas el siguiente componente que debes probar con el conjunto de componentes que ya están probados y vas incrementando progresivamente el número de componentes a probar.
  - **Integración no incremental:** pruebas cada componente por separado y, luego, los integras todos de una vez realizando las pruebas pertinentes. Este tipo de integración se denomina también Big-Bang (gran explosión).
- Con el tipo de prueba incremental lo más probable es que los problemas te surjan al incorporar un nuevo componente o un grupo de componentes al previamente probado. Los problemas serán debidos a este último o a las interfaces entre éste y los otros componentes.

Pruebas de Integración



Estrategias de integración

- **De arriba a abajo (top-down):** el primer componente que se desarrolla y prueba es el primero de la jerarquía (A).
  - Los componentes de nivel más bajo se sustituyen por componentes auxiliares o dobles de pueba, para simular a los componentes invocados. En este caso no son necesarios componentes conductores.
  - Una de las ventajas de aplicar esta estrategia es que las interfaces entre los distintos componentes se prueban en una fase temprana y con frecuencia.
- **De abajo a arriba (bottom-up):** en este caso se crean primero los componentes de más bajo nivel (E, F, G) y se crean componentes conductores para simular a los componentes que los llaman.
  - A continuación se desarrollan los componentes de más alto nivel (B, C, D) y se prueban. Por último dichos componentes se combinan con el que los llama (A). Los componentes auxiliares son necesarios en raras ocasiones.
  - Este tipo de enfoque permite un desarrollo más en paralelo que el enfoque de arriba a abajo, pero presenta mayores dificultades a la hora de planificar y de gestionar.
- **Estrategias combinadas:** A menudo es útil aplicar las estrategias anteriores conjuntamente. De este modo, se desarrollan partes del sistema con un enfoque "top-down", mientras que los componentes más críticos en el nivel más bajo se desarrollan siguiendo un enfoque "bottom-up".
  - En este caso es necesaria una planificación cuidadosa y coordinada de modo que los componentes individuales se "encuentren" en el centro.

Pruebas del Sistema

- Las pruebas del sistema tienen como objetivo ejercitar profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.
- Son pruebas de integración del sistema de información completo, que permiten probar el sistema en su conjunto y con otros sistemas con los que se relaciona para verificar que las especificaciones funcionales y técnicas se cumplen. Dan una visión muy similar a su comportamiento en el entorno de producción.
- Una vez que se han probado los componentes individuales y se han integrado, se prueba el sistema de forma global. En esta etapa pueden distinguirse diferentes tipos de pruebas, cada uno con un objetivo claramente diferenciado.

Pruebas del Sistema

- **Pruebas funcionales**: dirigidas a asegurar que el sistema de información realiza correctamente todas las funciones que se han detallado en las especificaciones dadas por el usuario del sistema.
- **Pruebas de humo**: son un conjunto de pruebas aplicadas a cada nueva versión, su objetivo es validar que las funcionalidades básicas de la versión se cumplen según lo especificado. Impiden la ejecución el plan de pruebas si detectan grandes inestabilidades o si elementos clave faltan o son defectuosos.
- **Pruebas de comunicaciones**: determinan que las interfaces entre los componentes del sistema funcionan adecuadamente, tanto a través de dispositivos remotos, como locales. Asimismo, se han de probar las interfaces hombre-máquina.
- **Pruebas de rendimiento**: consisten en determinar que los tiempos de respuesta están dentro de los intervalos establecidos en las especificaciones del sistema.
- **Pruebas de volumen**: consisten en examinar el funcionamiento del sistema cuando está trabajando con grandes volúmenes de datos, simulando las cargas de trabajo esperadas.
- **Pruebas de sobrecarga o estrés**: consisten en comprobar el funcionamiento del sistema en el umbral límite de los recursos, sometiéndole a cargas masivas. El objetivo es establecer los puntos extremos en los cuales el sistema empieza a operar por debajo de los requisitos establecidos.

Pruebas del Sistema

- **Pruebas de disponibilidad de datos**: consisten en demostrar que el sistema puede recuperarse ante fallos, tanto de equipo físico como lógico, sin comprometer la integridad de los datos.
- **Pruebas de seguridad**: consisten en verificar los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos.
- **Pruebas de usabilidad**: consisten en comprobar la adaptabilidad del sistema a las necesidades de los usuarios, tanto para asegurar que se acomoda a su modo habitual de trabajo, como para determinar las facilidades que aporta al introducir datos en el sistema y obtener los resultados.
- **Pruebas extremo a extremo (e2e)**: consisten en interactuar con la aplicación como un usuario regular lo haría, cliente-servidor, y evaluando las respuestas para el comportamiento esperado.
- **Pruebas de configuración**: consisten en comprobar todos y cada uno de los dispositivos, en sus propiedades mínimo y máximo posibles.
- **Pruebas de operación**: consisten en comprobar la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y re-arranque del sistema, etc.
- **Pruebas de entorno**: consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno.

Pruebas de Aceptación

- El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema, que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.
- Las pruebas de aceptación son preparadas por el usuario del sistema y el equipo de desarrollo, aunque la ejecución y aprobación final corresponde al usuario.
- Estas pruebas van dirigidas a comprobar que el sistema cumple los requisitos de funcionamiento esperado recogidos en el catálogo de requisitos y en los criterios de aceptación del sistema de información, y conseguir la aceptación final del sistema por parte del usuario.

Pruebas de Aceptación

- Previamente a la realización de las pruebas, el responsable de usuarios revisa los criterios de aceptación que se especificaron previamente en el plan de pruebas del sistema y dirige las pruebas de aceptación final.
- La validación del sistema se consigue mediante la realización de pruebas de caja negra que demuestran la conformidad con los requisitos y que se recogen en el plan de pruebas, el cual define las verificaciones a realizar y los casos de prueba.
- Dicho plan está diseñado para asegurar que se satisfacen todos los requisitos funcionales especificados por el usuario teniendo en cuenta, a su vez, los requisitos no funcionales relacionados con el rendimiento, seguridad de acceso al sistema, a los datos y procesos, así como a los distintos recursos del sistema.
- La formalidad de estas pruebas dependerá en mayor o menor medida de cada organización, y vendrá dada por la criticidad del sistema, el número de usuarios implicados en las mismas y el tiempo del que se disponga para llevarlas cabo, entre otros.

Pruebas alfa y beta

- Las pruebas alfa y beta son la alternativa a las pruebas de aceptación en el software comercial de distribución masiva (COTS). Las pruebas alfa y beta suelen ser utilizadas por los desarrolladores de COTS que desean obtener retroalimentación de los usuarios, clientes y/u operadores existentes o potenciales antes de que el producto de software sea puesto en el mercado. Uno de los objetivos de las pruebas alfa y beta es generar la confianza de que se pueden utilizar el sistema en condiciones normales y cotidianas, con la mínima dificultad, coste y riesgo.
- Las pruebas alfa son pruebas de software realizadas en las fases iniciales del proyecto cuando el sistema está en desarrollo y cuyo objetivo es asegurar que lo que estamos desarrollando es probablemente correcto y útil para el cliente. Si las pruebas alfa devuelven buenos comentarios positivos entonces podríamos seguir por esa vía. Si devolvieran resultados muy negativos tendríamos que replantear el problema para adaptarse mejor a los requerimientos.
- Las pruebas beta son las pruebas de software que se realizan cuando el sistema está completado y, en teoría, es correcto, antes de pasar a producción. Dado que no es viable realizar pruebas exhaustivas, siempre habrá fallos que no han sido descubiertos por los desarrolladores ni por el equipo de pruebas. Las pruebas beta son pruebas para localizar esos problemas no detectados y poder corregirlos antes de liberar la versión definitivas; la prueba debería ser realizada por usuarios finales.

Pruebas de Regresión

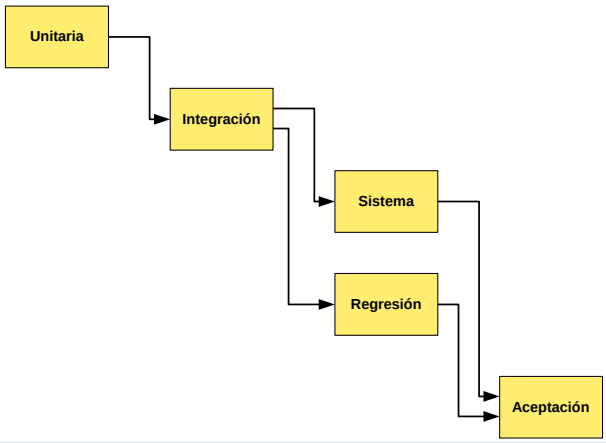
- El objetivo de las pruebas de regresión es eliminar el efecto onda, es decir, comprobar que los cambios sobre un componente de un sistema de información, no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error como para realizar una mejora.
- No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan (también conocidas como **pruebas de progresión o confirmación**), sino que también es necesario controlar que las modificaciones no produzcan efectos secundarios negativos sobre el mismo u otros componentes.
- Normalmente, este tipo de pruebas implica la repetición de las pruebas que ya se han realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

Pruebas de Regresión

- Las pruebas de regresión **pueden incluir**:
  - La repetición de los casos de pruebas que se han realizado anteriormente y están directamente relacionados con la parte del sistema modificada.
  - La revisión de los procedimientos manuales preparados antes del cambio, para asegurar que permanecen correctamente.
  - La obtención impresa del diccionario de datos de forma que se compruebe que los elementos de datos que han sufrido algún cambio son correctos.
- El **responsable** de realizar las pruebas de regresión será el equipo de desarrollo junto al técnico de mantenimiento, quién a su vez, será responsable de especificar el plan de pruebas de regresión y de evaluar los resultados de dichas pruebas.

Niveles de pruebas y orden de ejecución.

- De tal forma que la secuencia de pruebas es:



Smoke Testing y Sanity Testing

- Un smoke test (prueba de humo) es una prueba superficial realizada para asegurar de que una nueva distribución puede realizar la funcionalidad básica sin encontrar errores críticos. El objetivo principal de un smoke test, subconjunto de pruebas rápidas y automatizadas, es determinar si la compilación del software desplegado es suficientemente estable antes de realizar pruebas más exhaustivas y costosas.
- Un sanity test (prueba de cordura o sanidad) es una prueba superficial realizada para asegurar para comprobar si un cambio o nueva funcionalidad de una nueva distribución funciona correctamente. Un sanity test es un tipo de pruebas de regresión más enfocado y específico para validar los componentes mejorados en lugar del producto completo.
- Las pruebas de sanidad se ejecutan después de haber superado las pruebas de humo y antes de acometer las pruebas de regresión.
- Las pruebas de humo y sanidad son parte de las estrategias de pruebas y ahorran mucho tiempo y esfuerzo porque evitan que los equipos de control de calidad pierdan tiempo en pruebas más profundas antes de asegurarse de que las funciones básicas de la compilación del software funcionan como deberían.

Prueba exploratoria

- Incluso los esfuerzos de automatización de pruebas más diligentes no son perfectos. A veces se pierden ciertos casos extremos en sus pruebas automatizadas. A veces es casi imposible detectar un error en particular escribiendo una prueba unitaria. Ciertos problemas de calidad ni siquiera se hacen evidentes en las pruebas automatizadas (como en el diseño o la usabilidad).
- Las pruebas exploratorias es un enfoque de prueba manual que enfatiza la libertad y creatividad del probador para detectar problemas de calidad en un sistema en ejecución.
  - Simplemente tomate un tiempo en un horario regular, arremángate e intenta romper la aplicación.
  - Usa una mentalidad destructiva y encuentra formas de provocar problemas y errores en la aplicación.
  - Ten en cuenta los errores, los problemas de diseño, los tiempos de respuesta lentos, los mensajes de error faltantes o engañosos y, en general, todo lo que pueda molestarte como usuario de una aplicación.
  - Documenta todo lo que encuentre para más adelante.
- La buena noticia es que se puede automatizar la mayoría de los hallazgos con pruebas automatizadas.
- Escribir pruebas automatizadas para los errores que se detectan asegura que no habrá regresiones a ese error en el futuro. Además, ayuda a reducir la causa raíz de ese problema durante la corrección de errores.

Pruebas end-to-end (e2e)

- Las pruebas end-to-end, sistema y aceptación ejercitan los caminos de los flujos de trabajo de los usuarios, de principio a fin (de un extremo al otro), imitando las condiciones de los usuarios (GUI), buscando validar que se comportan como debe ser y no emergen errores al combinar los componentes involucrados. Surgieron para superar las limitaciones de otros tipos de procedimientos para pruebas.
- Verifican la correcta interacción e intercambio de datos los componentes tanto internos como externos. Verificar el rendimiento de los componentes de la aplicación permite una amplia cobertura y que se identifiquen de forma fácil defectos y errores ocultos. Los probadores pueden observar las respuestas de los usuarios a las funciones del software.
- Los desafíos que deben superar la pruebas e2e son: requieren comprender los objetivos de los usuarios y pueden ser difíciles y farragosas de diseñar (con record & replay se pueden simplificar), son muy costosas tanto en tiempo como en recursos, sin un ambiente integrado no son predecibles ni automatizables, son poco concluyentes por lo que requieren un análisis causa-error para aislar fallos.
- Pese a sus desafíos, las pruebas e2e son la base de las pruebas de aceptación porque utilizan la perspectiva del usuarios y ejercitan el sistema de una forma integral.

Automatización de pruebas

- Las pruebas exploratorias (manuales) son muy costosas y difícilmente repetibles, por lo que se impone una estrategia de automatización.
- Las pruebas funcionales de usuario final son caras de ejecutar que, por ejemplo, requieren abrir un navegador e interactuar con el. Además, normalmente requieren que una infraestructura considerable este disponible para estas ejecutarse de manera efectiva. Es una buena regla preguntarse siempre si lo que se quiere probar se puede hacer usando enfoques de prueba más livianos como las pruebas unitarias o con un enfoque de bajo nivel.
- Los analizadores de código son herramientas de fuerza bruta que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.

Aprender con pruebas unitarias

- La incorporación de código de tercero es complicado, hay que aprenderlo primero e integrarlo después. Hacer las dos cosa a la vez es el doble de complicado.
- Utilizar pruebas unitarias en el proceso de aprendizaje (pruebas de aprendizaje según Jim Newkirk) aporta importantes ventajas:
  - La inmediatez de las pruebas unitarias y sus entornos.
  - Realizar “pruebas de concepto” para comprobar si el comportamiento se corresponde con lo que hemos entendido, permitiéndonos clarificarlo.
  - Experimentar para encontrar los mejores escenarios de integración.
  - Permite saber si un fallo es nuestro, de la librería o del uso inadecuado de la librería.

Pruebas de aprendizaje

- Las pruebas de aprendizaje no suponen un coste adicional, es parte del coste de aprendizaje que, en todo caso, lo minoran.
- Es mas, las pruebas de aprendizaje son rentables. Ante la aparición de nuevas versiones del código ajeno, ejecutar la batería de pruebas de aprendizaje valida el impacto de la adopción de la nueva versión: detecta cambios relevantes, efectos negativos en las integraciones, ...
- Las pruebas de aprendizaje no sustituyen al conjunto de pruebas que respaldan los limites establecidos.

Cobertura

- La cobertura es la relación entre lo que se puede probar y lo que realmente se prueba con el equipo de prueba.
- La cobertura es un concepto útil que ayuda a los evaluadores con preguntas complejas, como:
  - Dado que es imposible probarlo todo y, por lo tanto, nos vemos obligados a probar solo un subconjunto de todas las posibilidades, ¿cuál es el mejor subconjunto?
  - ¿Cuál es la diferencia entre "pruebas elementales" y "pruebas exhaustivas"? ¿Qué significa concretamente para los casos de prueba que necesitamos para esto?
- La cobertura tiene mucho que ver con el deseo de encontrar la mayor cantidad de defectos posibles con el menor número de casos de prueba posibles.
- En lugar de simplemente probar "cualquier" subconjunto de opciones, nuestro objetivo es compilar un conjunto de casos de prueba que creen el mayor cambio posible para encontrar los defectos que existen.

Tipo e índice de cobertura

- El tipo de cobertura es la forma en que se expresa la cobertura de las situaciones de prueba deducibles de la base de la prueba. Indica qué tipo de posibilidades se distinguen y están involucradas en las situaciones de prueba.
- El índice de cobertura es el porcentaje de situaciones de prueba, según lo definido por el tipo de cobertura, que cubre la prueba. Indica cuántas de estas posibilidades se prueban y generalmente se expresa como porcentaje. Muestra qué parte de todas las alternativas realmente se han probado. El número (porcentaje) solo tiene significado cuando se asocia con el tipo de cobertura.
- No podemos hablar de la cobertura, sino de muchas formas o tipos de cobertura. Los diferentes tipos de cobertura tienden a complementarse en lugar de reemplazarse entre sí, NO PUEDEN compararse en términos de "X es mejor que Y", las pruebas más exhaustivas debe realizarse seleccionando varios tipos de cobertura. La elección de los tipos de cobertura depende de dónde se ubican los mayores riesgos y que tipos se adaptan mejor a dichas zona.
- Los tipos de cobertura se pueden dividir en cuatro grupos de cobertura básicos, las técnicas de diseño de pruebas permiten encontrar los casos de pruebas de un grupo concreto.

Grupos de tipos cobertura

- **Proceso**: Los procesos se pueden identificar en varios niveles. Existen algoritmos de control de flujos, procesos de negocio. Se pueden utilizar tipos de cobertura como caminos, cobertura de declaraciones y transiciones de estado para probar estos procesos.
- **Condiciones**: En casi todos los sistemas existen puntos de decisión que consisten en condiciones en las que el comportamiento del sistema puede ir en diferentes direcciones, dependiendo del resultado de dicho punto de decisión. Las variaciones de estas condiciones y sus resultados se pueden probar utilizando tipos de cobertura como cobertura de decisión, cobertura de decisión/condición modificada y cobertura de condición múltiple.
- **Datos**: Los datos se crean y se eliminan cuando finalizan. En el medio, los datos se utilizan consultándolos o actualizándolos. Se puede probar este ciclo de vida de los datos, pero también combinaciones de datos de entrada, así como los atributos (dominios) de los datos de entrada o de salida. Algunos tipos de cobertura aquí son valores límite, CRUD, flujos de datos y sintaxis.
- **Apariencia**: Cómo opera un sistema, cómo se desempeña, cuál debe ser su apariencia, a menudo se describe en requisitos no funcionales. Dentro de este grupo encontramos tipos de cobertura como perfiles operativos, de carga y presentación.

Métricas de código

- La mayor complejidad de las aplicaciones de software moderno también aumenta la dificultad de hacer que el código confiable y fácil de mantener.
- Las métricas de código son un conjunto de medidas de software que proporcionan a los programadores una mejor visión del código que están desarrollando.
- Aprovechando las ventajas de las métricas del código, los desarrolladores pueden entender qué tipos o métodos deberían rehacerse o hacer más pruebas.
- Los equipos de desarrollo pueden identificar los posibles riesgos, comprender el estado actual de un proyecto y realizar un seguimiento del progreso durante el desarrollo de software. Los desarrolladores pueden generar datos de métricas de código con que medir la complejidad y el mantenimiento del código administrado.
- Se insiste mucho en que la cobertura de código de test unitarios de los proyectos sea lo más alta posible, pero es evidente que cantidad (de test, en este caso) no siempre implica calidad, la calidad no se puede medir "al peso", y es la calidad lo que realmente importa. El criterio de la cobertura de código se basa en la suposición que a mayor cantidad de código ejecutada por las pruebas, menor la probabilidad de que el código presente defectos.

Calidad de las pruebas

- La cobertura del código a menudo se considera la métrica estándar de oro para comprender la calidad de las pruebas, y eso es algo desafortunado.
- Un problema aún más insidioso con la cobertura del código es que, al igual que otras métricas, rápidamente se convierte en un objetivo en sí mismo.
- La cobertura de prueba tradicional (líneas, instrucciones, ramas, etc.) solo mide la proporción del código qué participa en las pruebas. No comprueba si las pruebas son realmente capaces de detectar fallos en el código ejecutado, solo pueden identificar el código que no se ha probado. Los ejemplos más extremos del problema son las pruebas sin afirmaciones (poco comunes en la mayoría de los casos). Mucho más común es el código que solo se prueba parcialmente, cubrir todo los caminos no implica ejercitar todas las clases de equivalencia y valores límite.
- La calidad de las pruebas también debe ser puesta a prueba: No serviría de tener una cobertura del 100% en test unitarios, si no son capaces de detectar y prevenir problemas en el código. La herramienta que testea la calidad de los test unitarios son los test de mutaciones: Es un test de los test.

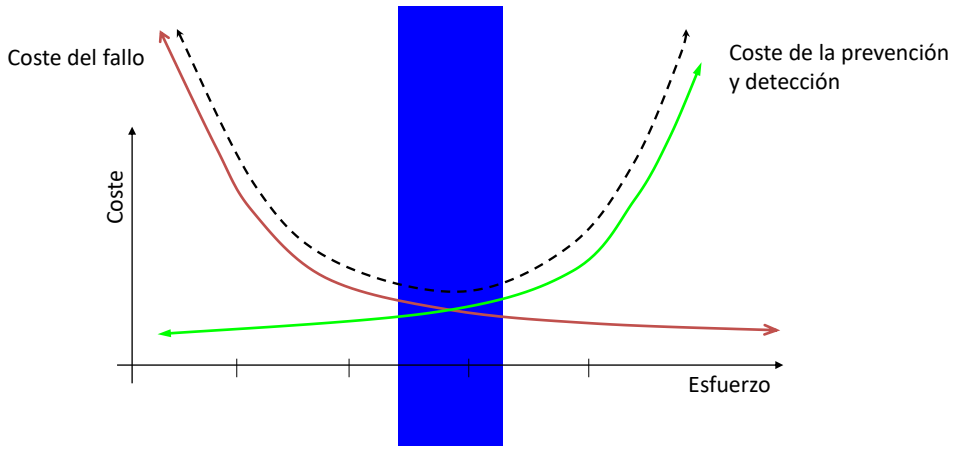
Pruebas de mutaciones

- Los pruebas de mutaciones son las pruebas de las pruebas unitarias y el objetivo es tener una idea de la calidad de las pruebas en cuanto a fiabilidad.
- Su funcionamiento es relativamente sencillo: la herramienta que se utilice debe generar pequeños cambios en el código fuente. A estos pequeños cambios se les conoce como mutaciones y crean mutantes.
- Una vez creados los mutantes, se lanzan todos los tests:
  - Si los test unitarios fallan, es que han sido capaces de detectar ese cambio de código. En este caso el mutante se considera eliminado.
  - Si, por el contrario, los test unitarios pasan, el mutante sobrevive y la fiabilidad (y calidad) de los tests unitarios queda en entredicho.
- Los test de mutaciones presentan informes del porcentaje de mutantes detectados: cuanto más se acerque este porcentaje al 100%, mayor será la calidad de nuestros test unitarios.

Cantidad correcta de pruebas

- Hay un feroz debate que gira en torno a la cantidad correcta de pruebas. Muy pocas pruebas son un problema: las funciones no se especifican correctamente, los errores pasan desapercibidos, ocurren regresiones. Pero demasiadas pruebas consumen tiempo de desarrollo y recursos, no generan ganancias adicionales y ralentizan el desarrollo a largo plazo.
- No es cuestión de hacer muchas pruebas, de hecho, hay que hacer las imprescindibles pero seleccionando buenas pruebas: las que mayor probabilidad tengan de detectar un error y cubran el mayor numero de escenarios.
- Las pruebas difieren en su valor y calidad. Algunas pruebas son más significativas que otras. Los recursos son limitados.
- Esto significa que la calidad de las pruebas es más importante que su cantidad.

Las pruebas tienen un coste



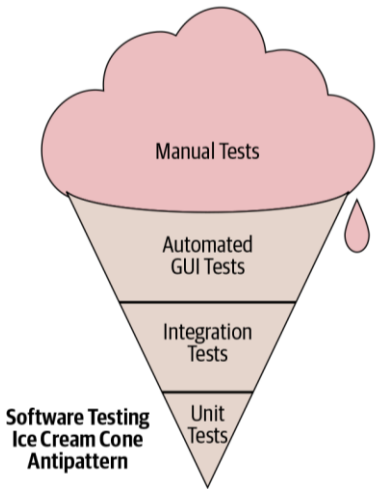
Pirámide de pruebas

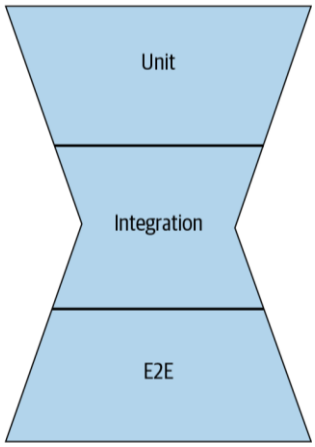
https://martinfowler.com/bliki/TestPyramid.html

Comparativa

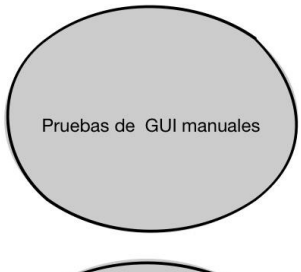
Nivel	Extremo a extremo	Integración	Unitarias
Cobertura	completa	grande	pequeña
Rendimiento	lenta	rápida	muy rápida
Fiabilidad	menos confiable	confiable	mas fiable
Aislar fallos	complicado	justo	fácil
Coste	muy alto	mediano	muy bajo
Simula el usuario real	sí	no	no

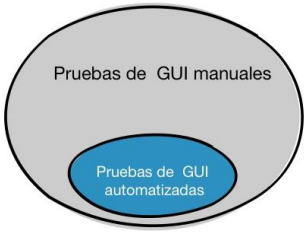
Antipatrones del conjunto de pruebas

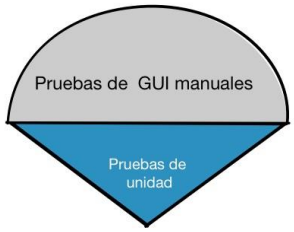


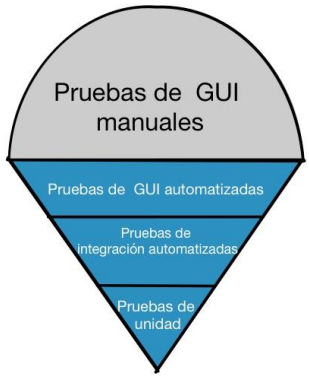


Anti-patrón: El helado









Análisis estático con herramientas

- El objetivo del análisis estático es detectar defectos en el código fuente y en los modelos de software.
- El análisis estático se realiza sin que la herramienta llegue a ejecutar el software objeto de la revisión, como ocurre en las pruebas dinámicas, centrándose mas en como está escrito el código que en como se ejecuta el código.
- El análisis estático permite identificar defectos difíciles de encontrar mediante pruebas dinámicas.
- Al igual que con las revisiones, el análisis estático encuentra defectos en lugar de fallos.
- Las herramientas de análisis estático analizan el código del programa (por ejemplo, el flujo de control y flujo de datos) y las salidas generadas (tales como HTML o XML).
- Algunos de los posibles aspectos que pueden ser comprobados con análisis estático:
  - Reglas, estándares de programación y buenas practicas.
  - Diseño de un programa (análisis de flujo de control).
  - Uso de datos (análisis del flujo de datos).
  - Complejidad de la estructura de un programa (métricas, por ejemplo valor ciclomático).

Valor del análisis estático

- La detección temprana de defectos antes de la ejecución de las pruebas.
- Advertencia temprana sobre aspectos sospechosos del código o del diseño mediante el cálculo de métricas, tales como una medición de alta complejidad.
- Identificación de defectos que no se encuentran fácilmente mediante pruebas dinámicas.
- Detectar dependencias e inconsistencias en modelos de software, como enlaces.
- Mantenibilidad mejorada del código y del diseño.
- Prevención de defectos, si se aprende la lección en la fase de desarrollo.

Defectos habitualmente detectados

- Referenciar una variable con un valor indefinido.
- Interfaces inconsistentes entre módulos y componentes.
- Variables que no se utilizan o que se declaran de forma incorrecta.
- Código inaccesible (muerto).
- Ausencia de lógica o lógica errónea (posibles bucles infinitos).
- Construcciones demasiado complicadas.
- Infracciones de los estándares de programación.
- Vulnerabilidad de seguridad.
- Infracciones de sintaxis del código y modelos de software.

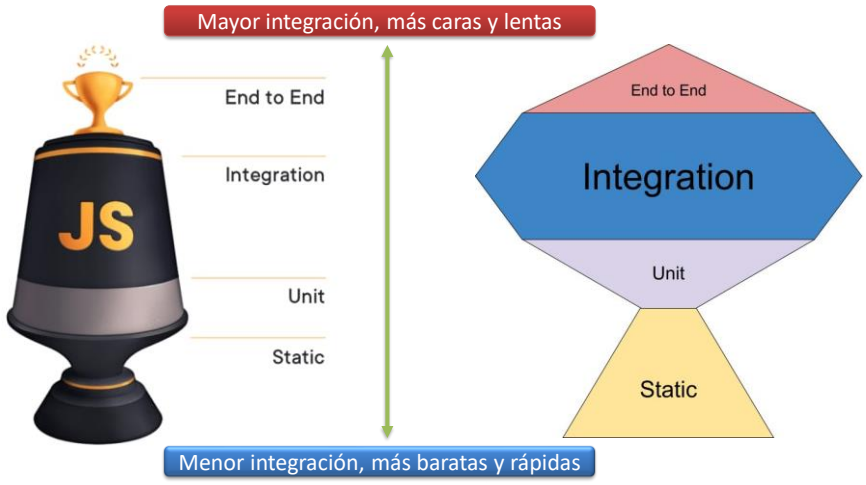
Ejecución del análisis estático

- Las herramientas de análisis estático generalmente las utilizan los desarrolladores (cotejar con las reglas predefinidas o estándares de programación) antes y durante las pruebas unitarias y de integración, o durante la comprobación del código.
- Las herramientas de análisis estático pueden producir un gran número de mensajes de advertencias que deben ser bien gestionados para permitir el uso más efectivo de la herramienta.
- Los compiladores pueden constituir un soporte para los análisis estáticos, incluyendo el cálculo de métricas.
- El Compilador detecta errores sintácticos en el código fuente de un programa, crea datos de referencia del programa (por ejemplo lista de referencia cruzada, llamada jerárquica, tabla de símbolos), comprueba la consistencia entre los tipos de variables y detecta variables no declaradas y código inaccesible (código muerto).
- El Analizador trata aspectos adicionales tales como: Convenciones y estándares, Métricas de complejidad y Acoplamiento de objetos.

El Trofeo de Pruebas

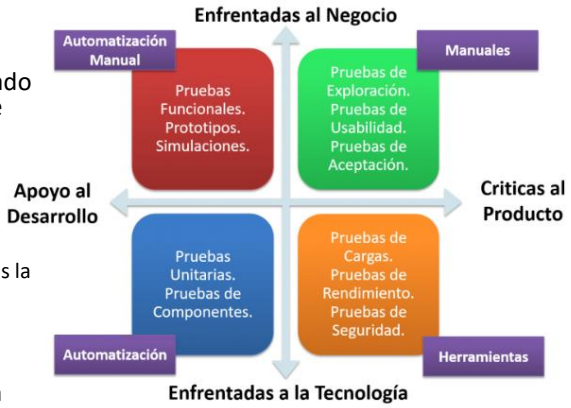
- Testing Trophy es un método de pruebas propuesto por Kent C. Dodds *para aplicaciones web*. Se trata de escribir suficientes pruebas, no muchas, pero si las pruebas correctas: proporciona la mejor combinación de velocidad, costo y confiabilidad.
- Se superpondrán las siguientes técnicas:
  - Usar un sistema de captura de errores de tipo, estilo y de formato utilizando linters, formateadores de errores y verificadores de tipo (ESLint, SonarQube, …).
  - Escribir pruebas unitarias efectivas que apunten solo al comportamiento crítico y la funcionalidad de la aplicación.
  - Desarrollar pruebas de integración para auditar la aplicación de manera integral y asegurarse de que todo funcione correctamente en armonía.
  - Crear pruebas funcionales de extremo a extremo (e2e) para pruebas de interacción automatizadas de las rutas críticas y los flujos de trabajo más utilizados por los usuarios.

Testing Trophy



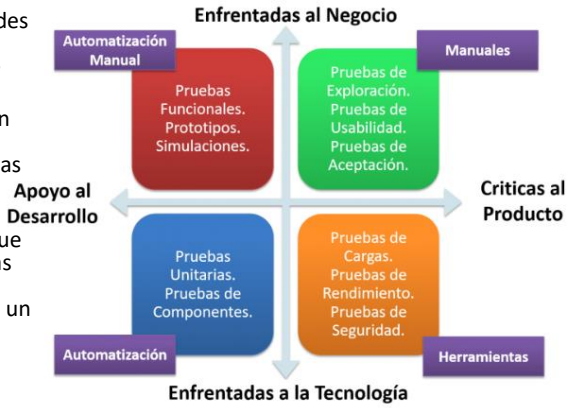
Cuadrante de Pruebas

- El cuadrante de prueba divide todo el proceso de prueba en 4 partes. Esto hace que el proceso de prueba sea fácil de entender.
- El cuadrante de pruebas fue originalmente publicado por Brian Marick en 2003 como parte de una serie de artículos sobre Testeo Ágil y originalmente fue conocido como “Matriz de pruebas de Marick” (Marick Test Matrix”).
- Hay cuatro cuadrantes de prueba que son el resultado de dos ejes.
  - El eje vertical muestra la perspectiva tecnológica versus la perspectiva empresarial.
  - El eje horizontal trata sobre la orientación durante la creación de un producto versus la evaluación del producto una vez que está listo.
- Hay que asegurarse de que se organicen y realicen pruebas para los cuatro cuadrantes.



Cuadrante de Pruebas

- El primer cuadrante (**abajo a la izquierda**) es donde la atención se centra en la tecnología y donde las actividades de prueba guían al equipo en la creación del producto. Suele implicar TDD para las unidades y la integración de las unidades. Las prueba deberían automatizarse.
- El segundo cuadrante (**arriba a la izquierda**) se centra en el negocio y también se centra en guiar al equipo. Esto implica, por ejemplo, el desarrollo impulsado por pruebas de aceptación (ATDD), donde se prueba un proceso de negocio, preferiblemente automatizadas.
- El tercer cuadrante (**arriba a la derecha**) tiene un enfoque empresarial en la evaluación del producto. Estas pruebas se ejecutan principalmente de forma manual, ya sea siguiendo escenarios de prueba preparados o mediante un enfoque exploratorio.
- El cuarto cuadrante (**abajo a la derecha**) se centra en la tecnología y evalúa los aspectos no funcionales de un producto que sólo pueden validarse una vez que el producto esté listo. Las pruebas de rendimiento dinámico son un buen ejemplo de ello.



Diseñar la prueba

- Para diseñar la prueba empiezas por identificar y describir los casos de prueba de cada componente.
- Un **caso de prueba** especifica una forma de probar el sistema, incluyendo la entrada con la que se ha de probar, los resultados que se esperan obtener y las condiciones bajo las que ha de probarse.
- La **base de prueba** es el punto de partida del caso de prueba (requisito, criterio de aceptación, …) y el **objeto de prueba** es el destinatario de la prueba (componente, sistema, documento, …).
- La selección de las técnicas de pruebas depende factores adicionales como pueden ser requisitos contractuales o normativos, documentación disponible, tiempo, presupuesto, conocimientos, experiencia, …
- Cuando dispongas de los casos de prueba, identificas y estructuras los procedimientos de prueba describiendo cómo ejecutar los casos de prueba.

Las pruebas deben

- Tener un objetivo único y un propósito claro.
- Centrarse en el comportamiento antes que en los detalles de la implementación.
- Empezar por los casos válidos (Happy Path) antes de pasar a los casos inválidos (extremos, limites).
- Tener nombres descriptivos y un código limpio, son parte de la documentación.
- Seguir el patrón AAA o una de sus variaciones.
- Ser breves, simples y eficientes.
- Ser deterministas y repetibles.
- Ser independientes entre si (inicializar el entorno), aisladas (dobles de pruebas), no tener efectos secundarios, no influir en el observado.

Casos de prueba de mala calidad

- Sin aserciones
- No comprueban el resultado completo esperado
- No utilizan las aserciones mas especificas
- Cubren múltiples escenarios
- Complejos, con mucho código, no dejan claro que están probando y son frágiles
- Su código apesta, difíciles de entender y mantener
- Dependen de otros casos de pruebas
- Indeterministas, unas veces fallan y otras no, sin cambiar el código fuente ni la prueba
- Cruzan limites, no respetan los limites de las pruebas

Características de una buena prueba unitaria

- El principio FIRST fue definido por Robert Cecil Martin en su libro Clean Code. Este autor, entre otras muchas cosas, es conocido por ser uno de los escritores del Agile Manifesto, escrito hace más de 15 años y que a día de hoy se sigue teniendo muy en cuenta a la hora de desarrollar software.
  - Fast: Los tests deben ser rápidos, del orden de milisegundos en las pruebas unitarias, hay cientos o miles de tests en un proyecto.
  - Isolated/Independent (Aislado/Independiente). Los tests no deben depender del entorno ni de ejecuciones de tests anteriores.
  - Repeatable. Los tests deben ser repetibles y ante el mismo escenario de entrada deben generar los mismos resultados.
  - Self-Validating. Los tests tienen que ser autovalidados, es decir, NO debe de existir la intervención humana en la validación de superado o fallido.
  - Thorough and Timely (Completo y oportuno). Los tests deben de cubrir el escenario propuesto, no el 100% del código, y se han de realizar en el momento oportuno

Patrones

- Los casos de prueba se pueden estructurar siguiendo diferentes patrones:
  - ARRANGE-ACT-ASSERT: Preparar, Actuar, Afirmar
  - GIVEN-WHEN-THEN: Dado, Cuando, Entonces
  - BUILD-OPERATE-CHECK: Generar, Operar, Comprobar
- Con diferencias conceptuales, todos dividen el proceso en tres fases:
  - Una fase inicial donde montar el escenario de pruebas que hace que el resultado sea predecible.
  - Una fase intermedia donde se realizan las acciones que son el objetivo de la prueba.
  - Una fase final (pero la primera en escribirse) donde se comparan los resultados con el escenario previsto. Pueden tomar la forma de:
    - Aserción: Es una afirmación sobre el resultado que puede ser cierta o no.
    - Expectativa: Es la expresión del resultado esperado que puede cumplirse o no.

Preparación mínima

- La sección de preparación, con la entrada del caso de prueba, debe ser lo más sencilla posible, lo imprescindible para comprobar el comportamiento que se está probando.
- Las pruebas se hacen más resistentes a los cambios futuros en el código base y más cercano al comportamiento de prueba que a la implementación.
- Las pruebas que incluyen más información de la necesaria tienen una mayor posibilidad de incorporar errores en la prueba y pueden hacer confusa su intención. Al escribir pruebas, el usuario quiere centrarse en el comportamiento. El establecimiento de propiedades adicionales en los modelos o el empleo de valores distintos de cero cuando no es necesario solo resta de lo que se quiere probar.

Actuación mínima

- Al escribir las pruebas hay que evitar introducir condiciones lógicas como if, switch, while, for, etc.
- Minimiza la posibilidad de incorporar un error a las pruebas.
- El foco está en el resultado final, en lugar de en los detalles de implementación.
- Al incorporar lógica al conjunto de pruebas, aumenta considerablemente la posibilidad de agregar un error. Cuando se produce un error en una prueba, se quiere saber realmente que algo va mal con el código probado y no en el código que prueba. En caso contrario, restan confianza y las pruebas en las que no se confía no aportan ningún valor.
- El objetivo de la prueba debe ser único, si la lógica en la prueba parece inevitable, denota que el objetivo es múltiple y hay que considerar la posibilidad de dividirla en dos o más pruebas diferentes.

Comprobación mínima

- Al escribir las pruebas, hay que intentar comprobar una única cosa, es decir, incluir solo una aserción por prueba, lo mas precisa posible.
  - Si se produce un error en una aserción, no se evalúan las aserciones posteriores.
  - Garantiza que no se estén declarando varios casos en las pruebas.
  - Proporciona la imagen exacta de por qué se producen errores en las pruebas.
- Al incorporar varias aserciones en un caso de prueba, no se garantiza que se ejecuten todas. Es un todas o ninguna, se sabe por cual fallo pero no si el resto también falla o es correcto, proporcionando la imagen parcial.
- Una excepción común a esta regla es cuando la validación cubre varios aspectos. En este caso, suele ser aceptable que haya varias aserciones para asegurarse de que el resultado está en el estado que se espera que esté.
- Los enfoques comunes para usar solo una aserción incluyen:
  - Crear una prueba independiente para cada aserción.
  - Usar pruebas con parámetros.

Características de las pruebas valiosa

- **Las pruebas formalizan y documentan los requisitos.**
  - Un conjunto de pruebas es una descripción formal, legible por humanos y máquinas, de cómo debe comportarse el código. Ayuda a los desarrolladores, en la creación, a comprender los requisitos que deben implementar. Ayuda a los desarrolladores, en el mantenimiento, a comprender los desafíos a que tuvieron que enfrentase los creadores.
  - *Una prueba valiosa describe claramente cómo debe comportarse el código de implementación.* La prueba utiliza un lenguaje adecuado para hablar con los desarrolladores y transmitir los requisitos. La prueba enumera los casos conocidos con los que tiene que lidiar la implementación.
- **Las pruebas son concluyentes, aseguran que el código implemente los requisitos y no muestre errores.**
  - Las pruebas aprovechan cada parte del código para encontrar fallos.
  - *Una prueba valiosa cubre los escenarios importantes*: tanto las entradas correctas como las incorrectas, los casos esperados y los casos excepcionales.

Características de las pruebas valiosa

- **Las pruebas ahorran tiempo y dinero.**
  - Las pruebas intentan cortar los problemas de software de raíz. Las pruebas previenen errores antes de que causen un daño real, cuando todavía son manejables y están bajo control.
  - *Una prueba valiosa es rentable.* La prueba previene errores que, en última instancia, podrían inutilizar la aplicación. La prueba es barata de escribir en comparación con el daño potencial que previene.
- **Las pruebas hacen que el cambio sea seguro al evitar las regresiones.**
  - Las pruebas no solo verifican que la implementación actual cumpla con los requisitos. También verifican que el código aún funcione como se esperaba después de los cambios. Con las pruebas automatizadas adecuadas, es menos probable que se rompa accidentalmente. La implementación de nuevas funciones y la refactorización de código es más segura.
  - *Una prueba valiosa falla cuando se cambia o elimina el código esencial.* Las prueba se diseñan para fallar si se cambia el comportamiento dependiente y deberían seguir pasando ante cambios no dependientes.

Características de las pruebas valiosa

- **Las pruebas respetan los limites.**
  - La pruebas unitarias ejercitan profundamente los componentes de formar aislada centrándose en la funcionalidad, los cálculos, las reglas de dominio y semánticas de los datos. Opcionalmente la estructura del código, es decir, sentencias, decisiones, bucles y caminos distintos.
  - Las pruebas de integración se basan en componentes ya probados (unitaria o integración) o en dobles de pruebas y se centran en la estructura de llamadas, secuencias o colaboración, así como en la transición de estados.
  - Hay muchos tipos de pruebas de sistema y cada uno pone el foco en un aspecto muy concreto, cada prueba solo debe cubrir un solo aspecto.
- **Las pruebas son oportunas.**
  - Las pruebas se crean y ejecutan en el momento oportuno.

AUTOMATIZACIÓN DE PRUEBAS

Automatización de la Prueba

- La automatización de la prueba permite ejecutar muchos casos de prueba de forma consistente y repetida en las diferentes versiones del sistema sujeto a prueba (SSP) y/o entornos. Pero la automatización de pruebas es más que un mecanismo para ejecutar un juego de pruebas sin interacción humana. Implica un proceso de diseño de productos de prueba, entre los que se incluyen:
  - Software.
  - Documentación.
  - Casos de prueba.
  - Entornos de prueba
  - Datos de prueba
- Una Solución de Automatización Pruebas (SAP) debe permitir:
  - Implementar casos de prueba automatizados.
  - Monitorizar y controlar la ejecución de pruebas automatizadas.
  - Interpretar, informar y registrar los resultados de pruebas automatizadas.

Objetivos de la automatización de pruebas

- Mejorar la eficiencia de la prueba.
- Aportar una cobertura de funciones más amplia.
- Reducir el coste total de la prueba.
- Realizar pruebas que los probadores manuales no pueden.
- Acortar el período de ejecución de la prueba.
- Aumentar la frecuencia de la prueba y reducir el tiempo necesario para los ciclos de prueba.

Ventajas y Desventajas de la automatización

Ventajas

- Se pueden realizar más pruebas por compilación.
- La posibilidad de crear pruebas que no se pueden realizar manualmente (pruebas en tiempo real, remotas, en paralelo).
- Las pruebas pueden ser más complejas.
- Las pruebas se ejecutan más rápido.
- Las pruebas están menos sujetas a errores del operador.
- Uso más eficaz y eficiente de los recursos de pruebas
- Información de retorno más rápida sobre la calidad del software.
- Mejora de la fiabilidad del sistema (por ejemplo, repetibilidad y consistencia).
- Mejora de la consistencia de las pruebas.

Desventajas

- Requiere tecnologías adicionales.
- Existencia de costes adicionales.
- Inversión inicial para el establecimiento de la SAP.
- Requiere un mantenimiento continuo de la SAP.
- El equipo necesita tener competencia en desarrollo y automatización.
- Puede distraer la atención respecto a los objetivos de la prueba, por ejemplo, centrándose en la automatización de casos de prueba a expensas del objetivo de las pruebas.
- Las pruebas pueden volverse más complejas.
- La automatización puede introducir errores adicionales.

Limitaciones de la automatización de pruebas

- No todas las pruebas manuales se pueden automatizar.
- La automatización sólo puede comprobar resultados predecibles e interpretables por la máquina.
- La automatización sólo puede comprobar los resultados reales que pueden ser verificados por un oráculo de prueba automatizado.
- No es un sustituto de las pruebas exploratorias.

Técnicas para la automatización de pruebas

- Las herramientas para la automatización de pruebas funcionales aplican técnicas diferentes para la creación y ejecución de pruebas:
  - Técnica de **registro/reproducción** (Record & Playback): Esta técnica consiste en grabar una ejecución de la prueba realizada en la interfaz de la aplicación y su reproducción posterior.
  - Técnica de **Programación de Scripts Estructurados**: Difiere de la técnica de registro/reproducción en la introducción de una librería de scripts reutilizables, que realizan secuencias de instrucciones que se requieren comúnmente en una serie de pruebas. Los script de pruebas se crean en un lenguaje de scripting común.
  - Técnica de **Programación con Frameworks**: Los marcos de pruebas unitarias (unitarias, integración, sistema) simplifican la creación y ejecución de los procedimientos de pruebas en el mismo lenguaje que el objeto de la prueba. La ejecución y cobertura de las pruebas se pueden integrar en los entornos de desarrollo.
    - Unit Test Frameworks: JUnit, MSTest, NUnit, xUnit, Jest, PHPUnit, …
    - Mocking Frameworks: Mockito, Microsoft Fakes, Moq, NSubstitute, Rhino Mocks, …
    - Code Coverage: Jacoco, Clover, NCover, RCov, …
  - Técnica de **Data-Driven**: Esta es una técnica que se enfoca en la separación de los datos de prueba de los scripts, y los almacena en archivos separados. Por lo tanto, los scripts sólo contendrá los procedimientos de prueba y las acciones para la aplicación.
  - Técnica de **Keyword-Driven**: Se basa en la recuperación de los procedimientos de prueba desde los scripts, quedando sólo los datos de prueba y acciones específicas de prueba, que se identifican por palabras clave.

Herramientas de automatización de pruebas

Entornos de pruebas

- Un enfoque de varios entornos permite compilar, probar y liberar código con mayor velocidad y frecuencia para que la implementación sea lo más sencilla posible. Permite quitar la sobrecarga manual y el riesgo de una versión manual y, en su lugar, automatizar el desarrollo con un proceso de varias fases destinado a diferentes entorno.
  - Desarrollo: es donde se desarrollan los cambios en el software.
  - Prueba: permite que los evaluadores humanos o las pruebas automatizadas prueben el código nuevo y actualizado. Los desarrolladores deben aceptar código y configuraciones nuevos mediante la realización de pruebas unitarias en el entorno de desarrollo antes de permitir que esos elementos entren en uno o varios entornos de prueba.
  - Ensayo/preproducción: donde se realizan pruebas finales inmediatamente antes de la implementación en producción, debe reflejar un entorno de producción real con la mayor precisión posible.
  - UAT: Las pruebas de aceptación de usuario (UAT) permiten a los usuarios finales o a los clientes realizar pruebas para comprobar o aceptar el sistema de software antes de que una aplicación de software pueda pasar a su entorno de producción.
  - Producción: es el entorno con el que interactúan directamente los usuarios.

TÉCNICAS

Introducción

- Durante las fases anteriores de definición y de desarrollo, has intentado construir el software partiendo de un concepto abstracto y llegando a una implementación tangible.
- A continuación llega la prueba, debes crear una serie de casos de prueba que intenten demoler el software que ha sido construido. De hecho, la prueba es uno de los pasos de la ingeniería del software que, por lo menos psicológicamente, se puede ver como destructivo en lugar de constructivo.
- Para comenzar vamos a establecer una serie de reglas que sirven como objetivos de prueba:
  1. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
  2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
  3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Introducción

- Los objetivos anteriores suponen un cambio dramático de punto de vista. Nos quitan la idea que, normalmente, tenemos de que una prueba tiene éxito si no descubre errores.
- El objetivo es diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.
- Si la prueba se lleva a cabo con éxito (de acuerdo con el objetivo anteriormente establecido), descubrirá errores en el software.
- Como ventaja secundaria, la prueba demuestra hasta qué punto las funciones del software parecen funcionar de acuerdo con las especificaciones y parecen alcanzarse los requisitos de rendimiento.
- Además, los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan una buena indicación de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo.
- Sin embargo, hay una cosa que no puede hacer la prueba: ”**La prueba no puede asegurar la ausencia de defectos, sólo puede demostrar que existen defectos en el software**”. Es importante tener en mente esta frase pesimista mientras se lleva a cabo la prueba.

Casos de prueba

- El diseño de pruebas para el software puede requerir tanto esfuerzo como el propio diseño inicial del producto.
- Recordando el objetivo de la prueba, debes diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y de tiempo.
- Un **caso de prueba** especifica una forma de probar el sistema, incluyendo la entrada con la que se ha de probar, los resultados que se esperan obtener y las condiciones bajo las que ha de probarse. La **base de prueba** es el punto de partida del caso de prueba (requisito, criterio de aceptación, …) y el **objeto de prueba** es el destinatario de la prueba (componente, sistema, documento, …).
- Un caso de prueba puede separar la lógica de la prueba, **caso de prueba lógico**, de los valores de todas las entradas, salidas y configuraciones requeridas, **caso de prueba física**.

Casos de prueba

- Cualquier producto de ingeniería puede ser probado de una de estas dos formas:
  - **Conociendo la función específica** para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa.
  - **Conociendo el funcionamiento del producto**, se pueden desarrollar pruebas que aseguren que "todas las piezas encajan"; o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada.
- La primera forma, se denomina **prueba de caja negra** o basada en la especificación y la segunda **prueba de caja blanca** o basada en la estructura.

Casos de prueba

- La **prueba de caja negra** o basada en la especificación se refiere a las pruebas que se llevan a cabo sobre la interfaz del software.
- O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa se mantiene.
- Una prueba de caja negra examina algunos aspectos del modelo fundamental del sistema sin tener demasiado en cuenta la estructura lógica interna del software, siendo las **más adecuadas para probar los casos de uso del "modelo de casos de uso"**. Las pruebas de caja negra pueden ser ejecutadas por personal sin experiencia en ingeniería de software.

Casos de prueba

- La **prueba de caja blanca** o basada en la estructura del software se basa en el minucioso examen de los detalles procedimentales.
- Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide con el esperado o afirmado.
- A primera vista parecería que una prueba de caja blanca muy profunda nos llevaría a tener "programas 100% correctos". Todo lo que tienes que hacer es definir todos los caminos lógicos, desarrollar casos de prueba que los ejerciten y evaluar los resultados; es decir, generar casos de prueba que ejerciten exhaustivamente la lógica del programa.

Casos de prueba

- Desgraciadamente, la prueba exhaustiva presenta problemas. Incluso para pequeños programas, el número de caminos lógicos posibles puede ser enorme. La prueba exhaustiva es imposible para los grandes sistemas software. **Las pruebas de caja blanca están especialmente indicadas para validar las realizaciones de casos de uso del "modelo de diseño"**.
- La prueba de caja blanca, sin embargo, no se debe desechar como impracticable.
- Se puede elegir y ejercitar una serie de caminos lógicos importantes. Se pueden comprobar las estructuras de datos más importantes para ver su validez. Se pueden combinar los atributos de la prueba de caja blanca así con los de caja negra, para llegar a un método que valide la interfaz del software y asegure selectivamente que el funcionamiento interno del software es correcto.

Casos de prueba

- Las pruebas de caja blanca y caja negra enfocan las pruebas desde el punto de vista de la ejecución.
- Existe otro punto de vista, el del usuario, donde las pruebas deben ir enfocadas a buscar los errores producidos en **la interfaz con el usuario**.
- Al contrario de las pruebas que exigen la ejecución de software, las pruebas dinámicas, **las pruebas estáticas** se basan en el examen manual (revisiones) y en el análisis automatizado (análisis estático) del código o de cualquier otra documentación del proyecto sin ejecutar el código.
- Formalmente, los casos de prueba escritos consisten principalmente en tres secciones:
  - Identificación
  - Definición
  - Resultados

Identificación

- **Identificador**: Es un identificador único para futuras referencias, por ejemplo, mientras se describe un defecto encontrado.
- **Nombre**: El caso de prueba debe tener un título entendible por las personas, para facilitar la comprensión del propósito del caso de prueba y su campo de aplicación.
- **Propósito**: Contiene una breve descripción del propósito de la prueba y la funcionalidad probada.
- **Creador**: Es el nombre del analista o diseñador de pruebas, quien ha desarrollado pruebas o es responsable de su desarrollo.
- **Versión**: Número de la definición actual del caso de prueba.
- **Referencias**: Identificadores de requerimientos, casos de uso o de especificaciones funcionales que están cubiertos por el caso de prueba.
- **Dependencias**: Orden de ejecución de casos de pruebas, razón de las dependencias.

Definición

- **Precondiciones**: situación previa a la ejecución de pruebas o características de un objeto de pruebas antes de llevar a la práctica (ejecución) un caso de prueba.
- **Valores de entrada**: descripción de los datos de entrada de un objeto de pruebas.
- **Resultados esperados**: datos de salida que se espera que produzca un objeto de prueba.
- **Poscondiciones**: características de un objeto de prueba tras la ejecución de pruebas, descripción de su situación tras la ejecución de las pruebas.
- **Requisitos**: características del objeto de pruebas que el caso de prueba debe evaluar. Contiene información acerca de la configuración del hardware o software en el cuál se ejecutará el caso de prueba.
- **Acciones**: Pasos a realizar para completar la prueba.

Resultados

- **Salida obtenida**: Contiene una breve descripción de lo que el analista encuentra después de que los pasos de prueba se hayan completado.
- **Resultado**: Indica el resultado cualitativo de la ejecución del caso de prueba, a menudo con un Correcto/Fallido.
- **Severidad**: Indica el impacto del defecto en el sistema: Critico, Grave, Normal, Menor.
- **Evidencia**: En los casos que se aplica, contiene información donde se evidencia la salida obtenida y como reproducirla. Puede ir acompañado por un enlace, un pantallazo (screenshot), una consulta a base de datos, el contenido de un fichero de trazas …
- **Seguimiento**: Si un caso de prueba falla, frecuentemente la referencia al defecto implicado se debe enumerar. Contiene el código correlativo del defecto, a menudo corresponde al código del sistema de seguimiento de defectos que se esté usando.
- **Estado**: Indica si el caso de prueba está: No iniciado, En curso o Terminado.

C.O.R.R.E.C.T?

- Conformance.
  - ¿Se muestran los valores de la forma esperada?
- Ordering.
  - ¿Se presentan los valores apropiadamente ordenados/desordenados?
- Range.
  - ¿Los valores generados están dentro del rango esperado [max, min]?
- Reference.
  - ¿Se refiere el código a algo externo que existe?
- Existence.
  - ¿Existe el valor (es no nulo), está presente en un array, etc?
- Cardinality.
  - ¿Se obtiene el número deseado de elementos o valores?
- Time.
  - ¿Ocurren las cosas cuando deben?

Técnica de diseño de pruebas

- Una técnica de diseño de pruebas es un método estandarizado para encontrar, a partir de una base de prueba específica, los casos de prueba que logran una cobertura específica.
- El uso de técnica de diseño de pruebas proporciona una elaboración justificada de la estrategia de prueba: la cobertura acordada en el lugar acordado. Debido a que una técnica de diseño de pruebas se centra en lograr una cobertura específica para detectar tipos específicos de defectos (por ejemplo, en las interfaces, las comprobaciones de entrada o el procesamiento), dichos defectos se detectan de manera más efectiva que especificando casos de prueba ad hoc.
- Las pruebas son reproducibles porque el orden y el contenido de la ejecución de la prueba se describen detalladamente. El método estandarizado garantiza que el proceso de prueba sea independiente del individuo, que especifica y ejecuta los casos de prueba, y que las especificaciones de la prueba sean transferibles y mantenibles. Resulta más fácil planificar y gestionar el proceso de prueba porque los procesos de especificación y ejecución de la prueba se pueden dividir en bloques claramente definibles.
- Disponemos de múltiples técnicas a seleccionar en función al estrategia, característica o cobertura buscada.

TÉCNICAS ESTÁTICAS

Técnicas estáticas

- Al contrario que las pruebas dinámicas, que exigen la ejecución de software, las técnicas de pruebas estáticas se basan en el examen manual (revisiones) y en el análisis automatizado (análisis estático) del código o de cualquier otra documentación del proyecto sin ejecutar el código.
- Las revisiones constituyen una forma de probar los productos de trabajo del software (incluyendo el código) y pueden realizarse antes de ejecutar las pruebas dinámicas. Los defectos detectados durante las revisiones al principio del ciclo (por ejemplo, los defectos encontrados en los requisitos) a menudo son mucho más baratos de eliminar que los detectados durante las pruebas realizadas ejecutando el código.
- Una revisión podría hacerse íntegramente como una actividad manual, pero también existen herramientas de soporte.

Técnicas estáticas

- Las revisiones, el análisis estático y las pruebas dinámicas tienen el mismo objetivo: Identificar defectos.
  - Se trata de procesos complementarios.
  - Las distintas técnicas pueden encontrar distintos tipos de defectos de una manera eficiente y efectiva.
  - En comparación con las pruebas dinámicas, las técnicas estáticas localizan las causas o los defectos más que los propios fallos.
- Entre los defectos típicos que resultan más fáciles de localizar en revisiones que en las pruebas dinámicas se incluyen:
  - Desviaciones de los estándares,
  - Defectos de requisitos,
  - Defectos de diseño,
  - Mantenibilidad insuficiente y
  - Especificaciones de interfaz incorrectas.

Revisión

- La principal actividad manual consiste en examinar un producto de trabajo y hacer comentarios al respecto. Cualquier producto de trabajo de software puede ser objeto de una revisión como por ejemplo:
  - Las especificaciones de requisitos,
  - Las especificaciones de diseño,
  - El código,
  - Los planes de pruebas,
  - Las especificaciones de pruebas,
  - Los casos de pruebas,
  - Los guiones de pruebas,
  - Las guías de usuario o las páginas web.
- Los beneficios de las revisiones incluyen la detección y corrección temprana de defectos, el desarrollo de mejoras de productividad, la reducción de los tiempos de desarrollo, el ahorro de tiempo y dinero invertido en pruebas, el menor coste de vida, menos defectos y comunicación mejorada. Las revisiones pueden encontrar omisiones, por ejemplo, en requisitos, que no suelen encontrarse en pruebas dinámicas.

Tipos de revisiones

- **Revisión Informal**: Es una forma barata de obtener beneficios, sin un proceso formal, cuyos resultados pueden estar documentados. Su utilidad varía en función de los revisores.
- **Revisión guiada**: Reunión guiada por el autor del artefacto, cuyo objetivo principal es aprender, entender y encontrar defectos. En la práctica puede variar desde bastante informal hasta muy formal, siendo opcional la elaboración de un informe de revisión en el que se incluya la lista de conclusiones.
- **Revisión técnica**: Es un proceso formal, documentado y definido para la detección de defectos cuyos objetivos principales son: debatir, tomar decisiones, evaluar alternativas, encontrar defectos, resolver problemas técnicos y comprobar la conformidad con las especificaciones, los planes, la normativa y los estándares. El informe de revisión debe incluir la lista de conclusiones, el veredicto de si el artefacto cumple los requisitos y, si procede, recomendaciones en base a las conclusiones. Requiere preparación previa a la reunión.
- **Inspección:** Dirigida por un especialista, es un proceso formal basado en normas y listas de comprobación cuyo objetivo principal es identificar defectos. Incluye una recopilación de métricas, criterios de entrada y salida especificados para la aceptación del producto de software y un proceso de seguimiento formal. El informe de revisión debe incluir la lista de conclusiones. Requiere preparación previa a la reunión.

Actividades de una revisión

- Una revisión formal típica incluye las siguientes actividades principales:
  1.- **Planificar**:
    - Definir los criterios de revisión.
    - Seleccionar al personal.
    - Asignar funciones.
  2.- **Definir los criterios de entrada y salida** (para tipos de revisión más formales).
    - Seleccionar qué partes de los documentos deben revisarse.
  3.- **Inicio**:
    - Repartir los documentos.
    - Explicar los objetivos, procesos y documentos a los participantes.
  4.- **Comprobar los criterios de entrada** (para tipos de revisión más formales).
  5.- **Preparación individual**.
    - Prepararse para la reunión de revisión repasando los documentos.

Actividades de una revisión

6.- **Prestar especial atención a posibles defectos, preguntas y comentarios**.

7.- **Examen/evaluación/registro de los resultados** (reunión de revisión):
  - Debatir o registrar, mediante resultados documentados o actas (para tipos de revisión más formales).
  - Prestar especial atención a los defectos, hacer recomendaciones sobre cómo manejar los defectos, tomar decisiones al respecto.

8.- **Examinar/evaluar y registrar** durante reuniones físicas o de seguimiento de los grupos comunicaciones electrónicas.

9.- **Adaptar**:
  - Corregir los defectos detectados (normalmente a cargo del autor).
  - Registrar el estado actualizado de los defectos (en revisiones formales).

10.- **Hacer un seguimiento**:
  - Comprobar que los defectos han sido tratados.
  - Recopilar métricas.

11.- **Comprobar los criterios de salida** (para tipos de revisión más formales).

Análisis estático con herramientas

- El objetivo del análisis estático es detectar defectos en el código fuente y en los modelos de software.
- El análisis estático se realiza sin que la herramienta llegue a ejecutar el software objeto de la revisión, como ocurre en las pruebas dinámicas, centrándose mas en como está escrito el código que en como se ejecuta el código.
- El análisis estático permite identificar defectos difíciles de encontrar mediante pruebas dinámicas. Al igual que sucede con las revisiones, el análisis estático encuentra defectos en lugar de fallos.
- Las herramientas de análisis estático analizan el código del programa (por ejemplo, el flujo de control y flujo de datos) y las salidas generadas (tales como HTML o XML).
- Algunos de los posibles aspectos que pueden ser comprobados con análisis estático:
  - Reglas, estándares de programación y buenas practicas.
  - Diseño de un programa (análisis de flujo de control).
  - Uso de datos (análisis del flujo de datos).
  - Complejidad de la estructura de un programa (métricas, por ejemplo valor ciclomático).

Valor del análisis estático

- La detección temprana de defectos antes de la ejecución de las pruebas.
- Advertencia temprana sobre aspectos sospechosos del código o del diseño mediante el cálculo de métricas, tales como una medición de alta complejidad.
- Identificación de defectos que no se encuentran fácilmente mediante pruebas dinámicas.
- Detectar dependencias e inconsistencias en modelos de software, como enlaces.
- Mantenibilidad mejorada del código y del diseño.
- Prevención de defectos, si se aprende la lección en la fase de desarrollo.

Defectos habitualmente detectados

- Referenciar una variable con un valor indefinido.
- Interfaces inconsistentes entre módulos y componentes.
- Variables que no se utilizan o que se declaran de forma incorrecta.
- Código inaccesible (muerto).
- Ausencia de lógica o lógica errónea (posibles bucles infinitos).
- Construcciones demasiado complicadas.
- Infracciones de los estándares de programación.
- Vulnerabilidad de seguridad.
- Infracciones de sintaxis del código y modelos de software.

Ejecución

- Las herramientas de análisis estático generalmente las utilizan los desarrolladores (cotejar con las reglas predefinidas o estándares de programación) antes y durante las pruebas unitarias y de integración, o durante la comprobación del código.
- Las herramientas de análisis estático pueden producir un gran número de mensajes de advertencias que deben ser bien gestionados para permitir el uso más efectivo de la herramienta.
- Los compiladores pueden constituir un soporte para los análisis estáticos, incluyendo el cálculo de métricas.
- El Compilador detecta errores sintácticos en el código fuente de un programa, crea datos de referencia del programa (por ejemplo lista de referencia cruzada, llamada jerárquica, tabla de símbolos), comprueba la consistencia entre los tipos de variables y detecta variables no declaradas y código inaccesible (código muerto).
- El Analizador trata aspectos adicionales tales como: Convenciones y estándares, Métricas de complejidad y Acoplamiento de objetos.

Pruebas de mantenibilidad

- Los equipos suelen centrarse en el desarrollo rentable de sistemas de TI. El mantenimiento ocupa aproximadamente el 70% de la vida útil completa del sistema de TI. En el costo total de propiedad, los costos de mantenimiento suelen ser mucho más altos que los costos de desarrollo. El desarrollo rápido y sucio (que causa una gran deuda técnica) puede derivar en costos totales más altos que los sistemas bien diseñados y desarrollados. El estándar ISO25010 define cinco subcaracterísticas de mantenibilidad dentro de las llamadas no funcionales:
  - Modularidad: el grado en que un sistema o programa está compuesto de componentes discretos, de modo que un cambio en un componente tiene un impacto mínimo en otros componentes.
  - Reutilizabilidad: el grado en que un activo se puede utilizar en más de un sistema o en la construcción de otros activos.
  - Analizabilidad: el grado de efectividad y eficiencia con el que es posible evaluar el impacto en un producto o sistema de un cambio previsto en una o más de sus partes, o diagnosticar un producto en busca de deficiencias o causas de fallas, o identificar partes. ser modificado.
  - Modificabilidad: el grado en que un producto o sistema puede modificarse efectiva y eficientemente sin introducir fallas ni degradar la calidad del producto existente.
  - Capacidad de prueba: el grado de efectividad y eficiencia con el que se pueden establecer criterios de prueba para un sistema, producto o componente y se pueden realizar pruebas para determinar si se han cumplido esos criterios.

Mantenibilidad

- La mantenibilidad consiste en modificar un sistema de TI de manera fácil y efectiva. Si se dedica más esfuerzo a mantener el código existente que a escribir código nuevo, esto es un indicio de que la capacidad de mantenimiento puede ser deficiente. Las modificaciones pueden incluir correcciones, mejoras o adaptaciones del software. Pero también se relaciona con cambios en el entorno y en los requisitos, especificaciones e historias de usuarios. También se refiere a la instalación de actualizaciones y mejoras. Los indicadores de mantenibilidad son la calidad del código, la calidad de la documentación, la calidad de las pruebas y muchos otros.
- Esta variedad de prueba no debe confundirse con las pruebas de mantenimiento. En las pruebas de mantenimiento, el motivo de la prueba es que se cambió el sistema. En las pruebas de mantenibilidad, el motivo de las pruebas es ver qué tan bien se puede actualizar, cambiar y mantener el sistema.
- La mantenibilidad se determinan midiendo un conjunto de propiedades del producto de software. Estas propiedades son volumen, duplicación, complejidad de la unidad, tamaño de la unidad, interfaz de la unidad, acoplamiento de módulos, equilibrio de componentes e independencia de los componentes

Medición

- Los artefactos de software que se pueden evaluar son, por ejemplo, archivos de código fuente, archivos de definición de datos, archivos de manipulación de datos y contenido activo de páginas web.
- Las pruebas de mantenibilidad pueden realizarse o respaldarse con herramientas automatizadas.
- La medición estática, normalmente con herramientas automatizadas, puede obtener:
  - Volumen del código, por ejemplo medido en líneas de código.
  - Complejidad ciclomática de una unidad y/o del sistema en su conjunto.
  - Proporción de comentario a código
  - Profundidad de anidamiento de declaraciones de decisión
  - Acoplamiento: número de elementos de datos pasados entre módulos
- La medición dinámica de la mantenibilidad consiste principalmente en cronometrar la duración de tareas de mantenimiento específicas.
  - Tiempo transcurrido para implementar un cambio
  - Estabilidad del sistema, por ejemplo, tiempo medio entre fallos.

Mantenibilidad del código

- Tenemos disponibles toda una serie de principios, patrones de diseño y buenas practicas para facilitar la mantenibilidad de los sistemas.
- Cuanto más complejo es un sistema más probable es que falle y mas difícil de entender, probar y mantener.
- El clean code (código limpio) esta basado en que el código se escribe una vez y se lee muchas veces, reúne una serie de principios que ayudan a producir código legible, intuitivo y fácil de modificar. El código que no sigue estos principios se denomina smell code (código apestoso).
- Cualquier proyecto no sólo debería tener código limpio, sino también una arquitectura limpia con un bajo acoplamiento (forma y nivel de las interdependencia entre las partes) y una alta cohesión (grado de por que los elementos de una parte están juntos). Esto conduce a una menor complejidad del módulo, una mayor reutilización del módulo y una mejor mantenibilidad del sistema, lo que hace que el código sea más fácil de entender, probar y depurar.
- Disponemos de principios arquitectónicos como:
  - SOLID (Principios de responsabilidad única, abierto-cerrado, sustitución de Liskov, segregación de interfaz e inversión de dependencia): se utiliza para hacer que el código de producción sea comprensible, flexible y mantenible.
  - YAGNI (You Aren't Gonna Need it o "no lo vas a necesitar"): Mantenga limpia su base de código y no desarrolle código que no sea necesario. Elimine el código si ya no se usa.
  - KISS (beso, Keep It Simple, Stupid, Hazlo simple, estúpido): No diseñe demasiado la base del código del sistema de TI. El código simple es más fácil de mantener y comprender.
  - DRY (seco, Don't Repeat Yourself, o no te repitas"): No repitas código, refactoriza. Lo contrario de DRY es WET (húmedo, We Enjoy Typing o nos lo pasamos bien tecleando).

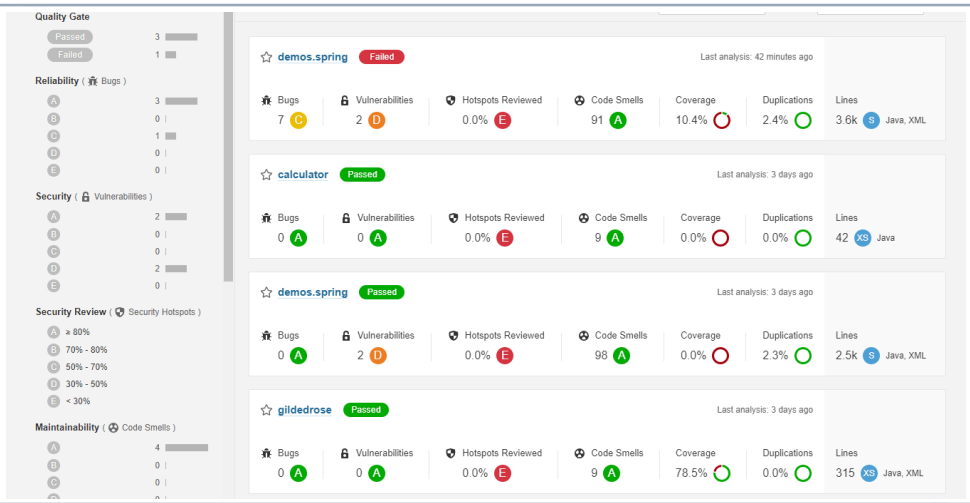
http://www.sonarqube.org/

SONARQUBE

Introducción

- SonarQube (conocido anteriormente como Sonar) es una herramienta de revisión automática de código para detectar errores, vulnerabilidades y malos olores en su código. Puede integrarse con su flujo de trabajo existente para permitir la inspección continua de código en las ramas de su proyecto y las solicitudes de incorporación de cambios.
- SonarQube es una plataforma para la revisión y evaluación del código fuente. Es una herramienta esencial para la fase de pruebas y auditoría de código dentro del ciclo de desarrollo de una aplicación y se considera perfecta para guiar a los equipos de desarrollo durante las revisiones de código.
- Es open source y realiza el análisis estático de código fuente integrando las mejores herramientas de medición de la calidad de código como Checkstyle, PMD o FindBugs, para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.
- Informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad ciclomática, posible errores, comentarios y diseño del software.
- Aunque pensado para Java, acepta mas de 20 lenguajes mediante extensiones. Se integra con Maven, Ant y herramientas de integración continua como Atlassian, Jenkins y Hudson.

Sonar



Proceso

1. Los desarrolladores desarrollan y combinan código en un IDE (preferiblemente usando SonarLint para recibir comentarios inmediatos en el editor) y registran su código en su plataforma DevOps.
2. La herramienta de integración continua (CI) de una organización verifica, crea y ejecuta pruebas unitarias, y un escáner SonarQube integrado analiza los resultados.
3. El escáner publica los resultados en el servidor de SonarQube, que proporciona comentarios a los desarrolladores a través de la interfaz de SonarQube, correo electrónico, notificaciones en el IDE (a través de SonarLint) y decoración en las solicitudes de extracción o combinación (cuando se usa Developer Edition y superior).

Beneficios

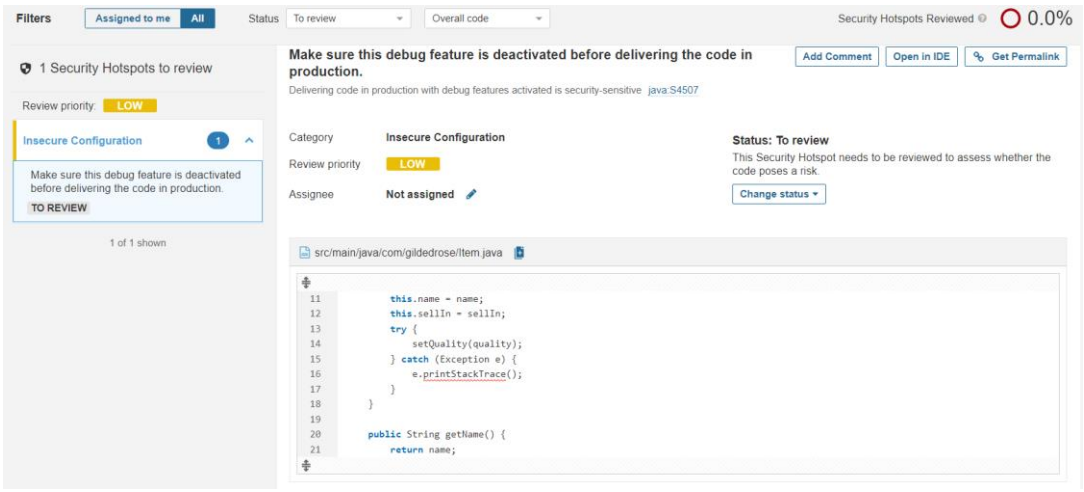
- Alerta de manera automática a los desarrolladores de los errores de código para corregirlos previamente a la implementación en producción.
- No sólo muestra los errores, también las reglas de codificación, la cobertura de las pruebas, las duplicaciones, la complejidad y la arquitectura, plasmando todos estos datos en paneles de control detallados.
- Ayuda al equipo a mejorar en sus habilidades como programadores al facilitar un seguimiento de los problemas de calidad.
- Permite la creación de paneles y filtros personalizables para centrarse en áreas clave y entregar productos de calidad a tiempo.
- Favorece la productividad al reducir la complejidad del código acortando tiempos y costes adicionales al evitar cambiar el código constantemente.

Herramientas

- SonarLint: es un producto complementario que funciona en su editor y brinda comentarios inmediatos para que pueda detectar y solucionar problemas antes de que lleguen al repositorio.
- Quality Gate: permite saber si su proyecto está listo para la producción.
- Clean as You Code: es un enfoque de la calidad del código que elimina muchos de los desafíos que conllevan los enfoques tradicionales. Como desarrollador, se enfoca en mantener altos estándares y asumir la responsabilidad específicamente en el Nuevo Código en el que está trabajando.
- Issues: SonarQube plantea problemas cada vez que una parte de su código infringe una regla de codificación, ya sea un error que romperá su código (bug), un punto en su código abierto a ataques (vulnerabilidad) o un problema de mantenimiento (código apestoso).
- Security Hotspots: Puntos de acceso de seguridad destaca piezas de código sensibles a la seguridad que deben revisarse. Tras la revisión, descubrirá que no hay ninguna amenaza o que debe aplicar una solución para proteger el código.

Problemas

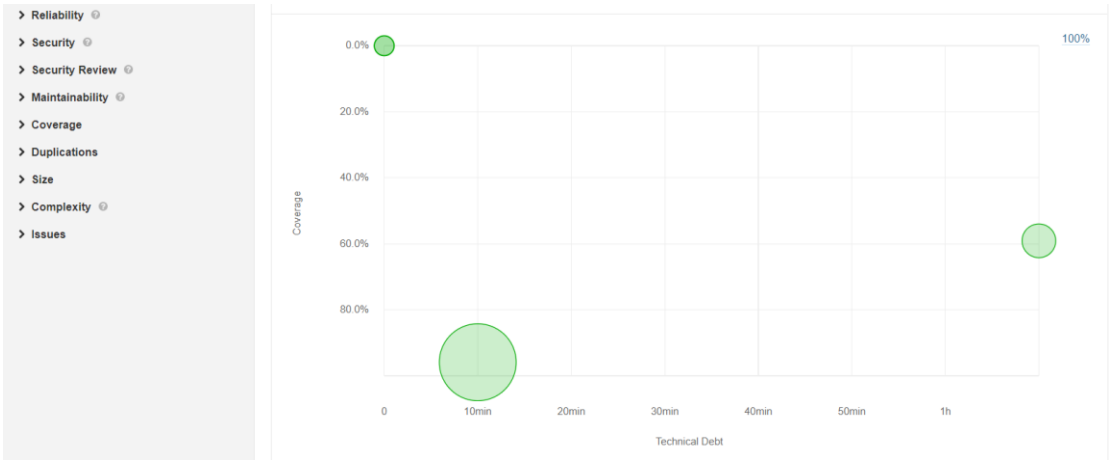
Puntos de acceso de seguridad



Métricas principales

- **Complejidad**: Refleja la Complejidad Ciclomática calculada en base al número de caminos a través del código normalmente observado a nivel de métodos o funciones individuales.
- **Duplicados**: Nos indica el número de bloques de líneas duplicados. Ayuda a evitar resultados distintos en operaciones iguales.
- **Evidencias**: Son los fragmentos nuevos de código de un proyecto que detecta que incumplen con alguna de las reglas establecidas.
- **Mantenibilidad**: Se refiere al recuento total de problemas de Code Smell.
- **Umbrales de calidad**: Define los requisitos del proyecto antes de ser lanzado a producción, como, por ejemplo, que no deben haber evidencias bloqueantes o la cobertura de código sobre el código nuevo debe ser mayor que el 80%.
- **Tamaño**: Permiten hacerse una idea del volumen del proyecto en términos generales.
- **Pruebas**: Son una forma de comprobar el correcto funcionamiento de una unidad de código y de su integración.

Métricas



Análisis

- SonarQube puede analizar mas de 20 lenguajes diferentes según la edición. El resultado de este análisis serán métricas y problemas de calidad (casos en los que se rompieron las reglas de codificación). Sin embargo, lo que se analiza variará según el lenguaje:
  - En todos los lenguajes, los datos de "culpa" se importarán automáticamente de los proveedores de SCM admitidos. Git y SVN son compatibles automáticamente.
  - En todos los lenguajes se realiza un análisis estático del código fuente (archivos Java, programas COBOL, etc.)
  - Se puede realizar un análisis estático del código compilado para ciertos lenguajes (archivos .class en Java, archivos .dll en C#, etc.)
- Durante el análisis, se solicitan datos del servidor, se analizan los archivos proporcionados para el análisis enfrentándolos a las reglas y los datos resultantes se envían de vuelta al servidor al final en forma de informe, que luego se analiza de forma asíncrona en el lado del servidor.

Reglas

- Una regla es un estándar o práctica de codificación que debe seguirse. El incumplimiento de las reglas de codificación conduce a errores, vulnerabilidades, puntos críticos de seguridad y código apestoso. Las reglas pueden comprobar la calidad de los archivos de código o las pruebas unitarias.
- Cada lenguaje de programación dispone de sus propias reglas, siendo diferentes en cantidad y tipo.
- Las reglas se clasifican en:
  - Bug: un punto de fallo real o potencial en su software
  - Code Smell: un problema relacionado con la mantenibilidad en el código.
  - Vulnerability: un punto débil que puede convertirse en un agujero de seguridad que puede usarse para atacar su software.
  - Security Hotspot: un problema que es un agujero de seguridad.
- Para Bugs y Code Smells and Bugs, no se esperan falsos positivos. Al menos este es el objetivo para que los desarrolladores no tengan que preguntarse si se requiere una solución. Para las vulnerabilidades, el objetivo es que más del 80 % de los problemas sean verdaderos positivos. Las reglas de Security Hotspot llaman la atención sobre el código que es sensible a la seguridad. Se espera que más del 80% de los problemas se resuelvan rápidamente como "Revisados" después de la revisión por parte de un desarrollador.

Control de calidad

- Los perfiles de calidad (Quality Profile) son un componente central de SonarQube donde se definen conjuntos de reglas que, cuando se violan, generan problemas en la base de código (ejemplo: los métodos no deben tener una complejidad cognitiva superior a 15). Cada lenguaje individual tiene su propio perfil de calidad predeterminado y los proyectos que no están asignados explícitamente a perfiles de calidad específicos se analizan utilizando los perfiles de calidad predeterminados.
- Un umbral de calidad (Quality Gate) es un conjunto de condiciones booleanas basadas en métricas. Ayudan a saber inmediatamente si los proyectos están listos para la producción. Idealmente, todos los proyectos utilizarán la misma barrera de calidad. El estado de Quality Gate de cada proyecto se muestra de forma destacada en la página de inicio.

Scanner

- Una vez que se haya instalado la plataforma SonarQube, estará listo para instalar un escáner y comenzar a crear proyectos. Para ello, se debe instalar y configurar el escáner más adecuado pare el proyecto:
  - Gradle - SonarScanner para Gradle
  - Ant - SonarScanner para Ant
  - Maven: use el SonarScanner para Maven
  - .NET - SonarScanner para .NET
  - Jenkins - SonarScanner para Jenkins
  - Azure DevOps - Extensión de SonarQube para Azure DevOps
  - Cualquier otra cosa (CLI) - SonarScanner

SonarScanner para Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.10.0</version>
        </plugin>
        <plugin>
            <groupId>org.sonarsource.scanner.maven</groupId>
            <artifactId>sonar-maven-plugin</artifactId>
            <version>3.9.1.2184</version>
        </plugin>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.7</version>
        </plugin>
    </plugins>
</build>
```


SonarScanner para Maven

```
<profiles>
    <profile>
        <id>coverage</id><activation><activeByDefault>true</activeByDefault></activation>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.jacoco</groupId>
                    <artifactId>jacoco-maven-plugin</artifactId>
                    <executions>
                        <execution>
                            <id>prepare-agent</id><goals><goal>prepare-agent</goal></goals>
                        </execution>
                        <execution>
                            <id>report</id><goals><goal>report</goal></goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

Analizando

- Analizar un proyecto Maven consiste en ejecutar un objetivo Maven: sonar:sonar desde el directorio que contiene el proyecto principal pom.xml.
- Se debe pasar un token de autenticación usando la propiedad sonar.login en la línea de comando.
  - mvn sonar:sonar -Dsonar.projectKey=Microservicios -Dsonar.host.url=http://localhost:9000 -Dsonar.login=1eeaaf436541...
- Para obtener información de cobertura, deberá generar el informe de cobertura antes del análisis.
  - mvn clean verify sonar:sonar

JavaScript

- El SonarScanner es el escáner que se debe usar cuando no hay un escáner específico para su sistema de compilación. Para descargar:
  - https://docs.sonarqube.org/8.9/analysis/scan/sonarscanner/
- Crear un archivo de configuración en el directorio raíz del proyecto llamado sonar-project.properties

```
sonar.projectKey=<projectKey>
sonar.projectName=<projectName>
sonar.projectVersion=1.0
sonar.language=js
sonar.sources=src
sonar.sourceEncoding=UTF-8
#sonar.host.url=http://localhost:9000
```

- Para ejecutar el scaner:
  - sonar-scanner.bat -D"sonar.projectKey=Web4Testing" -D"sonar.sources=." -D"sonar.host.url=http://localhost:9000" -D"sonar.login=9ea...51a3a"

TÉCNICAS DINÁMICAS

Pruebas de Caja Negra

- La prueba de la caja negra es un método de diseño de casos de prueba que se centra en los requerimientos funcionales del software para descubrir los casos de prueba.
- Los errores que intentan encontrar son:
  - Funciones incorrectas o ausentes.
  - Errores de interfaz.
  - Errores de estructuras de datos.
  - Errores de rendimiento.
  - Errores de inicialización y de terminación.
- Van orientadas al dominio de información de los datos, es decir, al conjunto de diferentes valores de cada variable: de entrada / salida, de interfaz, variables de programa...

Técnica de Partición de Equivalencia

- Trata de obtener un pequeño subconjunto de todas las posibles entradas con la mayor probabilidad de encontrar todos los errores.
- Todo caso de prueba bien seleccionado debe cumplir:
  - Reducir, en un coeficiente mayor que uno, el número de casos de prueba adicionales que se deben realizar para alcanzar una prueba razonable. Es decir, debe incluir tantas condiciones de entrada distintas como sea posible.
  - Cubrir un gran conjunto de posibles casos de prueba. Es decir, nos dice algo sobre la presencia o ausencia de un error asociado solamente con la prueba en particular que se encuentra disponible.
- Debemos intentar dividir el dominio de entrada de un programa en un número finito de clases de equivalencia tal, que se pueda asumir de forma razonable (aunque nunca estaremos totalmente seguros) que probar un valor representativo de cada clase es equivalente a probar algún otro valor.

Clases de equivalencia

- Es decir, si un caso de prueba en una clase de equivalencia detecta un error, todos los demás posibles casos de prueba en la clase de equivalencia deberán detectar el mismo error. Y también, si un caso de prueba no detecta un error, podríamos pensar que ningún otro caso de prueba en esa clase de equivalencia encontraría ese error.
- La **estrategia** a seguir se realiza en dos etapas:
  1. Identificar cada una de las clases de equivalencia.
  2. Definir los casos de prueba.
- Las **clases de equivalencia** las identificas dividiendo en dos o más grupos cada condición de entrada (normalmente ésta es una frase en la especificación). Se identifican siempre dos tipos de Clases de Equivalencia:
  1. Clases de equivalencia válidas.
  2. Clases de equivalencia inválidas (entradas erróneas).

Directrices para definir las clases de equivalencia

- Si una condición de entrada especifica un rango de valores (p. e. entre 1 y 999) se define una clase de equivalencia válida (1 <= valor <= 999) y dos inválidas (valor < 1 y valor > 999).
- Si una condición de entrada especifica una situación "debe ser" (por ejemplo: "el primer carácter del N.I.F. debe ser una letra"), se define una clase de equivalencia válida (es una letra) y otra inválida (no es una letra).
- Si una condición de entrada especifica un conjunto de valores de entrada y sabemos que cada valor va a tratarse de forma diferente en el programa, se define una clase de equivalencia válida para cada valor y una clase de equivalencia inválida (por ejemplo, "los colores pueden ser: ROJO, AMARILLO y VERDE", una clase inválida incluye "AZUL").
- Si una condición de entrada especifica el número de valores (ejemplo: una persona puede tener desde 1 hasta 3 nombres), identificar una clase de equivalencia válida y 2 inválidas (sin nombre o hay más de 3).
- Si tenemos una razón para creer que los elementos de una clase de equivalencia no se tratan de la misma manera en un programa, dividiremos la clase de equivalencia en clases más pequeñas.

Tabla de equivalencias

- Según vas identificando las condiciones y las clases de equivalencia, **rellenas la siguiente tabla:**

Condiciones de Entrada	Clases de equivalencia validas	Clases de equivalencia invalidas
La duración es	55	0 1278
El semáforo esta	“ROJO” ”AMARILLO” “VERDE"	“AZUL”
El NIF es	“Z12345678”	“12345678Z”

Definir los casos de prueba

- Para **definir los casos de prueba** das los siguientes **pasos**:
  1. Asignas un número único a cada clase de equivalencia.
  2. Escribes casos de prueba hasta que sean cubiertas todas las clases de equivalencia válidas, intentando cubrir en cada caso tantas clases de equivalencia como sea posible.
  3. Escribes casos de prueba hasta que sean cubiertas todas las clases de equivalencia inválidas cubriendo en cada caso una y sólo una clase de equivalencia aún no cubierta.
- La razón por la que las clases de equivalencia inválidas se cubren de forma individual se debe a que al detectar una entrada errónea seguramente no se chequea la siguiente entrada.

Ejemplo

- Tomemos como ejemplo de un módulo lee 3 valores enteros. Los 3 valores son la longitud de los 3 lados de un triángulo. El módulo devuelve un valor diciendo si el triángulo es escaleno, isósceles o equilátero.
- Aplicando el método:

Condiciones de Entrada	Clases de equivalencia validas	Clases de equivalencia invalidas
El número de valores es	(1) 3	(2) 2 (3) 4
Los valores son	(4) enteros	(5) cualquier otro dato
Valores mayores de 0	(6) todos >0	(7) alguno <=0

Casos de prueba

- Los casos de prueba resultantes son:

Valores de entrada: 3, 5, 4	(engloba a todas las clases válidas: 1, 4, 6)
Valores de entrada: 2, 5	(clase 2: 2 datos)
Valores de entrada: 2, 5, 8, 9	(clase 3: 4 datos)
Valores de entrada: 2, 5, 'C'	(clase 5: dato no entero)
Valores de entrada: 2, -2, 5	(clase 7: algún dato <=0)

- No siempre que se meten 3 enteros se puede formar un triángulo (se debe cumplir que a + b > c). Este error no le va a detectar la partición equivalente.
- Para lograr una cobertura del 100% con esta técnica, los casos de prueba deben cubrir todas las particiones identificadas (incluidas las particiones no válidas) utilizando, como mínimo, un valor de cada partición. La cobertura se mide como el número de particiones de equivalencia probadas dividido por el número total de particiones de equivalencia identificadas, normalmente expresado como un porcentaje.

Técnica de Análisis de Valores Límite

- Las condiciones límites son aquellas situaciones que se dan cuando se introducen valores que están justo en el límite de las clases de equivalencia de entrada y de salida. Está comprobado que los errores se acumulan en torno a los límites del dominio de un dato, en mayor medida que para valores intermedios.
- Esta técnica conduce a la elección de casos de prueba en los bordes de la clase correspondiente.
- Las diferencias con el método de la partición equivalente estriban en que no se selecciona un elemento representativo de una clase de equivalencia sino uno o más elementos tal que se prueben los extremos de esa clase de equivalencia, prestando atención no sólo a las condiciones de entrada (espacio de entradas) sino también al espacio de resultados.
- La **ventaja** de este método es que la probabilidad de detectar errores es mayor que con el método de la partición equivalente.

Estrategia

- La **estrategia** será:
  - Si una condición de entrada es un rango comprendido entre dos valores definidos (máximo y mínimo), definirás casos de prueba con ambos valores, y también para valores justo por encima y justo por debajo del máximo y mínimo respectivamente.
  - Si una condición de entrada es un número de valores, crearás casos de prueba para el valor máximo y mínimo (sí es que se conocen), y para los valores por encima y por debajo de ambos.
  - Debes usar ambas reglas para cada condición de salida. Es importante que examines los límites del resultado pues no siempre los límites de entrada coinciden con los límites de salida.
- Algunas variantes de esta técnica identifican tres valores por frontera: los valores antes, en y justo después de la frontera

Datos de salida

- Esta técnica también la empleas para los **datos de salida**:
  - Si la entrada y/o salida es un archivo o conjunto ordenado (fichero secuencial, tabla...), tienes que centrar la atención en el primer y último elemento del conjunto.
  - Si la estructura de datos interna tienen límites preestablecidos (una tabla de x elementos), debes asegurarte de diseñar un caso de prueba que ejercite la estructura de datos en sus límites.
- Esta técnica es uno de los mejores métodos para derivar casos de prueba, si se utiliza correctamente. A veces la determinación de valores límite es muy delicada y requiere una minuciosa elaboración.

Prueba por pares

- Las pruebas por pares se basan en el fenómeno de que la mayoría de los fallos son consecuencia de un factor en particular o de la combinación de 2 factores. El número de fallos causadas por una combinación específica de más de 2 factores se vuelve exponencialmente menor.
- El objetivo de la prueba por pares es probar todas las posibilidades de cualquier combinación de 2 factores. En lugar de probar todas las combinaciones posibles de todos los factores, es mucho mas eficaz si se prueba solo cada combinación de 2 factores. Esto ofrece una enorme reducción en la cantidad de casos de prueba requeridos y aún así proporciona un buen resultado en la detección de fallos. La investigación científica ha demostrado que las pruebas por pares encontrarían más del 97% de las fallos.
- Si hay un par de parámetros con un par clases de equivalencia, los casos de prueba necesarios para las pruebas por pares se pueden derivar manualmente. Sin embargo, en la mayoría de situaciones en la práctica, hay más parámetros y clases de equivalencia y no es posible hacerlo manualmente. En ese caso, hay dos formas de solucionar esto:
  - Aplicando matrices ortogonales
  - Usando herramientas (http://pairwise.org/).

Ejemplo

- Vamos a aplicar el método de análisis de valores límite al caso del triángulo ya resuelto mediante partición equivalente.
- Además de reflejar en la tabla las condiciones de entrada, debemos considerar las condiciones de salida.
- En segundo lugar, cuando tomemos los casos de prueba debemos recordar que elegiremos los valores extremos de las clases de equivalencia obtenidas.

Tabla las condiciones de entrada

Condiciones de Entrada	Clases de equivalencia válidas	Clases de equivalencia inválidas
El número de valores es	(1) 3	(2) 2 (3) 4
Los valores son	(4) enteros	(5) cualquier otro dato
Valores mayores de 0	(6) todos >0	(7) alguno <=0
Número de resultados	(8) 1	(9) <1 (10) >1
Tipo salida	(11) cadena	(12) cualquier otro
Relación entre longitudes	(13) a+b>c	(14) a+b<=c
Cuando a=b=c, devuelve	(15) Equilátero	(16) cualquier otro
Cuando dos lados son iguales, devuelve	(17) Isósceles	(18) cualquier otro
Cuando los tres son distintos, devuelve	(19) Escaleno	(20) cualquier otro

Casos de prueba resultantes

Caso	Valores de entrada	Valores de salida	Clases de equivalencia contempladas
1	1, 1, 1	Equilátero	1, 4, 6, 8, 11, 13, 15 (solo faltan 17 y 19)
2	1, 2, 2	Isósceles	1, 4, 6, 8, 11, 13, 17
3	2, 3, 4	Escaleno	1, 4, 6, 8, 11, 13, 19
4	1, 1	?	2
5	1, 1, 1, 1	?	3
6	2, 5, 'C'	?	5
7	2, 2, 5	No triángulo	7
8	1, 1, 1	(salida vacía)	9
9	1, 1, 1	Equil. + Isósc.	10
10	1, 1, 1	1	12
11	1, 2, 3	No triángulo	14
12	1, 1, 1	Isósceles	16
13	1, 2, 2	Escaleno	18
14	2, 3, 4	Equilátero	20

Observaciones

- Observa que no es posible incluir todas las clases de equivalencia válidas en un solo caso de prueba.
- No obstante, intentaremos incluir todas las clases de equivalencia válidas en el mínimo número de casos de prueba (y cada clase de equivalencia inválida supondrá 1 o más casos de prueba separados).
- Así mismo, fíjate en que se han elegido los valores 1, 1 y 1 por ser los valores válidos más cercanos al límite.

Consistencia de Datos

- Los procesos utilizan datos que se almacenan y mantienen en entidades que tienen un ciclo de vida que comienza cuando se crea una entidad y finaliza cuando se elimina, en el medio, la entidad se utiliza consultándola o actualizándola. Las reglas de integridad describen las condiciones previas bajo las cuales ciertos procesos CRUD están permitidos o no.
- Se obtiene una visión general del ciclo de vida de los datos o entidades con la ayuda de una matriz CRUD. Se trata de una matriz en la que las entidades se muestran horizontalmente en los ejes y las proceso verticalmente. Las operaciones se expresa con las letras C (reate), R (ead), U (pdate) y D (elite). Si un proceso ejecuta una acción particular en relación con una entidad, esto se muestra en la matriz mediante C, R, U y/o D. Una prueba de consistencia comprueba si las distintas funciones utilizan una entidad de forma coherente, busca si la entidad está siendo corrompida por un proceso de tal manera que los otros procesos ya no pueden utilizarla correctamente. Los casos de prueba se derivan reuniendo un ciclo de vida completo de una entidad. Esto se hace de la siguiente manera:
  - Cada caso de prueba comienza con una "C", seguida de todas las "U" posibles y termina con una "D . Si hay más posibilidades de crear o eliminar una entidad, se diseñan casos de prueba adicionales.
  - Después de cada acción (C, U o D) se realiza una "R" o más, para establecer que la entidad ha sido procesada correctamente y es utilizable para las otras funciones (no ha sido corrompida)
  - Con respecto a la entidad relevante, todos los casos de acciones (C, R, U y D) en todos los procesos deben estar cubiertos por los casos de prueba.

Matriz CRUD

	Entity 1	Entity 2	...	Entity n
Process 1	Q	QCU		
Process 2	QCUD	-		
Process 3	-	QC		
Entity n				

Q: Query, C: Create, U: Update, D: Delete, -: None

Validación de Datos

- La prueba sintáctica junto con la prueba semántica, pertenece a las pruebas de validación, con las que se prueba la validez de los datos de entrada. Esto establece el grado en que el sistema es a prueba de entradas inválidas o "sin sentido" que se ofrecen al sistema intencionalmente o de otra manera. Esta prueba también se utiliza para probar la validez de los datos de salida.
- La base de la prueba consiste en las reglas sintácticas o semántica, que especifican cómo debe cumplir un atributo para ser aceptado como entrada/salida válida por el sistema. Estas reglas en realidad describen el dominio de valor del atributo relevante. Si se ofrece un valor fuera de este dominio para el atributo, el sistema debería interrumpir el procesamiento de forma controlada, normalmente con un mensaje de error.
- Las reglas sintácticas y semánticas pueden establecerse en diversos documentos, pero normalmente se describen en:
  - El 'diccionario de datos' y otros modelos de datos, en los que se describen las características de todos los datos
  - Especificaciones funcionales de la función o pantalla de entrada correspondiente, que contienen los requisitos específicos con respecto a los datos y atributos.

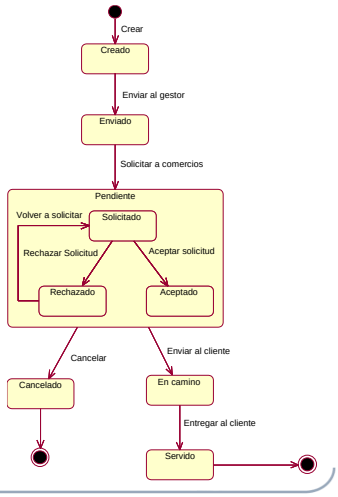
Validación de Datos

- Tipo de datos: numéricos, alfabéticos, alfanuméricos, etc.
- Longitud del campo: La longitud del campo de entrada suele ser limitada. Investiga qué sucede cuando intentas exceder esta longitud.
- Formatos: Para algunos atributos, se establecen requisitos específicos en cuanto a formato (fecha, código postal, teléfono, …)
- De entrada y salida: Hay 3 posibilidades aquí:
  - E: No se muestra ningún valor, pero puede o debe introducirse
  - S: El valor se muestra, pero no se puede cambiar.
  - E/S: Se muestra un valor y se puede cambiar.
- Por defecto
  - Si el atributo no se completa, el sistema debería procesar el valor predeterminado.
  - Si se trata de un campo de E/S, se debe mostrar el valor predeterminado.
- Obligatorio / No obligatorio
  - Un atributo obligatorio no puede permanecer vacío.
  - Un atributo no obligatorio puede permanecer vacío. En el procesamiento, el dato se deja vacío o se utiliza el valor predeterminado para este dato.
- Mecanismo de selección: Se debe elegir entre un número de posibilidades dadas. Es importante aquí si se puede elegir sólo una posibilidad o varias. Este es particularmente el caso de las GUI (interfaz gráfica de usuario), por ejemplo con:
  - Botones de radio (intenta activar varios), Marque casillas (intente activar varias), Cuadro desplegable (intente cambiar el valor o dejarlo vacío).
- Dominio: describe todos los valores válidos para este atributo. En principio, se puede mostrar de dos maneras:
  - Enumeración o lista de valores: Por ejemplo {M, F, N}.
  - Rango de valores: Todos los valores entre los límites dados están permitidos. En particular, deberían ponerse a prueba los propios límites de valores. Por ejemplo, [0, 100>, donde los símbolos indican que el rango de valores es de 0 a 100, incluido el valor 0, pero excluyendo el valor 100.

Los casos de prueba se derivan de las reglas sintácticas y semánticas de forma similar a las particiones de equivalencia.

Técnica de Transición de Estado

- Algunos elementos del sistema pueden tener estado y dichos estados pueden estar tipados: conjunto de valores predefinidos. El cambio de un valor del estado a otro valor de estado se denomina transición de estado y es desencadenado por una acción o evento. No todas las transiciones son posibles: combinaciones de valor inicial con valor final.
- Los casos de pruebas pueden ser diseñados para cubrir una secuencia de estados típica, para practicar todos los estados, para practicar cada transición, para practicar secuencias específicas de transiciones, o para probar transiciones inválidas.
- Los diagramas de estado representan todos los posibles estados de un elemento del sistema y sus correspondientes transiciones de estado. Una tabla de transición de estado muestra todas las transiciones válidas y las transiciones potencialmente inválidas entre estados, así como los eventos, las condiciones de guarda y las acciones resultantes para las transiciones válidas.



Pruebas de Caja Blanca

- La prueba de caja blanca es una técnica de diseño de casos de prueba que usa la estructura de control del diseño para descubrir los casos de prueba.
- Mediante los métodos de prueba de caja blanca, puedes obtener casos de prueba que:
  - Garanticen que se ejercitan por lo menos una vez todos los caminos independientes de cada módulo.
  - Ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa.
  - Ejecuten todos los bucles en sus límites y con sus límites operacionales.
  - Ejerciten las estructuras internas de datos para asegurar su validez.

- Las pruebas de caja blanca **no garantizan** que:
  - El programa cumpla con su especificación.
  - El programa esté completo. Un programa puede estar incompleto por falta de estructuras obviadas por el programador al generar el código.
  - No detectan problemas de dominio de datos.

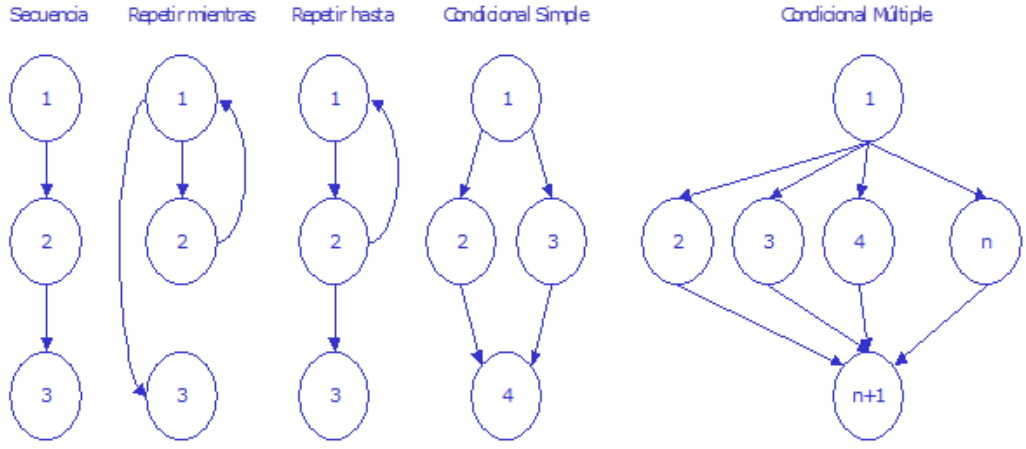
Pruebas de Caja Blanca

- Las pruebas de caja blanca se pueden aplicar a cualquier nivel donde se conozca la estructura:
  - Pruebas unitarias: la estructura del código, es decir, sentencias, decisiones, bucles y caminos distintos.
  - Pruebas de integración: la estructura de llamadas, secuencias o colaboración, o la transición de estados.
  - Pruebas de sistema: la estructura puede ser el árbol de navegación, los procesos de negocio o la estructura del sitio web.
- La pruebas de caja blanca permiten establecer la métrica de cobertura de código que indica el porcentaje de instrucciones que han participado en la prueba o, coloquialmente, que se han probado.

Técnica del Camino Básico

- El método del camino básico te permite obtener una medida de la complejidad lógica de un diseño y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución.
- Los casos de prueba derivados del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa. Los pasos que debes dar son:
  1. Numeras las sentencias del diseño o código, que aparecerán en los correspondientes nodos del grafo de flujo.
  2. Para numerar se escogen: alternativas, repetitivas, secuencias resultado de alternativas, conjuntos de sentencias útiles.
  3. Dibujas el grafo de flujo, partiendo del diseño o del código del procedimiento.
  4. Obtienes la complejidad ciclomática del grafo de flujo.
  5. Fijas un conjunto básico de caminos linealmente independientes.
  6. Determinar los casos de prueba que permitan la ejecución de cada camino del conjunto anterior.
- Para construir el **grafo de flujo**, hay que transformar las construcciones del diseño o de la programación en los subgrafos de flujo adecuados.

Sub grafos de flujo



Complejidad Ciclomática

- El círculo, denominado **nodo**, representa una o más sentencias. Las flechas, denominadas **aristas**, representan los flujos de control. Las áreas delimitadas por aristas se denominan **regiones**. El área más externa dentro del grafo también es una región.
- El número de caminos independientes del conjunto básico de un programa o una realización de caso de uso del diseño se puede calcular a partir de la **Complejidad Ciclomática** basada en la teoría de grafos, y nos da el número mínimo de casos de prueba que debemos diseñar para asegurar que se ejecuta cada sentencia al menos una vez:

Complejidad Ciclomática = Número de aristas - Número nodos + 2

Análisis de Decisiones

- Cuando un programa contiene alternativas dentro de la estructura de diseño, nos obliga a generar datos de prueba que aseguren que se ejecuta cada rama lógica del programa.
- Las alternativas pueden tener una o más condiciones, la estrategia para cada caso es diferente.
- Este análisis es complementario a la prueba de los caminos básicos.
- Cuando la alternativa tiene una sola condición el criterio de pruebas mínimo es generar datos de prueba que cubran:
  - Todos los valores de cada decisión al menos una vez.
  - Llamadas a cada punto de entrada, al menos una vez.
- Para programas con decisiones de tipo multicondición, el criterio a aplicar es el de establecer un número de casos de prueba suficiente para cubrir:
  - Todas las combinaciones posibles de todos los valores de cada condición.
  - Llamadas a cada punto de entrada, al menos una vez.

Tablas de decisión

- Las tablas de decisión representan relaciones lógicas entre las condiciones (entradas) y las acciones (salidas). Los casos de prueba son derivados sistemáticamente considerando cada combinación posible de condiciones y de acciones.
- Son una buena manera de capturar los requerimientos del sistema que contienen condiciones lógicas.
- Pueden ser utilizadas para registrar reglas de negocio complejas de un sistema.
  - Se analiza la especificación, y las condiciones y las acciones del sistema se identifican.
  - Las condiciones y las acciones de la entrada se indican de una manera tal que puedan ser verdaderas o falsas (booleano).
  - Cada columna de la tabla corresponde a una regla de negocio que define una combinación única de las condiciones que dan lugar a la ejecución de las acciones asociadas a esa regla.
  - La idea es tener por lo menos una prueba por columna, que implica cubrir todas las combinaciones para accionar condiciones.
- La cobertura estándar mínima habitual para la prueba de tabla de decisión es tener al menos un caso de prueba por regla de decisión en la tabla. Esto implica, normalmente, cubrir todas las combinaciones de condiciones. La cobertura se mide como el número de reglas de decisión probadas dividido por el número total de reglas de decisión, normalmente expresado como un porcentaje.

Tablas de decisión

		R1	R2	R3	R4	R5	R6	R7	R8	RR1
C1	Tarjeta Seguridad Social	SI	SI	SI	SI	NO	NO	NO	NO	SI
C2	Tarjeta Sociedad Medica	SI	SI	NO	NO	SI	SI	NO	NO	-
C3	Tarjeta Fidelidad	SI	NO	SI	NO	SI	NO	SI	NO	-
A1	Sin Pago	X	X	X	X	X				X
A2	Pago franquicia						X		X	
A3	Pago servicios							X	X	

- La fortaleza de las tablas de decisión es crear combinaciones de condiciones que pueden no ejercitarse de otra manera.
- Puede ser aplicada cuando la acción del software depende de varias decisiones lógicas.
- La simplificación de las tablas de decisión o reducción, optimiza las reglas de un programa, eliminando inconsistencias y redundancias. Para simplificar una tabla de decisión se identifican las reglas puras con las mismas acciones, se analiza sus condiciones para eliminar las irrelevantes (SI y NO) y se reducen las reglas puras a una regla mixta donde las condiciones irrelevantes se marcan con un - o una i.

Prueba de Bucles

- El método se centra en la validez de las construcciones de bucles. Debes intentar descubrir errores de inicialización, errores de indexación o de incremento y errores en los límites de los bucles.
- Este tipo especial de pruebas es complementario a la prueba de los caminos básicos.
- Podemos definir **tres tipos de bucles**:
  - Simples
  - Anidados
  - Concatenados

Prueba de Bucles

- A los **bucles simples** se les debe aplicar el siguiente conjunto de pruebas, donde n es el número máximo de pasos permitidos por el bucle:
  1. Saltar totalmente el bucle.
  2. Hacer n-1 pasos por el bucle.
  3. Hacer n pasos por el bucle.
  4. Hacer n+1 pasos por el bucle.
- Si extendiéramos el enfoque de prueba de los bucles simples a los **bucles anidados**, el número de posibles pruebas crecería geométricamente a medida que aumentara el nivel de anidamiento. Esto nos llevaría a un número impracticable de pruebas.

Prueba de Bucles

- Un enfoque para **reducir el número de pruebas** en este caso es:
  1. Comenzar en el bucle más interno. Disponer todos los demás bucles en sus valores mínimos.
  2. Llevar a cabo las pruebas de bucles simples con el bucle más interno mientras se mantienen los bucles externos con los valores mínimos para sus parámetros de iteración (por ejemplo contadores de bucle). Añadir otras pruebas para los valores fuera de rango o para valores excluidos.
  3. Progresar hacia fuera, llevando a cabo las pruebas para el siguiente bucle, pero manteniendo todos los demás bucles externos en sus valores mínimos y los demás bucles anidados con sus valores típicos.
  4. Continuar hasta que se hayan probado todos los bucles.
- Los **bucles concatenados** se pueden probar mediante el enfoque anteriormente definido para los bucles simples, mientras que cada uno de los bucles sea independiente del resto. Cuando los bucles no son independientes, se recomienda usar el enfoque aplicado para los bucles anidados.

Combinando las técnicas de caja blanca

- En el siguiente ejemplo vamos a estudiar un módulo, donde combinaremos las diferentes técnicas de caja blanca.
- El módulo lee como máximo 100 números de una tabla y calcula su media siempre y cuando los valores de los números estén entre 3 y 10, ignorándose el resto.
- Si no existe ningún número válido para la media, el módulo devuelve –1 como valor de la media.
- El final de la tabla se marca con el valor –1.

Pseudocódigo del módulo

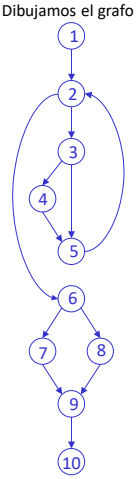
```
inicio calcular_media (entrada: tabla, salida: media)
        contador = 0
        suma = 0
        indice = 0
        mientras ((tabla[indice] <> -1) y (indice < 100))
                si ((tabla[indice] >= 3) y (tabla[indice] <= 10))
                        suma = suma + tabla[indice]
                        contador = contador + 1
                fin si
                indice = indice + 1
        fin mientras
        si (contador > 0)
                media = suma / contador
        si_no
                media = -1
        fin si
fin calcular_media
```


Generar el grafo

- Empezamos identificando y numerando los nodos.

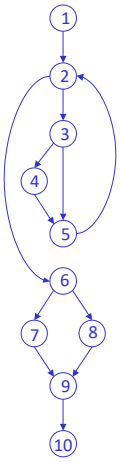
inicio calcular_media (entrada: tabla, salida: media)	
contador = 0 suma = 0 indice = 0	1
mientras ((tabla[indice] <> -1) y (indice < 100))	2
si ((tabla[indice] >= 3) y (tabla[indice] <= 10))	3
suma = suma + tabla[indice] contador = contador + 1	4
fin si indice = indice + 1 fin mientras	5
si (contador > 0)	6
media = suma / contador	7
si_no media = -1	8
fin si	9
fin calcular_media	10

Dibujamos el grafo



Complejidad Ciclomática y caminos independientes

- Contamos las aristas y los nodos para calcular la Complejidad Ciclomática.
- **Aplicamos la fórmula**:
  Complejidad Ciclomática = 12 aristas - 10 nodos + 2 = 4
- Obtenemos que tenemos 4 caminos y **pasamos a fijar el conjunto básico de caminos linealmente independientes:**
  CB1: 1 → 2 → 6 → 7 → 9 → 10
  CB2: 1 → 2 → 6 → 8 → 9 → 10
  CB3: 1 → 2 → 3 → 5 → 2 → 6 → (...resto del CB1 ó CB2...) → 8 → 9 → 10
  CB4: 1 → 2 → 3 → 4 → 5 → 2 → 6 → (...resto del CB1 ó CB2...) → 7 → 9 → 10



Pruebas de bucles

- **Identificamos las pruebas de bucles**:
  B1: saltar el bucle; la tabla vacía, la primera posición con valor –1
  B2: ejecutarlo 99 veces; la tabla con 99 números, la posición 99 con valor -1
  B3: ejecutarlo 100 veces; la tabla con 100 números, la posición 100 con valor –1
  B4: ejecutarlo 101 veces; la tabla con 101 números, la posición 101 con valor –1

Prueba de las condiciones

Nodo 2		R1	R2	R3	R4
C1	tabla[indice] <> -1	V	V	F	F
C2	indice < 100	V	F	V	F
A1	Permanencia en bucle	X			

Nodo 3		R5	R6	R7
C3	tabla[indice] >= 3	V	V	F
C4	tabla[indice] <= 10	V	F	-
A2	Acumular y contar	X		

Nodo 6		R8	R9
C5	contador > 0	V	F
A3	Calcular media	X	
A4	Devolver -1		X

Casos de prueba

- Una vez tengamos fijados los caminos, los bucles y las reglas, determinamos los casos de prueba que permitan la ejecución de cada camino del conjunto anterior.
- Caso de prueba 1:
  - Entrada:
    - La tabla vacía:

0	1	2	...
-1			

  - Salida:
    - Media = -1
  - Objetivo:
    - **CB2**, **B1**, **R3**, **R8**

Casos de prueba

- **Caso de prueba 2:**
  - Entrada:
    - La tabla con 99 números y todos están entre 3 y 10

0	1	2	3	4	5	...	98	99	100	101
2	11	13	-2	0	1	1	1	1	-1	

  - Salida:
    - Media = 5
  - Objetivo:
    - **CB1**, **CB4**, **B2**, **R1**, R3, **R6**, **R9**

Casos de prueba

- **Caso de prueba 3:**
  - Entrada:
    - La tabla con 100 números y ninguno está entre 3 y 10

0	1	2	3	4	5	...	98	99	100	101
2	11	13	-2	0	1	1	1	1	-1	

  - Salida:
    - Media = -1
  - Objetivo:
    - **CB3**, **B3**, R1, **R4**, **R5**, **R7**, R8

Casos de prueba

- **Caso de prueba 4:**
  - Entrada:
    - La tabla con 101 números surtidos

0	1	2	3	4	5	...	98	99	100	101
3	7	5	5	1	11	11	10	21	1	-1

  - Salida:
    - Media = 6
  - Objetivo:
    - **B4**, R1, **R2**, R5, R6, R7, R9

Cobertura

- Las pruebas de sentencia ejercitan las sentencias ejecutables en el código. La cobertura se mide como el número de sentencias ejecutadas por las pruebas dividido por el número total de sentencias ejecutables en el objeto de prueba, normalmente expresado como un porcentaje.
- La pruebas de decisión ejercitan las condiciones en el código y ejercitan el código que se ejecuta basado en los resultados de la decisión. Para ello, los casos de prueba siguen los flujos de control que se producen desde un punto de decisión (por ejemplo, para una declaración IF, uno para el resultado verdadero y otro para el resultado falso; para una declaración CASE, se necesitarían casos de prueba para todos los resultados posibles, incluido el resultado por defecto). La cobertura se mide como el número de resultados de decisión ejecutados por las pruebas dividido por el número total de resultados de decisión en el objeto de prueba, normalmente expresado como un porcentaje.
- Cuando se logra una cobertura del 100% de sentencia, se asegura de que todas las sentencias ejecutables del código se han probado al menos una vez, pero no asegura que se haya probado toda la lógica de decisión. Cuando se alcanza el 100% de cobertura de decisión, se ejecutan todos los resultados de decisión, lo que incluye probar el resultado verdadero y también el resultado falso, incluso cuando no hay una sentencia falsa explícita (por ejemplo, en el caso de una sentencia IF sin un ELSE en el código). La cobertura de sentencia ayuda a encontrar defectos en el código que no fueron practicados por otras pruebas. La cobertura de decisión ayuda a encontrar defectos en el código donde otras pruebas no han tenido ambos resultados, verdadero y falso.
- Lograr una cobertura del 100% de decisión garantiza una cobertura del 100% de sentencia (pero no al revés).

Pruebas de presentación

- A diferencia de los casos anteriores, las pruebas de interfaces de usuario con el diseño de entrada (pantallas) y salida (listas, informes) se basan en la experiencia para descubrir los casos de prueba.
- Las reglas de diseño pueden establecerse en varios documentos, pero generalmente se describen en:
  - Guías de estilo, que muchas veces contienen pautas o reglas para toda la organización, relativas a cuestiones como el uso del color, las fuentes, la disposición de la pantalla, etc.
  - Especificaciones para el diseño de la lista o pantalla correspondiente.
- La técnica se basa en una serie de cuestionarios que contienen las situaciones más habituales de error. La identificación de los casos de prueba la realizas recorriendo cada una de las cuestiones de cada lista, estudiando si es o no es aplicable. En caso de ser aplicable necesitas diseñar un caso de prueba.
- Las listas que te proponemos son más o menos estándar. No son cerradas. Debes irlas completando sobre la base de tu experiencia.

Ventanas

- ❑ ¿Se abrirán las ventanas basándose en órdenes basadas en el teclado o en un menú?
- ❑ ¿Se puede ajustar el tamaño, mover y desplegar la ventana?
- ❑ ¿Está todo el contenido de la información dentro de la ventana accesible adecuadamente con el ratón, teclas de función, flechas de dirección y teclado?
- ❑ ¿Se genera adecuadamente cuando se sobrescribe y se vuelve a abrir?
- ❑ ¿Están operativas todas las funciones relacionadas con la ventana?
- ❑ ¿Están disponibles y desplegados apropiadamente en la ventana todos los menús emergentes, barras de herramientas, barras deslizantes, cuadros de diálogo, botones, iconos y otros controles importantes?
- ❑ Cuándo se despliegan varias ventanas, ¿se representa adecuadamente el nombre de cada ventana?
- ❑ ¿Está resaltada adecuadamente la ventana activa?
- ❑ Si se utiliza multitarea, ¿están actualizadas todas las ventanas en los momentos adecuados?
- ❑ ¿Causan las selecciones múltiples o incorrectas del ratón dentro de la ventana efectos secundarios inesperados?
- ❑ ¿Están de acuerdo con las especificaciones los indicadores de audio y/o de color de la ventana o como consecuencia de operaciones de la ventana?
- ❑ ¿Se cierra adecuadamente la ventana?

Menús emergentes y operaciones con el ratón

- ❑ ¿Se muestra la barra de menú apropiada en el contexto apropiado?
- ❑ ¿Despliega la barra de menú de la aplicación características relacionadas con el sistema (por ejemplo: la pantalla de un reloj)?
- ❑ ¿Funcionan adecuadamente las operaciones de despliegue?
- ❑ ¿Funcionan adecuadamente los menús de escape, paletas y barras de herramientas?
- ❑ ¿Están listadas adecuadamente todas las funciones del menú y subfunciones emergentes?
- ❑ ¿Son todas las funciones del menú accesibles con el ratón?
- ❑ ¿Es correcto el tipo, tamaño y formato de texto?
- ❑ ¿Es posible invocar todas las funciones del menú usando su orden alternativa de texto?

Menús emergentes y operaciones con el ratón (cont.)

- ❑ ¿Están resaltadas las funciones del menú (o difuminadas) dependiendo del contexto de las operaciones actuales de la ventana?
- ❑ ¿Se ejecutan todas las funciones de cada menú como se anunciaba?
- ❑ ¿Son suficientemente claros los nombres de las funciones del menú?
- ❑ ¿Hay ayuda disponible para cada elemento del menú y es sensible al contexto?
- ❑ ¿Se reconocen apropiadamente las operaciones del ratón a lo largo de todo el contexto interactivo?
- ❑ Si se necesitan múltiples clic, ¿están apropiadamente reconocidos en el contexto?
- ❑ Si el ratón tiene varios botones, ¿son reconocidos apropiadamente en el contexto?
- ❑ ¿Cambian adecuadamente el cursor, el indicador de procesamiento y el puntero al invocar diferentes operaciones?

Entrada de datos

- ¿Se repiten y son introducidos adecuadamente los datos alfanuméricos en el sistema?
- ¿Funcionan adecuadamente los modos gráficos de entrada de datos (por ejemplo: una barra deslizante)?
- ¿Se reconocen adecuadamente los datos no válidos?
- ¿Son inteligibles los mensajes de entrada de datos?

Documentación y ayuda

- ¿Describe con exactitud la documentación cómo conseguir cada modo de empleo?
- ¿Es exacta la descripción de cada secuencia de interacción?
- ¿Son exactos los ejemplos?
- ¿Son consistentes con el programa real la terminología, las descripciones del menú y las respuestas del sistema?
- ¿Es relativamente fácil localizar ayuda en la documentación?
- ¿Se pueden solucionar problemas fácilmente con la documentación?
- ¿Son exactos y completos la tabla de contenido y el índice?
- ¿Facilita el diseño del documento (distribución, tipos de letra, indentación, grafos) la comprensión y rápida asimilación de la información?
- ¿Están descritos con gran detalle los mensajes de error para el usuario en el documento?
- Si se utilizan enlaces de hipertexto, ¿son exactos y completos?

Páginas Web

- ❑ La navegación
- ❑ Títulos, encabezados y pies de página (corta y pega)
- ❑ Las interacciones con la página y sus elementos
- ❑ El rellenado de formularios y sus validaciones
- ❑ El arrastrar y soltar, si procede
- ❑ La estética, presencia y visualización de elementos esenciales, con especial atención al Responsive Design
- ❑ La accesibilidad, en caso de estar legalmente obligados
- ❑ Moverse entre ventanas y marcos (aunque están prohibidos por la WAI)
- ❑ Uso de cookies, Local Storage, Session Storage, Service Worker, …

Herramientas de Pruebas de GUI

- Hay herramientas disponibles, especialmente para aplicaciones web, que pueden realizar todo tipo de comprobaciones. Ejemplos de estos controles son:
  - ¿Los gráficos y animaciones cuentan con una alternativa (un cuadro de texto) para proporcionar la misma información en caso de que los gráficos, animaciones, etc. no funcionen? Este puede ser el caso si utilizas un navegador diferente, no tienes una tarjeta de vídeo o tienes una discapacidad visual.
- ¿El tamaño de los gráficos es demasiado grande, lo que hace que el sitio sea lento?
- ¿Cada página contiene un enlace para regresar a la página anterior y/o un enlace para continuar a la página siguiente?
- ¿Quizás los cuadros de texto son demasiado largos en un campo de desplazamiento?
- ¿Son todos los hipervínculos (todavía) válidos?

Técnicas basadas en la experiencia

- Hasta ahora hemos visto técnicas formales, basadas en pruebas sistemáticas, pero existe un conjunto de pruebas basadas en la experiencia que derivan de la habilidad e intuición del probador y de su experiencia con aplicaciones y tecnologías similares.
- Los casos de prueba están basados en la intuición y experiencia:
  - ¿Dónde se han acumulado errores en el pasado?
  - ¿Dónde falla normalmente el software?
- Se trata de un enfoque especialmente útil en los casos en los que las especificaciones son escasas o inadecuadas y existe una importante presión temporal.
- Pueden complementar a las técnicas sistemáticas e identificar pruebas especiales que no pueden capturarse fácilmente mediante técnicas formales, aunque el grado de efectividad esta directamente relacionado con la experiencia del probador.
- Asimismo, puede servir como comprobación del proceso de pruebas, para ayudar a garantizar que los defectos más graves han sido efectivamente detectados.
- Casi cualquier proceso de pruebas suele comenzar con prueba ad hoc y pruebas exploratorias basadas en la experiencia.

Diseño de casos de prueba.

- Las pruebas basadas en la experiencia también se denomina pruebas intuitivas (intuitive testing) e incluyen predicciones de errores (error guessing - pruebas orientadas a puntos débiles) y pruebas exploratorias (pruebas iterativas basadas en el conocimiento adquiridos respecto del sistema).
- **Intuición:**
  - ¿Dónde se pueden esconder los defectos? La intuición caracteriza a un buen probador.
- **Experiencia:**
  - ¿Qué **defectos** han sido **detectados** en el pasado y dónde?
- **Conocimiento / percepción:** Se incluyen **detalles** específicos **del proyecto**.
  - ¿Dónde se **esperan defectos** específicos?
  - ¿Dónde se producirán defectos como consecuencia de la **presión por los plazos** y la **complejidad**?
  - ¿Están involucrados programadores sin experiencia?

Diseño intuitivo de casos de prueba

- Las posibles fuentes del diseño intuitivo de casos de prueba son:
  - Resultados de pruebas y experiencia práctica con sistemas similares.
    - Posiblemente con el sistema actual a sustituir por el nuevo sistema u otro sistema con funcionalidad similar.
  - Experiencia de usuario.
    - Aplicar la experiencia con el sistema como un usuario del mismo.
  - Enfoque del despliegue.
    - ¿Qué partes del sistema serán utilizados con mayor frecuencia?
  - Problemas de desarrollo.
    - ¿Hay algún punto débil como resultado de dificultades en el proceso de desarrollo?

Predicción de errores

- La fuente de la predicción de errores es la lista de defectos comunes que has ido elaborándote en la prueba de otros sistemas anteriores.
- Los pasos a dar para la creación de los casos de prueba serian:
  - Comprobar lista de defectos.
    - Enumerar posibles errores.
    - Ponderar factores dependientes del riesgo y probabilidad de ocurrencia.
  - Diseño de caso de prueba.
    - Creación de casos de prueba dirigidos a producir los defectos de la lista.
    - Aumentar la prioridad a aquellos casos de prueba considerando el valor de su riesgo.
  - Actualizar la lista de defectos durante las pruebas.
    - Es un procedimiento iterativo incremental.
    - Es útil para cuando se repita el procedimiento en futuros proyectos.

Pruebas exploratorias

- Es un procedimiento de diseño de casos de prueba especialmente apropiado cuando la información base se encuentra poco estructurada o cuando el tiempo disponible para pruebas es escaso.
- Procedimiento:
  - Revisar las partes constituyentes (individuales/identificables) del objeto de prueba.
  - Ejecutar un número reducido de casos de prueba exclusivamente sobre aquellas partes que deben ser probadas, aplicando predicción de errores ("errores guessing").
  - Analizar los resultados, desarrollar un modelo preliminar ("rough model") de cómo funciona el objeto de prueba.
  - Por iteración:
    - Diseñar nuevos casos de prueba aplicando el conocimiento adquirido recientemente.
    - Por lo tanto concentrándose en las áreas relevantes y explorando características adicionales del objeto de prueba.
- Es recomendable utilizar herramientas de captura, que son útiles para registrar las actividades de pruebas, y seleccionar objetos pequeños y/o concentrándose en aspectos particulares del objeto de pruebas, una iteración unitaria no debería llevar más de un par de horas.
- Los resultados de una iteración constituyen la base de información para la siguiente iteración, un conocimiento que permite la elección apropiada del métodos de diseño de casos de prueba.

Pruebas exploratorias

- Las pruebas exploratorias están estructuradas en tres partes claras:
  - Preparación: un documento en papel o electrónico que contiene información para usar durante la sesión de prueba exploratoria.
    - Determinar el alcance: pueden ser las historias de usuario.
    - Establecer un horario: se especifica la duración de la prueba, cuando se acabe el tiempo, finalizará la prueba.
    - Determinar el equipo de prueba: se realizan en parejas o en grupos más grandes: dos personas conocen a más de una y ven a más de una.
    - Ideas de prueba: son cualquier pensamiento, dato, técnica, heurística o cualquier cosa útil para que durante su sesión de prueba exploratoria tenga una gran cantidad de posibilidades para variar sus pruebas.
    - Restricciones: delimita el ámbito de la prueba con lo que probar y lo que no.
  - Ejecutar pruebas y mantener un registro: Durante la prueba se registra cada caso de prueba, junto con los resultados esperados, los resultados reales y las observaciones, para realizar un seguimiento de lo que se probó y los resultados reales que ocurrieron frente a las expectativas que originaron las observaciones y las anomalías, si corresponde.
  - Discutir resultados, para obtener conclusiones y consejos, y comunicarlos a todas las partes interesadas.
- La información y las conclusiones extraídas se conservan como base de conocimiento.

Selección de la técnica de pruebas

- La selección de la técnica de pruebas a utilizar depende de una serie de factores entre los que se incluyen el tipo de sistema, los estándares normativos, los requisitos contractuales o de cliente, el nivel de riesgo, el tipo de riesgo, el objetivo de la prueba, la documentación disponible, el conocimiento de los probadores, el tiempo y el presupuesto, el ciclo de vida de desarrollo, los modelos de caso de uso y la experiencia previa en los tipos de defectos detectados.
- Algunas técnicas resultan más aplicables a ciertas situaciones y niveles de prueba, mientras que otras son aplicables a todos los niveles de pruebas.
- Para crear los casos de prueba, tendrás generalmente que aplicar una combinación de técnicas de pruebas, entre las que se incluyen técnicas guiadas por procesos, reglas y datos, con vistas a garantizar la correcta cobertura del objeto que se está probando.

METODOLOGÍA

Introducción

- La pruebas son costosas. Se necesita una planificación cuidadosa para obtener lo máximo del proceso de prueba y controlar los costes.
- Las pruebas, como parte fundamental del control de calidad, debe estar planificada y gestionada. El propósito del plan de pruebas es explicitar el alcance, enfoque, recursos requeridos, calendario, responsables y manejo de riesgos de un proceso de pruebas.
- Un plan de prueba es un documento que utilizamos para describir la prueba. Puede contener temas sobre "qué necesitamos probar", "quién probará qué y en qué momento", "cómo realizaremos la prueba", "cuánto tiempo llevará la prueba", "cuándo se realizará la prueba", '¿Cómo podemos organizar y gestionar las pruebas?',... y muchos más.
- Utilizamos el plan de prueba para comunicar la planificación de la prueba a las partes interesadas para la prueba. Puede tener muchas formas y tamaños diferentes, siempre que resuelva el problema en cuestión: describir el enfoque de prueba y generar apoyo de las partes interesadas para este enfoque. El objetivo es identificar el mayor número de posibles causas de fallo y probar si se producen errores.
- Puede haber un plan global que explicite el énfasis a realizar sobre los distintos tipos de pruebas (verificación, integración) y tantos planes específicos como sean necesarios.

Diseño de pruebas

- Uno de los objetivos más importantes de las pruebas es dar un asesoramiento claro sobre calidad y riesgo de tal forma que todas las partes implicadas ganen confianza en el producto. Para poder hacer esto, un evaluador debe recopilar información sobre el comportamiento del sistema. Una de las principales herramientas para recopilar información es la ejecución de casos de prueba. Los resultados de esos casos dan información sobre el comportamiento del sistema. Las preguntas principales son:
  - ¿Qué casos de prueba?
  - ¿Cuántos?
  - ¿Y cómo conseguimos esos casos?
- Para responder a esas preguntas el diseño de la prueba es indispensable.
- Diseñar el conjunto correcto de casos de prueba es el vínculo esencial entre la estrategia de prueba y la implementación de la estrategia de prueba: las pruebas que se ejecutan.

Acuerdos de prueba genéricos (GTA)

- Algunas organizaciones, proyectos o equipos utilizan el llamado documento de acuerdos de prueba genéricos (GTA).
- Los acuerdos de prueba genéricos describen el enfoque general para la configuración y organización de procesos de prueba que se aplica a más de un proyecto o versión. Contiene acuerdos generales sobre, por ejemplo, el proceso de prueba, la estrategia estándar, el método de estimación, los procedimientos, la organización, la comunicación, la documentación, etc.
- Los acuerdos de prueba genéricos describen la forma de trabajar (WoW) de un grupo de equipos. De esta manera se promueve una comprensión compartida y una terminología compartida entre los equipos para facilitar la colaboración.
- Los acuerdos de prueba genéricos deben permanecer estables durante un largo período de tiempo y, por lo tanto, deben describirse y almacenarse de manera que las personas involucradas puedan acceder fácilmente a ellos (por ejemplo, en una intranet, en el repositorio o en un documento de fácil acceso).

Estrategia de calidad

- La estrategia de calidad describe la asignación de medidas de calidad a los elementos de entrega de TI (por ejemplo, historias de usuario, características, etc.), para equilibrar la inversión en actividades de ingeniería de calidad y realizar una distribución óptima del esfuerzo sobre las actividades fundamentales y las variedades de prueba. La clase de riesgo de calidad se utiliza para asignar la intensidad de las medidas de calidad que deben aplicarse.
- El beneficio de aplicar una estrategia de ingeniería de calidad es que un equipo (o grupo de equipos colaboradores) crea una guía fácil y clara para todos los involucrados sobre qué actividades aplicar para lograr una calidad integrada en sistemas de TI adecuados para su propósito. Esta guía tendrá un enfoque diferente, algunas medidas de calidad están vinculadas a historias de usuarios individuales, otras medidas de calidad se aplican a características o epopeyas y la estrategia de ingeniería de calidad también contendrá medidas de calidad genéricas que el equipo aplica siempre.

Estrategia de pruebas

- La estrategia de prueba describe la distribución de los recursos de prueba en las distintas piezas y aspectos que se van a probar y tiene como objetivo encontrar los defectos más importantes lo antes posible y al menor coste. ¿Cuánto vamos a probar las diferentes partes y aspectos del sistema?
- La estrategia de prueba refleja los riesgos y oportunidades comerciales y tecnológicas, utilizando el PRA (Análisis de riesgos del producto) como insumo, así como el ciclo de vida del desarrollo de software.
- La estrategia de prueba define las variedades de prueba que se emplearán y cuán exhaustivas, si las hubiera, deben realizarse las pruebas en las variedades de prueba individuales.
- La estrategia de prueba establece el enfoque de prueba de tal manera que todos en un proyecto de prueba puedan entenderlo.
- La estrategia de prueba es un acuerdo entre partes, y el éxito de una estrategia de prueba puede ser percibido y será percibido de manera diferente por estas partes.
  - Desde el punto de vista del cliente, una estrategia de prueba exitosa se traduce en comprender qué riesgos se cubrirán mediante las pruebas y, en qué medida, ¿cuáles son los riesgos residuales?
  - Desde el punto de vista de las pruebas, una estrategia de prueba exitosa no solo aporta esta comprensión, sino que también proporciona una base sólida para seleccionar las variedades de pruebas, los tipos de cobertura adecuados y las técnicas de diseño de pruebas adecuadas.

Tabla de riesgos

- En el póquer de riesgo, el objetivo no es obtener una estimación perfectamente precisa aunque asignar números de riesgo puede dar una falsa sensación de precisión. Por eso muchos equipos trabajan con letras: por ejemplo, una A si el número de riesgo es 9 (alto), una B si es 6 o 4 (medio) y una C si es 1, 2 o 3 (bajo).

Item	Characteristic	Impact	Chance of failure	Risk Class	
US 1	Functionality	3	3	9	A
	Usability	2	1	2	C
US 2	Functionality	2	2	4	B
	Security	3	2	6	B
US 3	Functionality	2	1	2	C
Spike 1	Performance	2	1	2	C
Feature 1	Performance	1	1	1	C
Feature 2	Functionality	2	2	4	B
	Suitability	2	2	4	B
...	...	...	...	...	...

Tabla de estrategias de prueba

- Los símbolos •••, •• y • se utilizan para especificar la intensidad de la prueba. Cuanto más •••, más intensa será la prueba: tres ••• para la A, dos •• para la B y uno • para la C. Se permite desviarse de esta regla, cuando exista una buena razón. Se puede agregar columnas por tipos de prueba.

Item	Characteristic	Risk Class	Static Testing	Dynamic Testing	Other Quality Measures
US 1	Functionality	A	●●	●●●	●●●
	Usability	C	●	●	
US 2	Functionality	B	●	●●	●
	Security	B	●	●●	●
US 3	Functionality	C	●	●	
Spike 1	Performance	C	●	●	
Feature 1	Performance	C	●	●	
Feature 2	Functionality	B	●	●●	●
	Suitability	B	●	●●	●
...	...	...	...	...	...

Matriz de intensidad

- Se debe disponer de una matriz de intensidad de prueba que brinde sus opciones iniciales para implementar pruebas de intensidad baja, media y alta.
- Esta matriz contiene, para cada grupo de cobertura, las técnicas de diseño de pruebas para lograr la intensidad: baja, media y alta.
- Por supuesto, también se pueden utilizar otras medidas de calidad para alcanzar la intensidad de la prueba, especialmente en el caso de algunas características de calidad específicas.
- El contenido de dicha matriz normalmente será estable durante un largo período de tiempo pero, por supuesto, cuando surgen nuevas situaciones, la matriz debe ajustarse si es necesario.

Group of test design techniques	Intensity: • (low)	Intensity: •• (medium)	Intensity: ••• (high)
Process	Happy path State Testing 0-switch	Process Cycle Tst TM-1 State Testing 1-switch	Process Cycle Tst TM-2 State Testing 2-switch + Eploratory Testing
Conditions	Elementary Comparison Test with CDC	Elementary Comparison Test with MCDC	MCC (full Decision Tbl) Or Elementary Compari-son Test with MCC
Data	Equivalence Partit. DCoT – class coverage	Boundary Value An. DCoT – pairwise	Equiv. Part. + Bound. V. DCoT –triplewise
Appearance	Exploratory Testing 1 profile	Syntax Testing Few profiles	Syntax Testing + Expl. T. Many profiles

Table: example of test intensity table

Tabla de intensidad de prueba

- La combinación de las variedades de pruebas, los tipos de cobertura adecuados y las técnicas de diseño de pruebas permiten seleccionar las estrategias mas adecuadas.

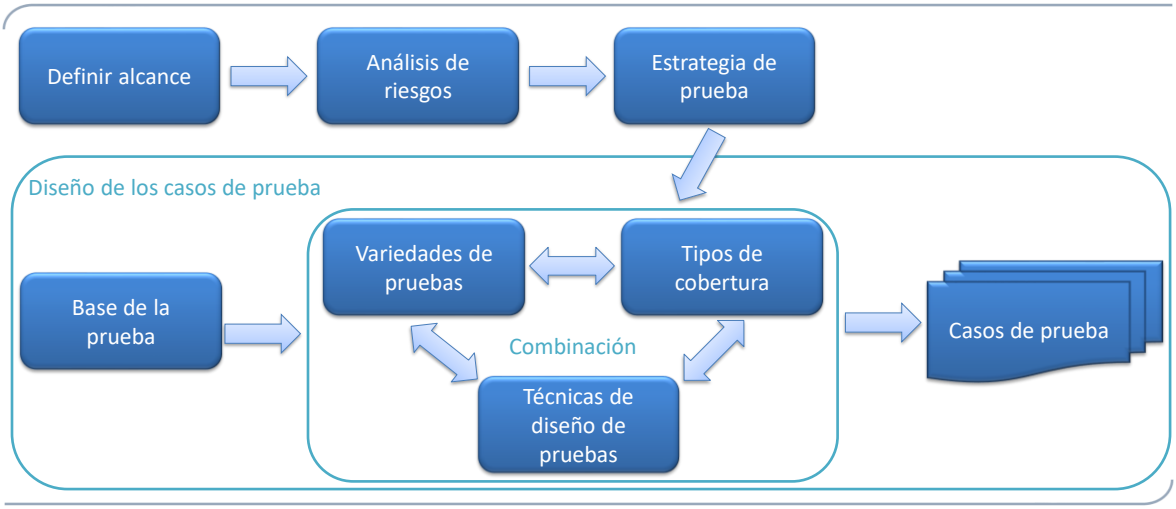
Item	Characteristic	Risk Class	Static Testing	Dynamic Testing	Other Quality Measures
US 1	Functionality	A	INVEST, Inspection	PCT-TDL3	TDD, BDD
	Usability	C	INVEST	A/B Testing	
US 2	Functionality	B	INVEST	DCoT-PAIRWISE	TDD
	Security	B	INVEST	Penetration Test	TDD
US 3	Functionality	C	INVEST	DCoT-EQ	
Spike 1	Performance	C	Technical Review	Algorithm Test	
Feature 1	Performance	C	Technical Review	Algorithm Test	
Feature 2	Functionality	B	INVEST	DCoT-PAIRWISE	TDD
	Suitability	B	INVEST	PCT-TDL2	TDD
...	...	...	...		

Tabla de intensidad de prueba

- La combinación de las variedades de pruebas, los tipos de cobertura adecuados y las técnicas de diseño de pruebas permiten seleccionar las estrategias mas adecuadas.

TMAP: Quality Engineering Strategy Table EXAMPLE				Quality Measures and their intensity (more focus on preventive measures to shift left and more focus on corrective measures to shift right)							
	Result of Quality Risk Analysis			build quality in		demonstrate the quality level				improve the quality	
Item ID	Item short description	Quality characte-ristic	Risk class	intensity preventive quality measures	preventive quality measures (preventive)	intensity static quality measures	static quality measures (preventive & detective)	intensity dynamic quality measures	dynamic quality measures (preventive & detective)	intensity corrective quality measures	corrective quality measures (corrective)
Generic1	Generic quality measures			••	Definition of Ready	••	Apply "INVEST" during reviews	•••	Parallel testing	••	Refactoring
Generic1	Generic quality measures			••	Definition of Done	•	Coding standards				
US1	User story enter client	functionality	A	•••	Pair Programming	••	User story refinement in 4-amigos session	•••	Path testing TDL-2 and exploratory testing	•••	feature toggles
		performance	C	•	Automated perfor-mance monitoring (live environment)	•	Informal review of performance requirements	•	Automated performance testing		
Feature1	Enter order feature	functionality	B	••		••		••		••	

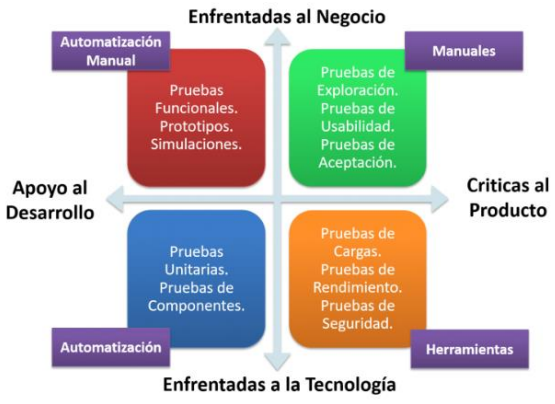
Implementar la estrategia de pruebas



Distribución de los casos de pruebas

Según la pirámide de pruebas:

Según el cuadrante de pruebas:



Factores para la Estimación de Pruebas

- Características del Producto
  - Los riesgos asociados al producto.
  - La calidad de la base de prueba.
  - El tamaño del producto.
  - La complejidad del dominio del producto.
  - Los requisitos para las características de calidad (por ejemplo, seguridad, fiabilidad).
  - El nivel de detalle necesario para la documentación de la prueba.
  - Requisitos para el cumplimiento de normas legales y reglamentarias.
- Características de las Personas
  - Las competencias y la experiencia de las personas involucradas, especialmente con proyectos y productos similares (por ejemplo, conocimiento del dominio).
  - Cohesión y liderazgo del equipo.
- Características del Proceso de Desarrollo
  - La estabilidad y madurez de la organización.
  - El modelo de desarrollo en uso.
  - El enfoque de prueba.
  - Las herramientas utilizadas.
  - El proceso de prueba.
  - Presión con respecto al tiempo.
- Resultados de la Prueba
  - El número y la severidad de los defectos detectados.
  - La cantidad de reconstrucción necesaria.

Técnicas de Estimación de Pruebas

- Estimación experta.
  - Identificar todas las tareas a ejecutar
  - Obtener estimaciones para cada tarea por los responsables (de su ejecución) o por expertos.
  - Sumar todos los valores de las tareas. Incluir los factores de corrección (si hay experiencias respecto de la exactitud de ciertos estimadores).
  - Incluir elementos amortiguadores (buffers o márgenes)/elementos adicionales, con el objeto de cubrir tareas omitidas o subestimadas.
- Estimación basada en analogías.
  - Clasificar las tareas de pruebas requeridas.
  - Buscar un proyecto que se haya desarrollado en el pasado que contenga una tarea similar a una específica.
  - Utilizar el esfuerzo real de esta tarea como base de la estimación.
  - A través del uso de métricas (líneas de código, número de módulos, número de casos de prueba, etc.) como base, calcular el valor de la estimación total.
  - Considerar factores de corrección.

Técnicas de Estimación de Pruebas

- Estimación basada en porcentajes:
  - El esfuerzo para las actividades de pruebas se estiman sobre la base de la totalidad de las actividades del proyecto.
  - El valor del porcentaje requiere ser determinado basándose en la experiencia.
  - La estimación basada en porcentajes no tiene en cuenta el esfuerzo de las pruebas de regresión, que pueden ser una parte sustancial de las pruebas de mantenimiento y asociadas a cambios.
- Basados en COCOMO: TPA (Test Point Analysis)
  - Selección del elemento base: requisito, caso de uso, …
  - Clasificación individual de cada elemento base: tipo, complejidad, importancia, …
  - Asignación de Test Point.
  - Contabilización de los Test Point.
  - Conversión de los Test Point (calculo del esfuerzo).

Modelo de prueba

- Para elaborar el plan de pruebas **estudiaremos toda la documentación anteriormente generada**: el modelo de casos de uso con el documento de requisitos adicionales, el modelo de análisis, el modelo de diseño, el modelo de implementación y el documento de descripción de arquitectura.
- La calidad de dicha documentación es fundamental para la planificación de las pruebas, incidiendo directamente en la calidad del proceso de pruebas.
- Una baja calidad en la documentación o la ausencia de algunos documentos degrada la validez del proceso de pruebas llegando incluso a imposibilitar dicho proceso.
- El **modelo de pruebas** describe cómo se prueban los componentes ejecutables del modelo de implementación con pruebas de integración y de sistema, así como aspectos específicos como puede ser la consistencia de la interfaz de usuario, si el manual del usuario cumple su cometido, y otros.
- Como vimos en su momento, la base son los **casos de prueba**. Recuerda que un caso de prueba específica una forma de probar el sistema, incluyendo la entrada con la que se ha de probar, los resultados que se esperan obtener y las condiciones bajo las que ha de probarse.
- Un **procedimiento de prueba** (o caso de prueba lógico) especifica cómo realizar uno o varios casos de prueba. En el procedimiento documentas los pasos que deben darse para cada uno de los casos de prueba. Los procedimientos de prueba pueden reutilizarse para varios casos de prueba similares. Así mismo, un caso de prueba puede estar incluido en varios procedimientos de prueba.

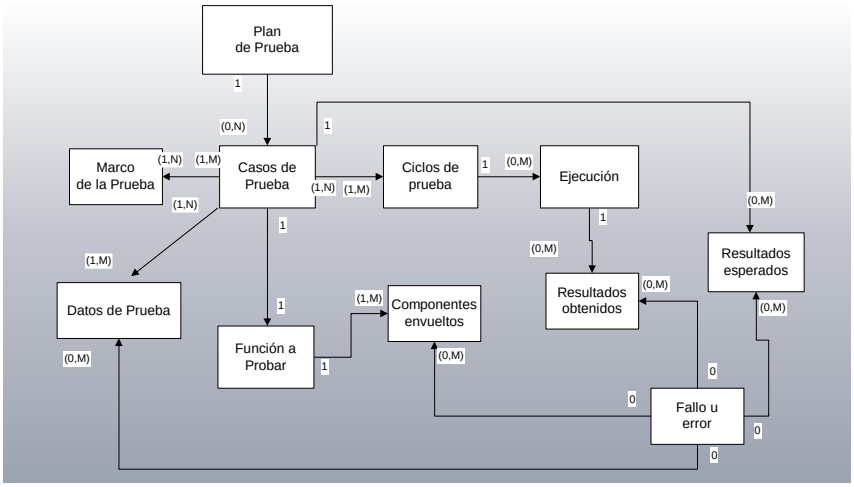
Modelo de prueba

- Un **componente de prueba** (o caso de prueba físico) automatiza uno o varios procedimientos de prueba o partes de éstos. Los componentes de prueba se diseñan e implementan de forma específica para proporcionar las entradas, controlar la ejecución e informar de la salida de los elementos a probar.
- Los componentes de prueba son necesarios puesto que la mayoría de las veces el elemento que estas probando no dispone de un interfaz de usuario que te permita la comunicación directa con él.
- Un **defecto** es una anomalía del sistema, como por ejemplo un síntoma de un fallo software o un problema descubierto en una revisión. Los defectos deben estar documentados indicando síntomas, posibles causas y consecuencias.
- Existen herramientas específicas de prueba de software que permiten establecer los procedimientos de pruebas, ejecutar los procedimientos y evaluar los resultados guardando un registro de los mismos. Si utilizas dichas herramientas, en muchos casos, evitarás la necesidad de desarrollar componentes de prueba.
- Por lo tanto, el modelo de prueba está compuesto de casos de prueba, procedimientos de prueba y componentes de prueba. El modelo se organiza mediante los planes de prueba.

Arquitectura del plan de pruebas

- Casos de prueba (qué probar).
  - Elemento a probar.
  - Condiciones a probar.
  - Juegos de pruebas que aplican.
  - Resultados esperados.
- Procedimientos de prueba (cómo probar).
- Calendario de pruebas (ciclos de prueba).
- Recursos.
- Componentes de pruebas.
  - Modelo de pruebas.
  - Drivers y stubs.
  - Guiones de prueba (scripts).
- Entorno de pruebas.
  - Repositorio y herramientas de prueba.

Arquitectura conceptual del plan de pruebas



Detalles del Plan de pruebas

Identificador del plan

- Preferiblemente de alguna forma mnemónica que permita relacionarlo con su alcance.
- A modo de ejemplo: TP-Global (plan global del proceso de pruebas), TP-REQ-Seguridad1 (plan de verificación del requerimiento 1 de seguridad), TP-Contr-X (plan de verificación del contrato asociado al evento de sistema X), TP-Unit-Despachador.iniciar (plan de prueba unitario para el método iniciar de la clase Despachador).
- Como todo artefacto del desarrollo, está sujeto a control de configuración, por lo que debe distinguirse adicionalmente la versión y fecha del plan.

Detalles del Plan de pruebas

Alcance

- Indicas el nivel de prueba y las propiedades / elementos del software a ser probado.

Elementos a probar

- Indicas la configuración a probar y las condiciones mínimas que debe cumplir para comenzar a aplicarle el plan.
- Por un lado, es difícil y arriesgado probar una configuración que aún contiene fallos; por otro lado, si esperamos a que todos los módulos estén perfectos, puede que detectemos fallos graves demasiado tarde.

Detalles del Plan de pruebas

Estrategia

- Describes la técnica, patrón y/o herramientas a utilizar en el diseño de los casos de prueba.
- Por ejemplo, en el caso de pruebas unitarias de un procedimiento, esta sección podría indicar: "Se aplicará la estrategia caja negra de valores límite de la precondición" o "Ejercicio de los caminos básicos válidos".
- En lo posible, la estrategia debe precisar el número mínimo de casos de prueba a diseñar, por ejemplo el 100% de las fronteras, el 60% de los caminos ciclomáticos,... La estrategia también especifica el grado de automatización que se exigirá, tanto para la generación de casos de prueba como para su ejecución.

Detalles del Plan de pruebas

Procedimientos de prueba

- Especificas las condiciones bajo las cuales, el plan debe ser:
  - Suspendido.
  - Repetido.
  - Finalizado.
- En algunas circunstancias (que deben ser especificadas) el proceso de prueba debe suspenderse a la vista de los defectos o fallos que se han detectado. Al corregirse los defectos, el proceso de prueba previsto por el plan puede continuar, pero debes especificar a partir de qué punto, ya que puede ser necesario repetir algunas pruebas.
- Los criterios de culminación pueden ser tan simples como aprobar el número mínimo de casos de prueba diseñados o tan complejos como tomar en cuenta no sólo el número mínimo, sino también el tiempo previsto para las pruebas y la tasa de detección de fallos.

Detalles del Plan de pruebas

Tangibles

- Especificas los documentos a entregar al terminar el proceso previsto por el plan (por ejemplo: subplanes, especificación de pruebas, casos de prueba, resumen gerencial del proceso e histórico de pruebas).
- No es suficiente con ejecutar simplemente las pruebas, sino que se deben almacenar sus resultados de forma sistemática. Es posible auditar el proceso para comprobar que se ha llevado a cabo de forma correcta.

Procedimientos especiales

- Identificas el grafo de las tareas necesarias para preparar y ejecutar las pruebas, así como cualquier habilidad especial que se requiere.

Detalles del Plan de pruebas

Infraestructura y entorno

- Especificas las propiedades necesarias y deseables del ambiente de prueba, incluyendo las características del hardware, el software de sistemas, cualquier otro software necesario para llevar a cabo las pruebas, así como la colocación específica del software a probar (por ejemplo, qué módulos se colocan en qué máquinas de una red local) y la configuración del software de apoyo.

Recursos Humanos

- La sección incluye una estimación de los recursos humanos necesarios para el proceso.
- También se indica cualquier requerimiento especial del proceso: actualización de licencias, espacio de oficina, tiempo en la máquina de producción, seguridad.

Detalles del Plan de pruebas

Calendario

- Esta sección describe los hitos del proceso de prueba y el grafo de dependencia en el tiempo de las tareas a realizar.
- Estableces un calendario de prueba global y la asignación de recursos para este calendario. Esto, obviamente, está ligado al calendario general de desarrollo del proyecto.

Gestión de riesgos

- Especifica los riesgos del plan, las acciones mitigantes y de contingencia.

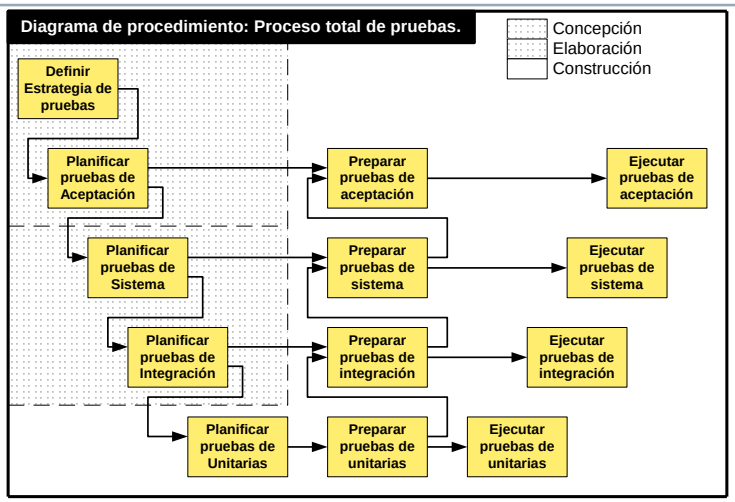
Responsables

- Especifica quién es el responsable de cada una de las tareas previstas en el plan.

Características del plan de pruebas

- Este plan debería contemplar cantidades significativas de eventualidades, de forma que errores en el diseño o la implementación se puedan solucionar y parte del personal que realiza las pruebas se pueda dedicar a otras actividades.
- Como otros planes, el plan de prueba no es un documento estático. Debe revisarse regularmente puesto que la prueba es una actividad dependiente del avance de la implementación. Si parte del sistema a probar está incompleto, el proceso de prueba del sistema no puede comenzar.
- Debes contar con un plan de pruebas global y tantos planes de pruebas como sean necesarios para cubrir el alcance del plan global.
- La planificación de las pruebas comienza desde las fases iniciales del desarrollo tal y como muestra el siguiente diagrama:

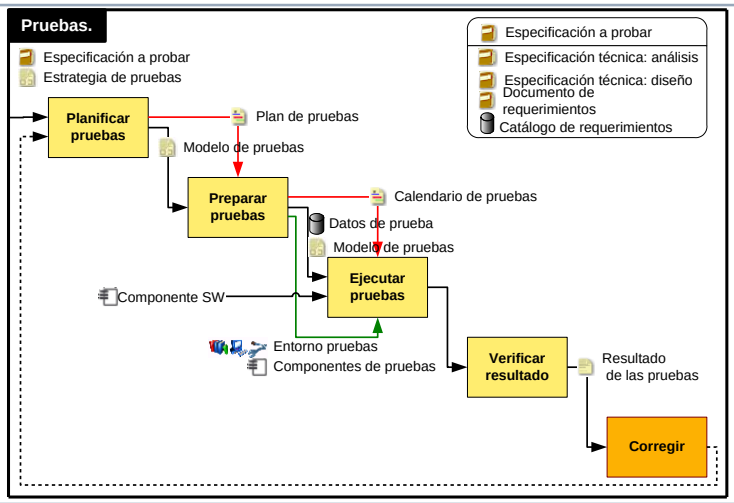
Proceso total de pruebas



Metodología

- El primer paso que debes dar es la **planificación de la prueba**, pasando a continuación a su **diseño**.
- Para diseñar la prueba debes identificar y diseñar los casos de prueba de integración, los casos de prueba de sistema, los casos de prueba de regresión y los procedimientos de prueba.
- Una vez concluido el diseño, debes **implementar los componentes de pruebas necesarios**.
- Por último realizas las **pruebas de integración** y la **prueba de sistema**.
- Concluyes el proceso con la **evaluación de la prueba**.

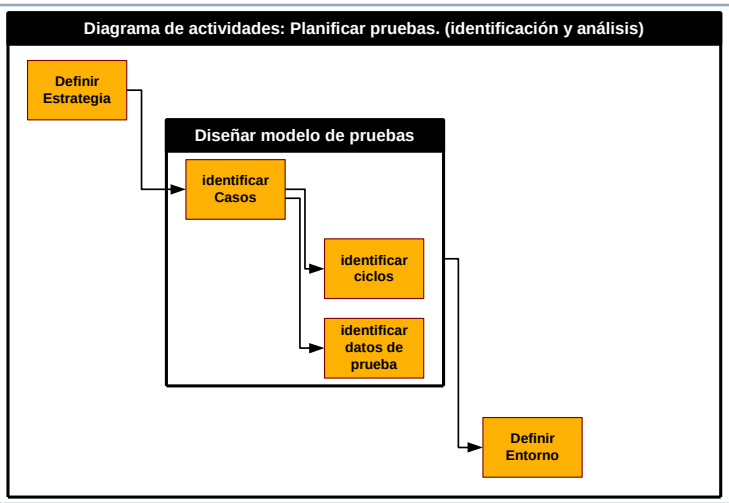
Metodología



Planificar la prueba

- El primer paso que debes dar es planificar los esfuerzos de prueba describiendo una estrategia y estimando los requisitos de la prueba, tanto en recursos humanos como en sistemas necesarios.
- El modelo de casos de uso y los requisitos adicionales, junto con el modelo de diseño te ayudan a decidir el tipo adecuado de pruebas y a estimar el esfuerzo necesario para llevarlas a cabo.
- Utilizando ambos modelos estableces una estrategia de prueba decidiendo qué tipo de pruebas ejecutar, cómo y cuándo ejecutar dichas pruebas y cómo determinar si el esfuerzo de la prueba tiene éxito.

Planificar las pruebas



Diseñar la prueba

- Para diseñar la prueba empiezas por identificar y describir los casos de prueba de cada componente.
- La selección de las técnicas de pruebas depende factores adicionales como pueden ser requisitos contractuales o normativos, documentación disponible, tiempo, presupuesto, conocimientos, experiencia, …
- Cuando dispongas de los casos de prueba, identificas y estructuras los procedimientos de prueba describiendo cómo ejecutar los casos de prueba.

Diseño de los casos de prueba unitarias

- Los casos de prueba unitaria deben estar integrados en el proceso de codificación.
- Es responsabilidad del programador crear e implementar los casos de prueba que verifiquen que el componente cumple con los requerimientos de implementación recibido. Habitualmente se emplean técnicas de caja blanca con un determinado nivel de cobertura de código.
- En caso de que debas crear casos de prueba adicionales a los del programador, se recomienda utilizar técnicas de caja negra como técnica complementaria a la empleada por el programador.
- El desarrollo guiado por pruebas o Test-driven development (TDD) es una práctica de ingeniería de software que involucra escribir las pruebas primero y refactorizar depues. Para escribir las pruebas generalmente se utilizan las pruebas unitarias:
  - Antes de crear el componente, se escribe una prueba y se verifica que las pruebas fallan.
  - A continuación, se implementa el componente con el código que hace que la prueba pase satisfactoriamente y se verifica.
  - Seguidamente se refactoriza el código escrito con el código real del componente y se verifica.

Características de una buena prueba unitaria

- Rápida. No es infrecuente que los proyectos maduros tengan miles de pruebas unitarias. Las pruebas unitarias deberían tardar muy poco tiempo en ejecutarse.
- Aislada. Las pruebas unitarias son independientes, se pueden ejecutar de forma aislada y no tienen ninguna dependencia en ningún factor externo, como sistemas de archivos o bases de datos.
- Reiterativa. La ejecución de una prueba unitaria debe ser coherente con sus resultados, es decir, devolver siempre el mismo resultado si no cambia nada entre ejecuciones.
- Autocomprobada. La prueba debe ser capaz de detectar automáticamente si el resultado ha sido correcto o incorrecto sin necesidad de intervención humana.
- Oportuna. Una prueba unitaria no debe tardar un tiempo desproporcionado en escribirse en comparación con el código que se va a probar. Si observa que la prueba del código tarda mucho en comparación con su escritura, considere un diseño más fácil de probar.

Bases de las pruebas unitarias

Entradas

- Requerimiento del componente
- Criterios de aceptación
- Diseño de detalle
- Código

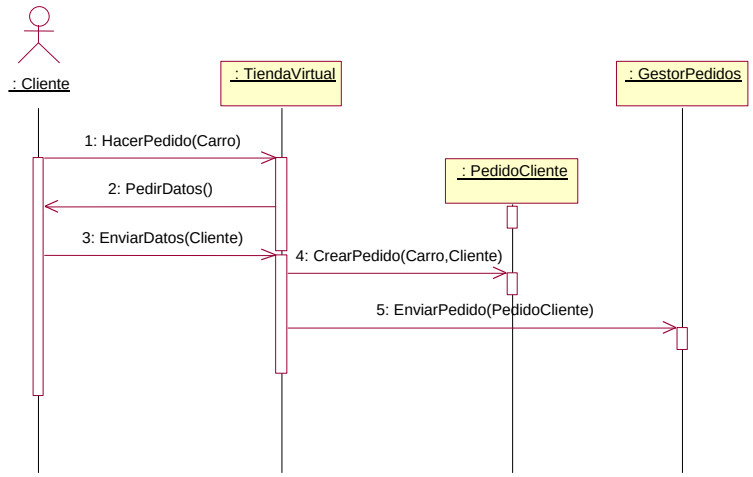
Objetivos

- Componentes
- Programas
- Módulos de bases de datos

Diseño de los casos de prueba de integración

- Los casos de prueba de integración se utilizan para verificar que los componentes interaccionan entre sí de la forma apropiada después de haber sido integrados en una construcción.
- En este caso te serán de especial utilidad los diagramas de secuencia, los diagramas de colaboración y los diagramas de estados que se elaboran en las realizaciones de casos de uso del "*modelo de diseño*".
- Dichos diagramas representan las interacciones entre los objetos del diseño.
- Para encontrar los casos de prueba debes buscar las combinaciones de entrada, salida y estado que den lugar a escenarios interesantes que utilicen los objetos representados en los diagramas.
- El conjunto de casos de prueba que selecciones debe ir encaminado a conseguir los objetivos del plan de pruebas con el esfuerzo mínimo.
- Para ello eliges casos de uso que no se solapen entre sí y que cada uno de ellos pruebe un camino o escenario interesante en la realización de caso de uso.

Diagrama de secuencias



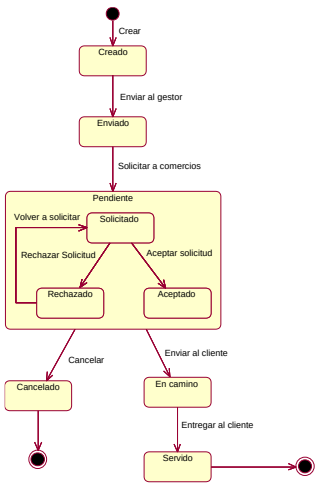
Análisis del Diagrama de secuencias

- Puedes extraer un caso de pruebas que describa cómo probar la secuencia representada en el diagrama, tomando en cuenta el estado inicial del sistema, las entradas del actor y demás circunstancias necesarias que hacen que ocurra la secuencia.
- Si el diagrama representara varias secuencias posibles, debes extraer tantos casos de pruebas como sean necesarios para probar todas las secuencias.

S1	Los datos del carro son correctos	SI	SI	NO	NO
S3	Los datos del cliente son correctos	SI	NO	SI	NO
S2	Pedir datos de cliente		X		
S4	Crear pedido	X			
S5	Enviar pedido	X			

Diagramas de estado

- De igual forma debes actuar con los diagramas de estado que representan todos los posibles estados de un elemento del sistema y sus correspondientes transiciones de estado.
- Debes extraer casos de prueba que te permitan probar cada una de las transiciones con sus posibles itinerarios.
- No olvides que NO debes centrarte exclusivamente en las transiciones posibles sino que también debes estudiar las imposibles (quizás las más importantes), como por ejemplo la cancelación de un pedido que se encuentra en camino.



Bases de las pruebas de integración

Entradas

- Diseño de software y sistema
- Arquitectura
- Flujos de trabajo
- Casos de uso
- Criterios de aceptación

Objetivos

- Componentes
- Interfaces
- Infraestructura
- Bases de datos
- Configuración del sistema

Diseño de los casos de prueba de sistema

- Las pruebas de sistema se usan para probar que el sistema funciona correctamente como un todo.
- Las pruebas de sistema se centran fundamentalmente en probar combinaciones de diferentes casos de uso ejecutados bajo diferentes condiciones.
- Las condiciones deben incluir, a su vez, combinaciones diferentes de configuraciones hardware, niveles de carga del sistema, número de actores y tamaños de base de datos.
- La forma de identificar los casos de prueba depende estrechamente del tipo de sistema que estés probando y de sus características especiales.

Diseño de los casos de prueba de sistema

- El **modelo de casos de uso** te servirá de punto de partida, fíjate en sus flujos y requisitos especiales (como son las restricciones funcionales y los requisitos de rendimiento).
- El **modelo de despliegue**, elaborado en la fase de diseño, también te aporta mucha información.
- El modelo de despliegue describe la distribución física del sistema, asignando funcionalidades a nodos de cómputo. Así mismo, describe las relaciones de los nodos entre sí mediante medios de comunicación (Internet, Intranet y similares).

Diseño de los casos de prueba de sistema

- Debes hacer hincapié en los casos de uso que:
  - Son críticos para el sistema.
  - Tienen tiempos de respuesta determinados.
  - Manejan grandes volúmenes de información.
  - Usan frecuentemente los recursos del sistema (procesadores, bases de datos,...).
  - Utilizan medios de comunicación.
  - Requieren procesamiento especial (en paralelo, multiprocesador,...)

Bases de las pruebas de sistema

Entradas

- Casos de uso
- Especificaciones funcionales
- Especificaciones de requisitos de software y sistema
- Criterios de aceptación
- Informe de análisis de riesgos

Objetivos

- Sistema
- Manuales de sistema, usuario y funcionamiento
- Bases de datos
- Configuración del sistema

Diseño de los casos de prueba de regresión

- Las pruebas de regresión son aquellas que diseñan o se repiten para buscar los errores que se producen después del proceso de corrección de los errores.
- Algunos de los casos de pruebas diseñados en los pasos anteriores son utilizados en las pruebas de regresión.
- Dependiendo de la cobertura, la repetición de casos de pruebas existentes y se deben diseñar casos específicos buscando el impacto del cambio.
- El diseño de los casos de prueba utilizados en las pruebas de regresión lo debes realizar con suficiente flexibilidad como para que soporten los cambios del software que estás probando.

Bases de las pruebas de regresión

Entradas

- Notificaciones de defectos
- Solicitudes de cambio
- Casos de uso
- Especificaciones funcionales
- Especificaciones de requisitos de software y sistema
- Criterios de aceptación
- Informe de análisis de riesgos

Objetivos

- Componentes
- Sistema
- Manuales de sistema, usuario y funcionamiento
- Bases de datos
- Configuración del sistema

Diseño de los casos de prueba de aceptación

- Las pruebas de aceptación se basan fundamentalmente en el punto de vista del usuario final.
- El **modelo de casos de uso y las historias de usuario** te indican como va a utilizar el usuario el sistema y te permite crear casos de prueba para cada uso del sistema.
- Requieren una revisión formal de que toda la funcionalidad pactada es entregada, su objetivo principal no es tanto localizar defectos como evaluar la buena disposición del sistema para su despliegue y explotación.
- Las pruebas de aceptación tienen una vertiente contractual y normativa que toman como base los criterios de aceptación previstos en el contrato de desarrollo o las normas de obligado cumplimiento.
- En dicha situación, los casos de prueba se diseñan para comprobar el efectivo cumplimiento de cada uno de los criterios y normas.

Bases de las pruebas de aceptación

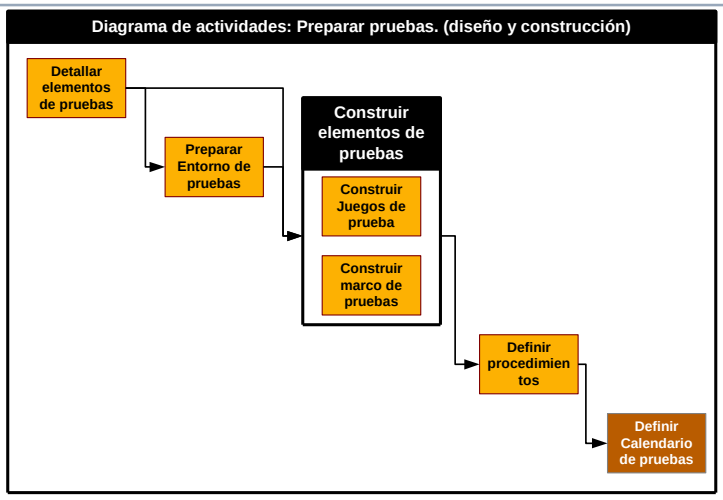
Entradas

- Requisitos del usuario
- Requisitos del sistema
- Casos de uso
- Historias de usuario
- Criterios de aceptación
- Procesos de negocio
- Informe de análisis de riesgos

Objetivos

- Procesos de negocio
- Procesos operativos y de mantenimiento.
- Procedimientos de usuario
- Formularios
- Informes

Preparar las pruebas



Identificación y estructuración de los procedimientos de prueba

- Con los casos de prueba ya diseñados, pasas a identificar qué procedimientos se han de seguir para realizar los casos de prueba.
- En un documento especificas cómo realizar el proceso.
- Por economía, debes intentar reutilizar los procedimientos de prueba ya existentes, aunque sea necesario adaptarlos o modificarlos.
- De la misma forma, si creas un nuevo procedimiento intenta que sea reutilizable para varios casos de pruebas.
- Esto te permitirá usar un conjunto reducido de procedimientos de pruebas con rapidez y precisión para muchos casos de prueba.

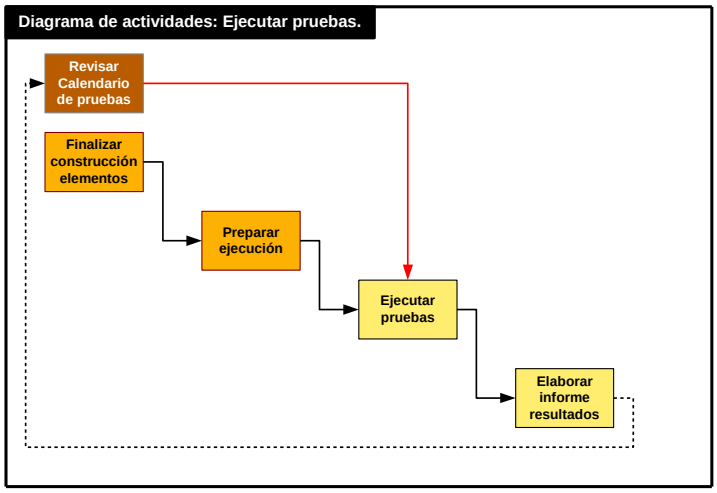
Implementar las pruebas

- El propósito de la implementación de las pruebas es automatizar, siempre que sea posible, los procedimientos de pruebas creando componentes de prueba (módulos conductores y módulos resguardo o auxiliares) y el repositorio o diccionario de datos de prueba.
- Esto lo harás siempre que el procedimiento se pueda automatizar (no siempre se puede, por ejemplo en el caso de las pruebas de interfaz de usuario) y no dispongas de herramientas específicas de pruebas o no puedas reutilizar componentes previamente desarrollados.
- La implementación de una prueba requiere un desarrollo completo.

Implementar las pruebas

- Partiendo del procedimiento de prueba como documento de requisitos debes realizar el análisis, diseño, implementación y prueba del componente de prueba.
- Aunque parezca surrealista, no es tan infrecuente como se podría pensar que el componente de prueba falle, invalidando la prueba para la que está diseñado.
- Los componentes de pruebas usan a menudo grandes cantidades de datos de entrada y generan, a su vez, grandes cantidades de datos de salida como resultado.
- Es por tanto deseable que incorpores como requisito el uso de bases de datos y hojas de cálculo donde se definan los elementos de entrada y salida del componente, que permitan la visualización e interpretación de los mismos de una forma clara e intuitiva.

Ejecutar las pruebas



Realizar las pruebas de integración

- Para realizar las pruebas de integración llevas a cabo los siguientes pasos:
  1. Realizas las pruebas siguiendo los pasos indicados el procedimiento de prueba.
  2. Comparas los resultados obtenidos con los esperados e investigas aquellos que no coinciden con lo esperado.
  3. Informas de los defectos a los responsables de los componentes que crees que contienen fallos.
  4. Recopilas los defectos para su posterior evaluación y uso en las pruebas de regresión.

Realizar la prueba de sistema

- Puedes comenzar con la prueba de sistema cuando las pruebas de integración indiquen que el sistema cumple los objetivos de calidad, fijados en el plan de prueba, para la integración (por ejemplo: el 98% de los casos de prueba se ejecutan con el resultado esperado).
- La prueba de sistema la realizas siguiendo los mismos pasos que en las pruebas de integración.

Evaluar la prueba

- Ya por último, pero no menos importante, realizas la evaluación de las pruebas.
- La evaluación la realizas estudiando los resultados y defectos obtenidos en las pruebas.
- Dicho estudio te permite preparar unas métricas que determinen el nivel de calidad y el espectro de pruebas.
- El elenco de métricas de calidad es muy amplio permitiendo medir la fiabilidad, seguridad y disponibilidad, entre otras, del sistema.
- La cobertura de pruebas indica el porcentaje de casos de prueba que han sido ejecutados y el porcentaje de código que ha sido probado.

Evaluar la prueba

- Basándote en el análisis de las tendencias de los defectos puedes sugerir:
  - Realizar pruebas adicionales.
  - Rebajar los criterios de calidad.
  - Aislar partes del sistema cuya calidad parece aceptable para su entrega, dejando el resto para ser revisado y probado de nuevo.
- El resultado de la evaluación lo debes recoger en un documento o informe con todas tus conclusiones, métricas y sugerencias.
- Como existen múltiples partes interesadas, los informes deben adaptarse a sus necesidades. En general, proponemos tres niveles de presentación de informes:
  - Informes detallados para los desarrolladores, evaluadores, …
  - Informes generales para personas que analizan el objeto de prueba desde una perspectiva empresarial, como el propietario del producto y los usuarios.
  - Informes de alto nivel para las partes interesadas que están más distantes de las actividades del equipo, como los gerentes de alto nivel.

Evaluar la prueba

- Cuando una prueba falla, esto a menudo significa que el resto del escenario de prueba no se puede completar y, por lo tanto, algunos de los casos de prueba no se ejecutarán. Por lo tanto, la ejecución de la prueba tiene tres resultados posibles para un caso de prueba: pasar, fallar o no ejecutarse.
- Cuando se supera el caso de prueba, el evaluador lo registra y no habrá más acciones. Cuando falla el caso de prueba, el evaluador deberá realizar una investigación.
- En la ejecución de pruebas automatizadas, el script de prueba también contendrá la verificación automatizada de los resultados esperados y reales y la herramienta de prueba se encargará de este registro. Se informarán las diferencias para que los miembros del equipo puedan realizar sus investigaciones.
- Principalmente, hay dos posibles razones por las que no se supera una prueba: hay un fallo en el objeto de la prueba o es un error de la propia prueba (resultado esperado incorrecto, escenarios incorrectos, especificación ambigua, …). Es aconsejable comenzar a examinar el caso de prueba en sí, primero hay que asegurarse de que se ha cometido un error en la prueba para evitar que hacer perder el tiempo a otras personas con un informe de anomalía.

Evaluar la prueba

- Para investigar las pruebas no superada hay que realizar las siguientes tareas:
  - Reunir evidencias (como capturas de pantalla o volcados de bases de datos)
  - Reproducir el fallo (y registrar los pasos para reproducirlo)
  - Verificar fallos en la prueba.
  - Determinar las causas sospechosas
  - Aislar la causa (opcional)
  - Generalizar la anomalía
  - Comparar con otras anomalías y eliminar duplicados
  - Registrar el informe de anomalía
- Los fallos y errores se registran como defectos en un sistema de gestión de defectos que permite la trazabilidad desde su creación hasta su cierre y permite las comunicación entre todas las partes interesadas. Cuando la anomalía se soluciona de inmediato, no es necesario registrar el defecto.

Gestión de Defectos

- Dado que uno de los principales objetivos de prueba es encontrar defectos, es esencial contar con un proceso establecido de gestión de defectos.
- Aunque comúnmente nos referimos a "defectos", las anomalías notificadas pueden resultar ser defectos reales u otra cosa (por ejemplo, un falso positivo, una solicitud de cambio); esto se resuelve durante el proceso de tratamiento de los informes de defecto.
- Las anomalías pueden informarse durante cualquier fase del ciclo de vida y la forma depende de éste.
- Como mínimo, el proceso de gestión de defectos incluye un flujo de trabajo para tratar las anomalías individuales desde su descubrimiento hasta su cierre y reglas para su clasificación. El flujo de trabajo suele incluir actividades para registrar las anomalías notificadas, analizarlas y clasificarlas, decidir una respuesta adecuada como arreglarlas o mantenerlas tal cual y, por último, cerrar el informe de defecto. El proceso debe ser seguido por todos los implicados. Es aconsejable tratar los defectos de las pruebas estáticas (especialmente el análisis estático) de forma similar.
- Hay disponibles muchas herramientas que simplifican el proceso de gestión de defectos.

Terminología de las anomalías

- Una anomalía es una diferencia entre el comportamiento esperado y el resultado real de una prueba. Este queda registrado para que se pueda analizar y resolver la causa.
- Los términos que se utilizan a menudo en relación con anomalías son:
  - Un **error** es un error humano que puede, aunque no necesariamente, provocar fallas o fallas.
  - Un **defecto** o falta es la manifestación de un error residente en el código o en un documento o en un sistema. Esto puede causar fallos. Se pueden detectar los defectos mediante las pruebas estáticas.
  - Un **fallo** indiaca la desviación de lo esperado y es la manifestación de uno o mas defectos es una. Se puede detectar los fallos mediante pruebas dinámicas.
  - Un **incidente** es una interrupción no planificada de un servicio de TI o una reducción en la calidad de un servicio de TI o una falla de un elemento de configuración.
  - Un **problema** es una causa, o causa potencial, de una o más anomalías o incidentes.

Características de los defectos

- **Consecuencias del error**: no es lo mismo una cabecera errónea que un fallo en el suministro eléctrico.
- **Gravedad del error**: indica el impacto de la anomalía en el proceso de negocio. La gravedad es objetiva y no cambia con el tiempo.
- **Prioridad del error**: indica el orden en el que deben tratados. Las prioridades pueden cambiar.
- **Frecuencia del error**: ¿cuántas veces se produce un error? Evidentemente, será necesario prestar una mayor atención a los errores más comunes.
- **Coste de corrección**: este coste es una combinación del coste de descubrir el error y del coste de corregirlo. Además, este coste crece exponencialmente según se avanza en el proceso de desarrollo.
- **Coste de instalación**: especialmente en entornos distribuidos, a la hora tanto de instalar un sistema como de implantar cualquier corrección sobre el mismo puede llegar a ser un factor muy importante.
- **Origen causal**: fallo ← defecto ← causa.

Definición de los defectos

- Identificador único.
- Título con un breve resumen de la anomalía que se informa.
- Fecha en que se observó la anomalía, organización emisora y autor, incluido su rol.
- Identificación del objeto de prueba y del entorno de prueba.
- Contexto del defecto (por ejemplo, caso de prueba que se está realizando, actividad de prueba que se está llevando a cabo, fase del CVDS y otra información relevante como la técnica de prueba, la lista de comprobación o los datos de prueba que se están utilizando).
- Descripción del fallo para permitir su reproducción y resolución, incluidos los pasos que detectaron la anomalía, y cualquier registro de prueba, volcado de la base de datos, captura de pantalla o grabación pertinentes.
- Resultados esperados y resultados reales.
- Severidad del defecto (grado de impacto) sobre los intereses de los implicados o los requisitos.
- Prioridad de corrección.
- Estado del defecto (por ejemplo, abierto, aplazado, duplicado, a la espera de ser corregido, a la espera de pruebas de confirmación, reabierto, cerrado, rechazado).
- Referencias (por ejemplo, al caso de prueba).

Tipos de error

- Requerimientos
- Funcionales
- Estructurales
- Procedurales:
  - control,
  - lógica,
  - proceso,
  - inicialización
- Datos
- Implementación
- Interfaces
- Plataforma

Gestión de Cambios

- En el último paso de la realización de las pruebas y en el documento de evaluación de las pruebas has informado de los defectos encontrados. La corrección de dichos defectos desencadena una solicitud de cambio. Dichos cambios deben ser gestionados.
- El control de cambios para el software es fundamental para el control de costes así como para la estabilidad de la tecnología. Los cambios de diseño y código deben ser realizados para corregir problemas encontrados en el desarrollo y pruebas, pero estos deben ser cuidadosamente introducidos.
- Si los cambios no son controlados, entonces el diseño ordenado, la implementación y las pruebas son imposibles y el plan de calidad no puede ser efectivo.
- Al introducir cambios se invalidan los resultados de las pruebas, rendimientos pasados no garantizan rendimientos futuros, la pruebas superadas en la versión anterior pueden fallar en la versión actual. Hay que probar los cambios (pruebas de progresión) y controlar que las modificaciones no produzcan efectos secundarios negativos (pruebas de regresión).
- El siguiente esquema muestra la secuencia de actividades para realizar la gestión de cambios:

Gestión de Cambios