



UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA



CURSO:
Métodos Formales de Pruebas - Testing de Software

Tema:
Introducción a Pruebas de Software

Docente: Mg. Wilder Inga

Reglas básicas

- 10 minutos de tolerancia
- Toma de asistencia al inicio de la clase
- Participación durante la clase
- Participación proactiva en los ejercicios grupales, se usaran herramientas colaborativas de G Suite.
- Las ausencias justificadas no dan derecho a nota (actividades en clase)
- Break 15 minutos

Video de Calidad de Software

Introducción

- ⑨ “All in all, coders introduce bugs at the rate of 4.2 defects per hour of programming. If you crack the whip and force people to move more quickly, things get even worse”
 - ③ Watts Humphrey (note),
<http://www.cs.usask.ca/grads/jpp960/490/BombSquad.html>
- ⑨ 20 por Día / 100 por Semana / 400 por Mes / 5000 por Año
- ⑨ “5000 Defect Project (not atypical for IBM)”
 - ③ Paul Gibson, Testing Challenges for IBM, UK Test 2005 Keynote, <http://www.uktest.org.uk>

Introducción.

Algunas cifras

- ⑨ Total de recursos empleados en pruebas:
 - ③ 30% a 90% [Beizer 1990]
 - ③ 50% a 75% [Hailpern & Santhanam, 2002]
 - ③ 30% a 50% [Hartman, 2002]
- ⑨ Mercado de herramientas: \$2,6 billion
 - ③ Coste por infraestructura inadecuada:
 - Transporte y manufactura: \$1,840 billion
 - Servicios financiero: \$3,342 billion
 - ③ Costes de reparación en función del instante en el ciclo de vida [Baziuk 1995]
 - Requisitos: x 1
 - Pruebas de sistema: x 90
 - Pruebas de instalación: x 90-440
 - Pruebas de aceptación: x 440
 - Operación y mantenimiento: x 880
 - ③ Fuente general: The economic impact of inadequate infrastructure for software testing. NIST Report – May 2002



El control de calidad se ocupa de las actividades que garantizan que el producto se desarrolle según los requisitos definidos. Se ocupa de todas las acciones que son importantes para controlar y verificar determinadas características del producto, incluidas las pruebas. La inspección y las pruebas de los productos es el aspecto más importante del control de calidad.

Software Testing VS Quality Assurance

- En la industria de TI, a menudo se observa que las personas generalmente no distinguen entre QA del software y las pruebas de software. Los tester a menudo se consideran profesionales de QA del software porque los objetivos de las pruebas de software y el aseguramiento de la calidad son los mismos, ejm. Garantizar que el software sea de la mejor calidad.
- Como su nombre indica, los procesos de QA se llevan a cabo para asegurar que la calidad del producto esté en línea con los requisitos del cliente. Los profesionales de QA trabajan en el desarrollo e implementación de todos los procesos necesarios para asegurar que todos los procedimientos necesarios del ciclo de vida del desarrollo de software se sigan correctamente.

Software Testing VS Quality Assurance

Atención proactiva de QA:

1. Prevención de defectos
2. Procesos
3. Mejora continua de procesos

Atención de testing:

Identificar o descubrir defectos y errores en el software. Implica una prueba rigurosa real del software para ver si hay algún defecto o variación del requisito del cliente que deba solucionarse.

Son parte de QC y se enfocan solo en actividades orientadas al producto. La prueba del software se lleva a cabo durante la fase de testing y solo se identifican los defectos y no se corrigen en este proceso. La reparación de defectos no forma parte de las pruebas de software.

Cost of poor quality

“The Cost of Poor Software Quality in the US: a 2018 report” **How much was spent on IT products, services, and labor?**:

- \$2 trillion+ in the United States (estimate by CISQ) – 6.2% vs 2017
- \$3.7 trillion globally (estimate by Gartner) – 11.1% vs 2017
- \$3.74 trillion in 2019 (0.6% vs 2018)
- \$6.3 trillion globally (estimate by Apptio)
- Aprox. 45% labor cost

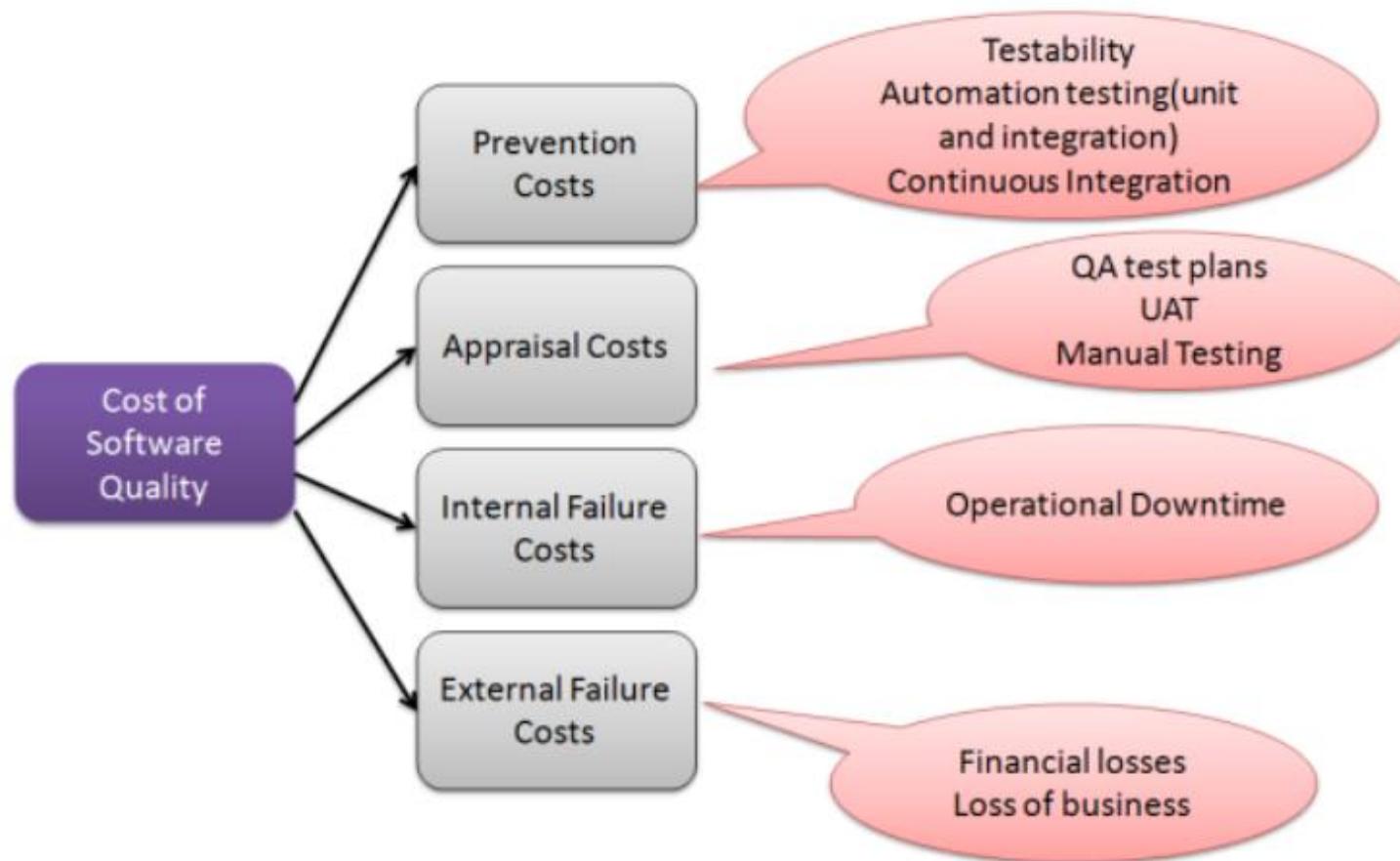
CPSQ (Cost of Poor Software Quality) for the United States in 2018:

- External failures and deficiencies – \$1.43 trillion
- Internal failures and deficiencies – \$.8 trillion
- Technical debt – \$.54 trillion
- Management failures – unknown contribution at this time

Cost of Software Quality

El costo de la calidad es importante porque cuando decide realizar pruebas de software para su producto, realmente va a invertir su tiempo, dinero y esfuerzo en realizar controles de calidad. Al realizar un análisis del costo de la calidad del software, sabría cuál es el retorno de esa inversión (ROI).

Cost of Software Quality



Cost of Software Quality

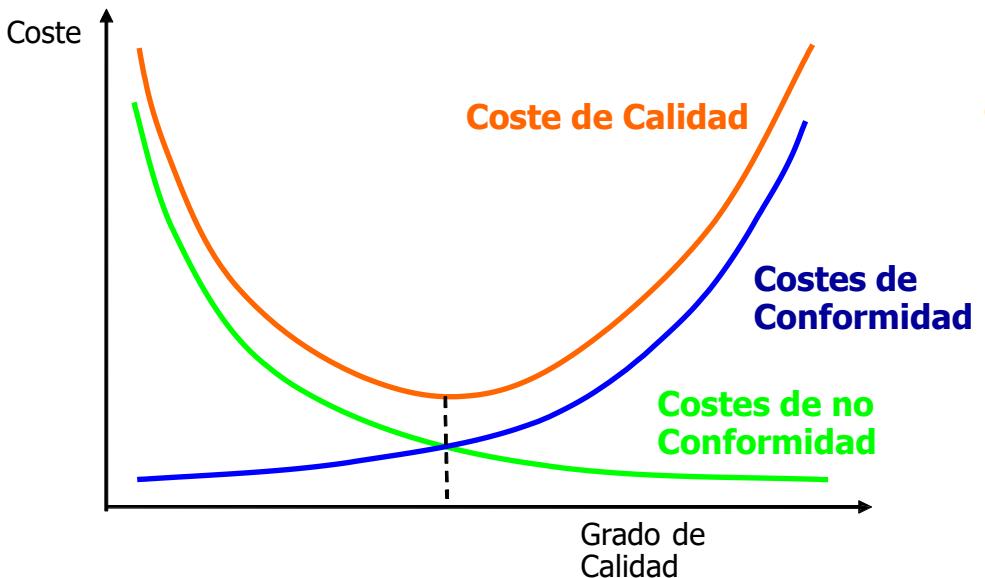
Costos de conformidad

- Costos de prevención: monto gastado para garantizar que todas las prácticas de garantía de calidad se sigan correctamente. Esto incluye tareas como capacitar al equipo, revisiones de código y cualquier otra actividad relacionada con el control de calidad, etc.
- Costos de evaluación: es la cantidad de dinero gastada en planificar todas las actividades de prueba y ejecutarlas, como la elaboración de casos de prueba y luego ejecutarlos.

Costos de no conformidad

- A nivel internos: es el gasto que ocasiona la subsanación de los errores. Costo de las pruebas unitarias por subsanación.
- A nivel externos: es el gasto que se produce cuando el defecto es encontrado por el cliente. Estos gastos son mucho más de los que surgen a nivel interno, especialmente si el cliente no está satisfecho o el error se disemina afectando colateralmente a otros componentes de un sistema.

Costes



$$⑨ C = C_{\text{conformidad}} + C_{\text{noconformidad}}$$

$$③ C_{\text{conformidad}} = C_{\text{prevención}} + C_{\text{evaluación}}$$

$$③ C_{\text{noconformidad}} = C_{\text{internos}} + C_{\text{externos}}$$

⑨ Invertir en pruebas (y en general en calidad) es rentable

③ (Krasner)

- En manufactura: 5-25%

- En software: 10-70%

③ En Empresas de Automoción:

- 4% Excelencia

- 4-8% Buenos

- >10% No deseables

⑨ Costes de inactividad en cliente (N. Donfrio, 2002):

- ③ Cadena Suministro: 300K\$/hora

- ③ ERP/e-comm: 480K\$/hora

Introducción.

Definiciones

- ⑨ Definición 1: La prueba (testing) es el proceso de ejecutar un programa con la intención de encontrar fallos [Glenford J. Myers]
 - ③ Un buen caso de prueba es el que tiene alta probabilidad de detectar un nuevo error
 - ③ Un caso de prueba con éxito es el que detecta un error nuevo
- ⑨ Definición 2 [Cem Kaner]:
 - ③ Una investigación técnica del producto bajo prueba
 - ③ ...para proporcionar a los interesados (stakeholders)
 - ③ ...información relacionada con la calidad

Deuda técnica

- La Deuda Técnica es una excelente metáfora (creada por Ward Cunningham) que nos ayuda a pensar sobre algunos problemas del desarrollo de software. Según la metáfora, hacer las cosas rápido y mal nos incrementa la deuda técnica, la cual es similar a la deuda financiera. Al igual que la deuda financiera, la deuda técnica tiene pago de intereses, que vienen en la forma del esfuerzo extra que será necesario hacer en el futuro por una elección rápida y mala de diseño. Podemos decidir seguir pagando el interés, o podemos pagar el capital al hacer un refactor del diseño hacia un diseño mejor. Aunque hay un costo de pagar este capital, nos ahorraremos el pago de intereses en el futuro.

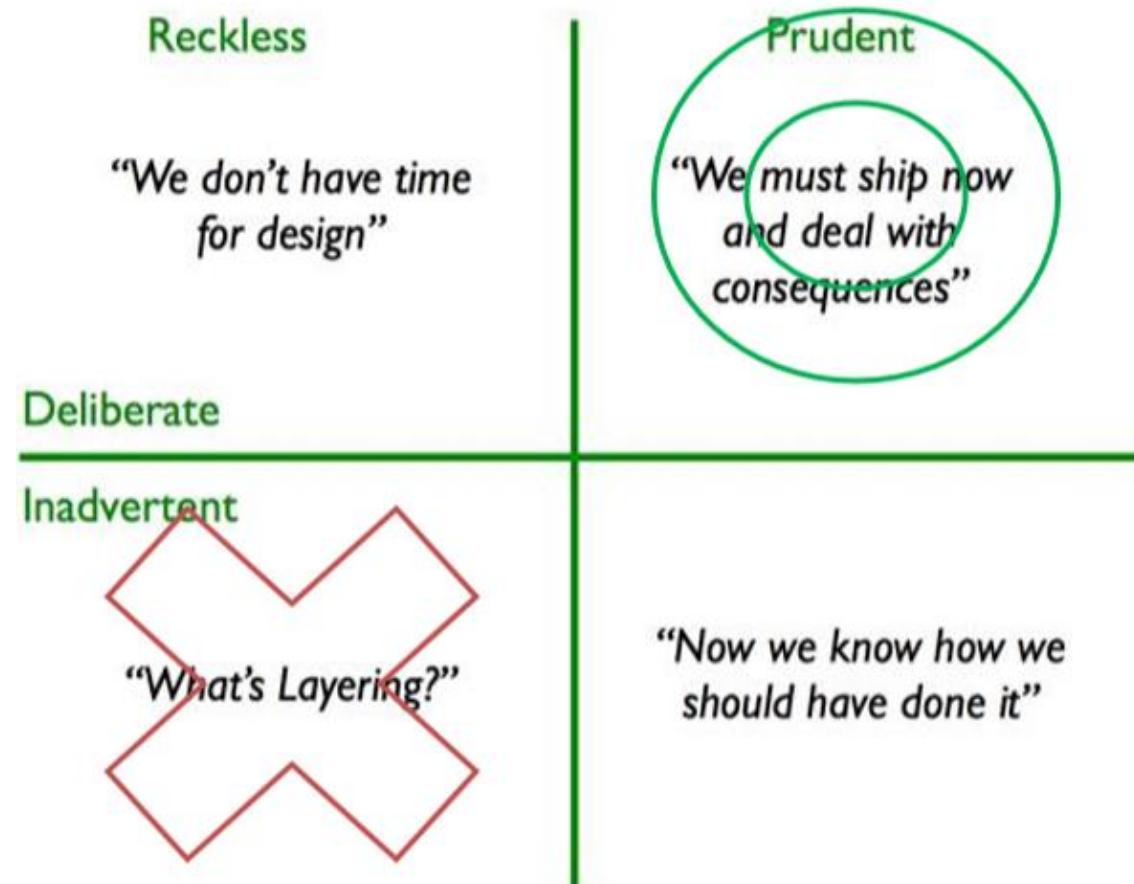
Deuda técnica

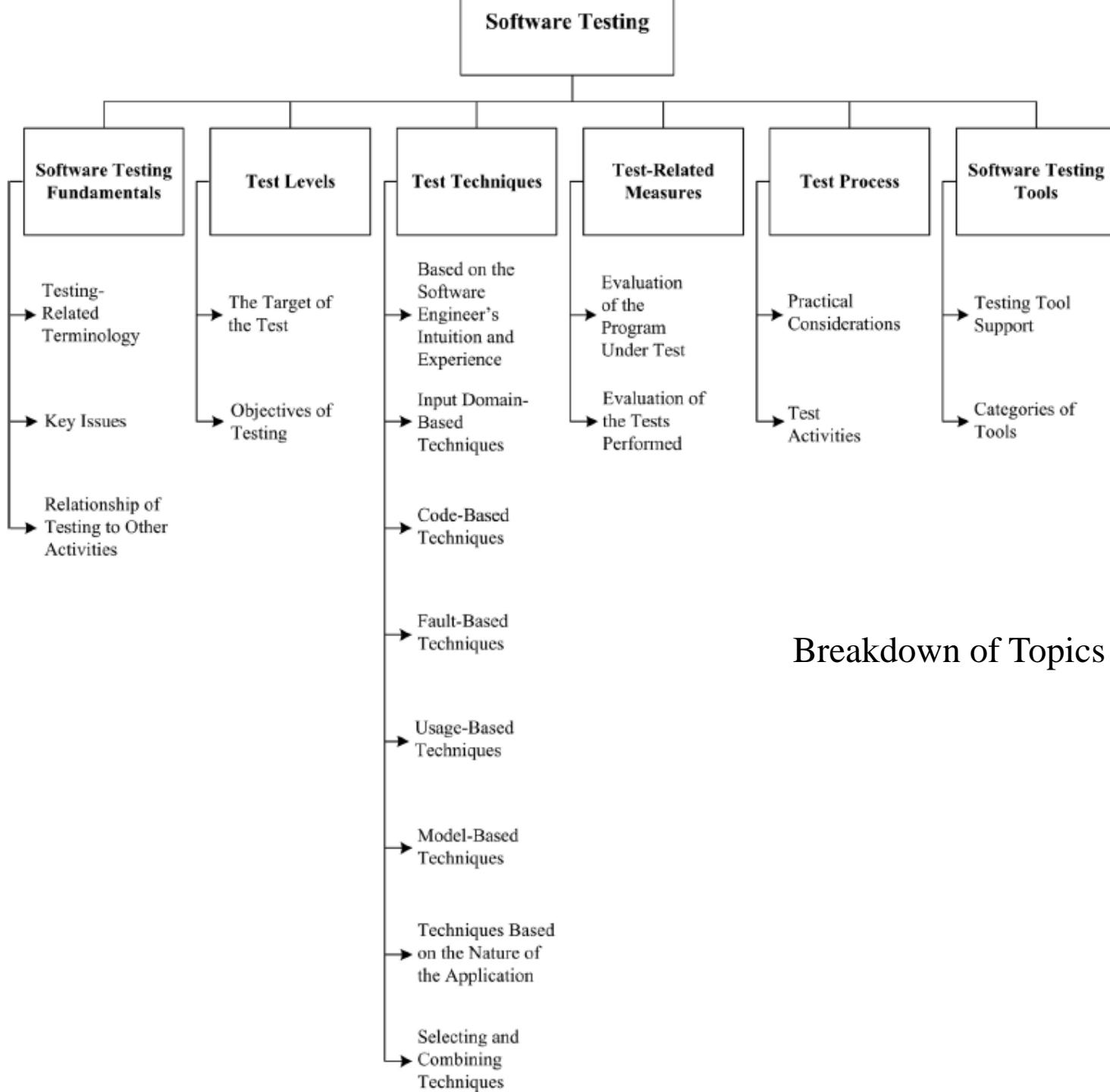
La deuda técnica es el coste y los intereses a pagar por hacer mal las cosas.

La deuda técnica al final siempre alguien la paga. O la paga el proveedor que desarrolla el software o la paga el cliente que lo usa o compra.

La deuda puede ser prudente o impreudente, y también hay una diferencia entre una deuda deliberada y otra inadvertida.

Cuadrante de Fowler:





Breakdown of Topics for the Software Testing KA



Software Testing Methodology in Software Engineering

Actividad

Tarea para la semana

- Definir el grupo de 5 miembros
- Describa el proyecto de software que usará para el curso. Debe estar desarrollado.
- Valide que el software que va a usar para el curso es de una calidad aceptable



UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA



CURSO:
Métodos Formales de Pruebas - Testing de Software

Tema:
Fundamentos de Pruebas de Software

Docente: Mg. Wilder Inga

Software vs. otras industrias

- Muy difícil de interpretar muchas cláusulas para la industria del software
- El desarrollo de software es radicalmente diferente del desarrollo de otros productos.

Software vs. otras industrias

- El Software es intangible
 - Por ello difícil de controlar.
 - Es complicado controla lo que no ves ni sientes.
 - A diferencia de una fábrica de carros:
 - Podemos ver un producto que se está desarrollando a través de etapas como el montaje del motor, el montaje de puertas, etc.
 - Uno puede decir con precisión sobre el estado del producto en cualquier momento.
 - La gestión de un proyecto de software es otro tipo de gestión.

Software vs. otras industrias

- Durante el desarrollo de software:
 - La única materia prima consumida son los datos.
- Para cualquier otro desarrollo de producto:
 - Gran cantidad de materias primas consumidas
 - p.ej. La industria del acero consume grandes volúmenes de mineral de hierro, carbón, piedra caliza, etc.
- Las normas ISO 9000 tienen muchas cláusulas correspondientes al control de materias primas.
 - no relevante para las organizaciones de software.

Software vs. otras industrias

- Existen diferencias radicales entre el desarrollo de software y el de otros productos,
 - Difícil de interpretar varias cláusulas del estándar ISO original en el contexto de la industria del software.

Software Testability

La capacidad de prueba del software se utiliza para medir la facilidad con la que se puede probar un sistema de software:

1. Controlabilidad: el proceso de prueba se puede optimizar solo si podemos controlarlo.
2. Observabilidad: lo que ves es lo que se puede probar. Los factores que afectan el resultado final son visibles.
3. Disponibilidad: Para probar un sistema, tenemos que hacerlo.
4. Simplicidad: cuando el diseño es autoconsistente, las características no son muy complejas y las prácticas de codificación son simples, entonces hay menos que probar. Cuando el software ya no es simple, se vuelve difícil de probar.
5. Estabilidad: si se realizan demasiados cambios en el diseño de vez en cuando, habrá muchas interrupciones en las pruebas de software.
6. Información: la eficacia de las pruebas depende en gran medida de la cantidad de información disponible para el software.

Software Testability

- A Mayor capacidad de prueba significa mejores pruebas y menos cantidad de errores
- A Una menor capacidad de prueba significa que las pruebas no son de gran calidad y existe la posibilidad de que haya más errores en el sistema



Software Testing Methodology in Software Engineering

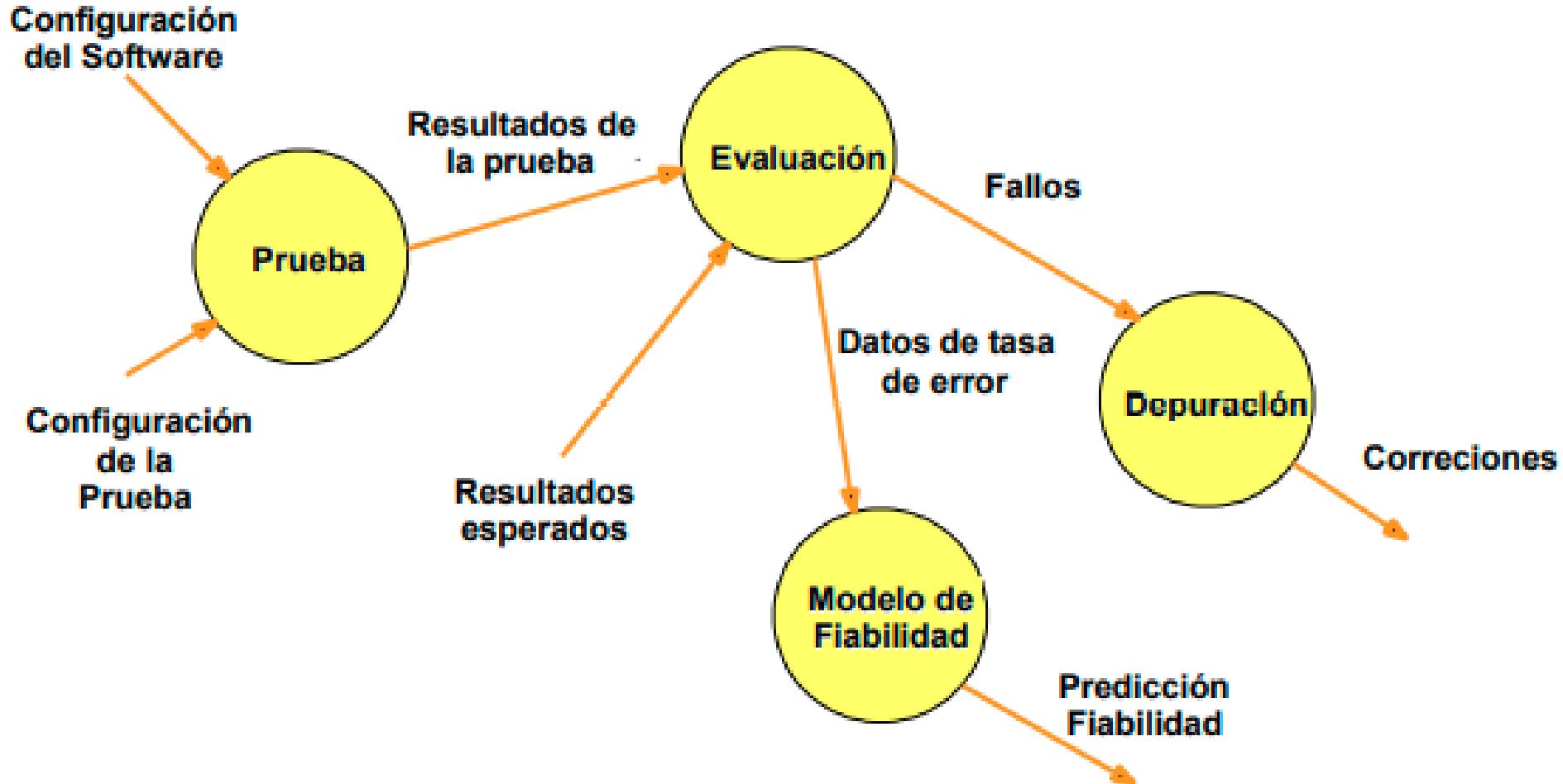


Figura 1. Contexto de la Prueba de Software

7 principios

1. Las pruebas demuestran la presencia de defectos, no su ausencia.
2. Las pruebas exhaustivas no existen o son imposibles.
3. Las pruebas tempranas ahoran tiempo y dinero
4. Agrupación de defectos.
5. Tener cuidado con la paradoja del pesticida.
6. Las pruebas dependen del contexto.
7. Falacia de ausencia de errores.

7 principios

- **1. Las pruebas demuestran la presencia de defectos, no su ausencia.** Nunca podremos decir que nuestros desarrollos están libres de defectos, tenemos un ejemplo en Samsung, uno de sus productos, el Samsung Galaxy Note 7, tuvo que ser retirado del mercado en octubre del 2016 a tan sólo dos meses de su lanzamiento debido a que el dispositivo se incendiaba por sí solo tanto en reposo como en uso, ¿creen ustedes que una compañía con estos niveles de recursos no probaron el dispositivo lo suficiente como para asegurarse de su éxito en el mercado y no arriesgar su reputación?
- Las pruebas demuestran la presencia de defectos pero no pueden asegurarnos que no los hay

7 principios

- **2. Las pruebas exhaustivas no existen o son imposibles.** Nunca podremos probar todo con todas las combinaciones de entradas y precondiciones excepto en casos triviales.
- Un ejemplo, imaginen que tenemos una funcionalidad que incluye completar 10 campos de textos, cada uno de estos campos tiene seis posibles valores que aceptan, así que calculamos que las combinaciones de pruebas sería 10 elevando las 6 eso sería igual a un millón de combinaciones, por esta razón en lugar de intentar realizar pruebas exhaustivas debemos utilizar el análisis de riesgos, las técnicas de pruebas y establecer prioridades para enfocar los esfuerzos dedicados a las pruebas.

7 principios

- **3. Las pruebas tempranas ahorrarán tiempo y dinero.** Las actividades de pruebas deben realizarse lo antes posible en el ciclo de vida de desarrollo del software y esto aplica tanto para las actividades de pruebas unitarias como para las pruebas integrales.
- Durante la realización de pruebas tempranas tratamos de encontrar defectos antes de que éstos pasen a la próxima etapa de desarrollo del software, según una investigación realizada por IBM el costo de eliminación de un defecto aumenta con el tiempo por lo que un defecto encontrado en la etapa de post-producción costaría 30 veces más que si fuera encontrado en la etapa de diseño.
- Realizar pruebas tempranas en el ciclo de vida de desarrollo del software ayuda a reducir o eliminar el costo de los cambios.

7 principios

- **4. Agrupación de defectos.** Usualmente la mayoría de los defectos encontrados durante las pruebas previas al lanzamiento y aquellos defectos responsables de la mayoría de los fallos que ocurren en producción se encuentran en un número reducido de módulos, esta agrupación de defectos en estos módulos puede estar dada porque estos módulos poseen una alta complejidad o porque han sufrido más cambios que el resto y con esto la introducción de nuevos defectos o por otras causas.
- Este fenómeno está muy relacionado con el principio de Pareto o también llamado regla 80-20, que aplicado a este problema podríamos decir que el 80% de los defectos se encuentran en el 20% de los módulos, por lo que si queremos descubrir una mayor cantidad de defectos es útil aplicar este principio y enfocar nuestros esfuerzos en aquellas áreas o módulos donde se haya encontrado una mayor densidad de defectos.

7 principios

- **5. Tener cuidado con la paradoja del pesticida.** Si realizamos las mismas pruebas una y otra vez eventualmente estas pruebas ya no encontrarán nuevos defectos, las pruebas ya no serán eficaces para encontrar defectos de la misma forma que si usamos un mismo pesticida después de un tiempo ya no tendrá efectos sobre los insectos.
- Para detectar nuevos defectos debemos actualizar las pruebas existentes, los datos de las pruebas y además escribir pruebas nuevas.

7 principios

- **6. Las pruebas dependen del contexto.** No probamos de la misma forma el software de un avión de pasajeros que un sitio web que solo provee información.
- Podemos ver que el riesgo es un factor crítico a la hora de definir las pruebas necesarias, mientras más probabilidades hay de pérdidas de vidas humanas o de pérdidas económicas más necesitamos invertir en nuestras pruebas de software.

7 principios

- **7. Falacia de ausencia de errores.** Es una falacia o sea una creencia errónea esperar que encontrando y solucionando un gran número de defectos podemos asegurar el éxito del sistema.
- Por ejemplo aún probando todos los requisitos del sistema a fondo y corrigiendo todos los defectos encontrados podríamos producir un sistema que es difícil de usar que no cumple con las necesidades y expectativas de los usuarios y que además es inferior en correspondencia con el resto de los sistemas de la competencia.

Inspección y Testing :

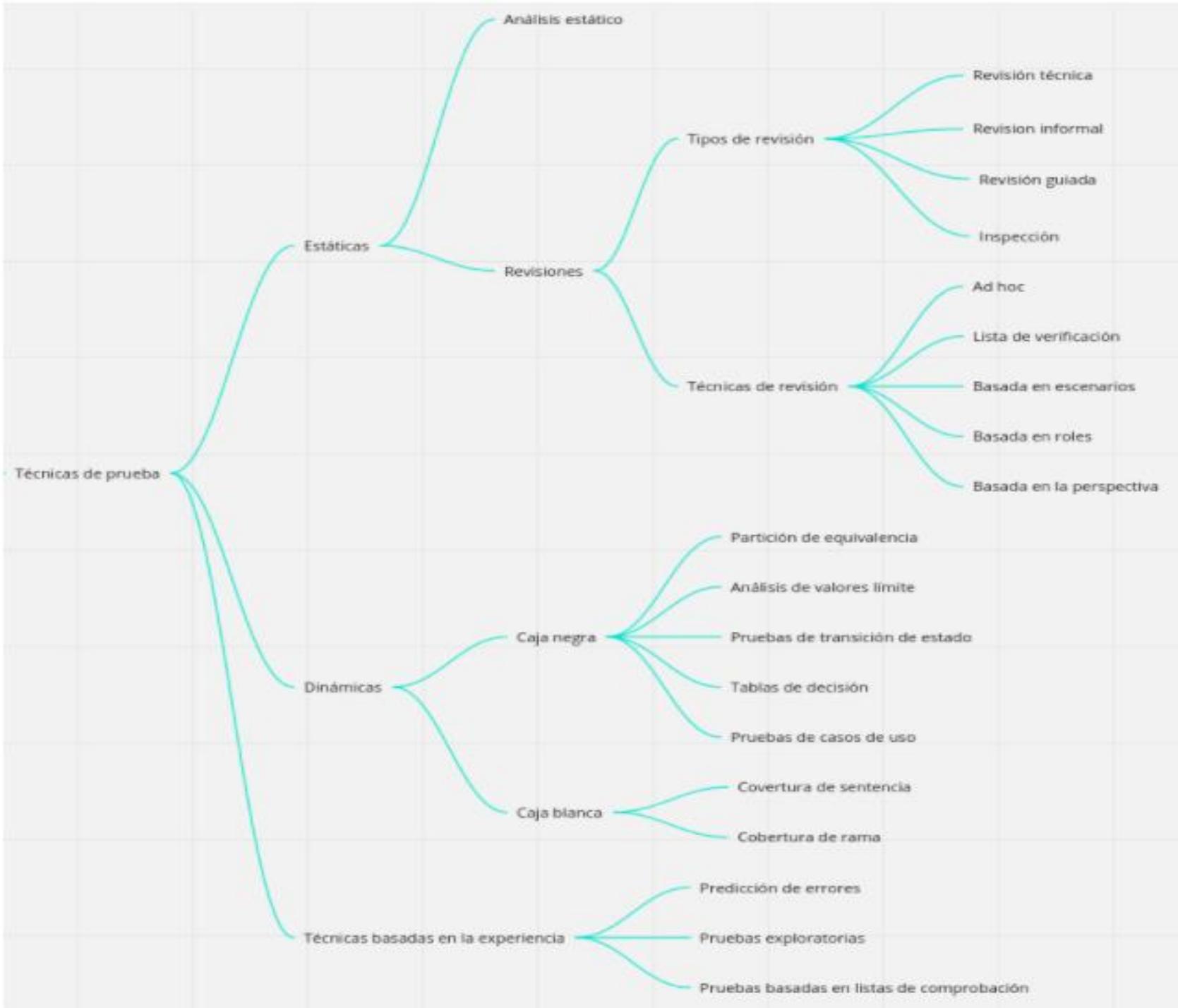
- Se requiere testing efectivo,
 - Unitarias, Integración y sistema.
- Se debe dejar evidencias del testing.

Inspección y Testing :

- Si se utilizan herramientas de Inspección, medición y testing,
 - debe configurarse y calibrarse adecuadamente.

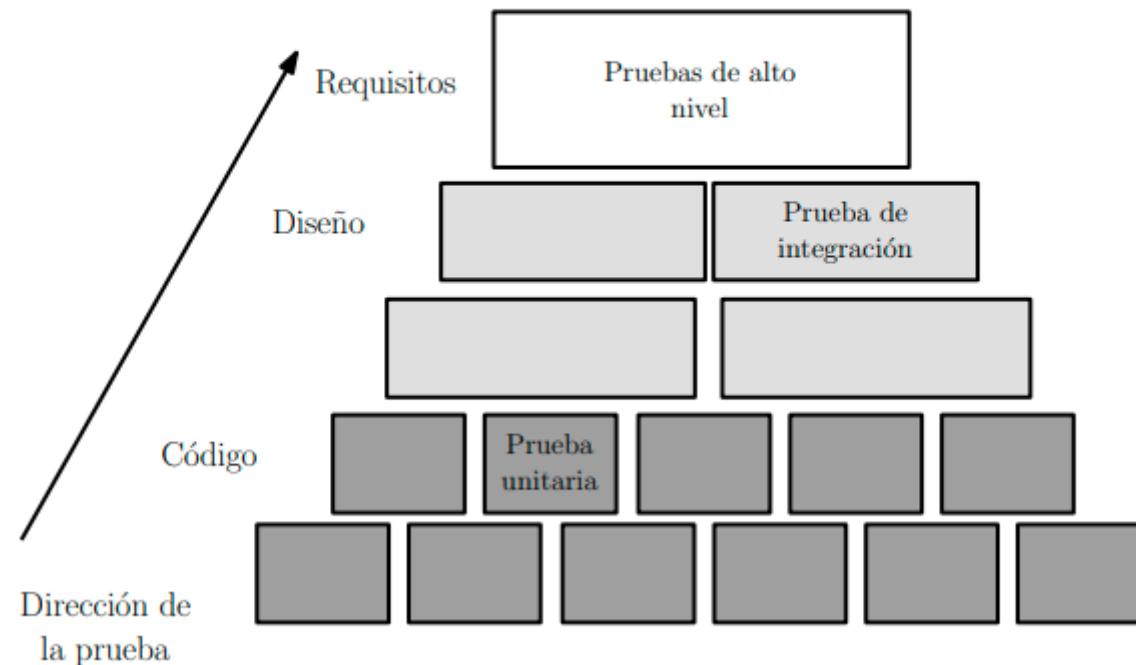
Técnicas de pruebas

- Pruebas estáticas
- Pruebas dinámicas
- Pruebas basadas en la experiencia



Niveles de prueba

Niveles de pruebas

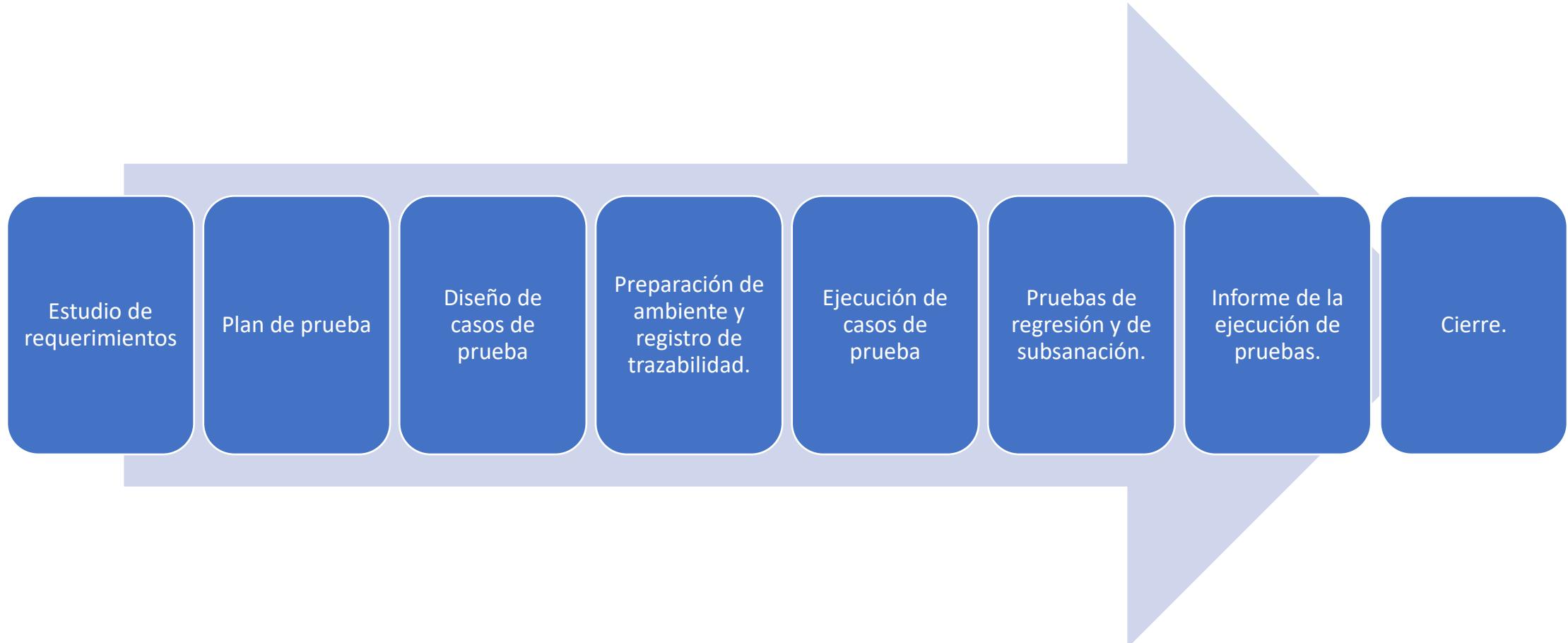


Espectro de los principales niveles y técnicas de pruebas

Testing Level	Opacity	Specification	Who will do this testing?	General Scope
Unit	White Box Testing	Low-Level Design Actual Code structure	Generally Programmers who write code they test	For small unit of code generally no larger than a class
Integration	White & Black Box Testing	Low and High Level Design	Generally Programmers who write code they test	For multiple classes
Functional	Black Box Testing	High Level Design	Independent Testers will Test	For Entire product
System	Black Box Testing	Requirements Analysis phase	Independent Testers will Test	For Entire product in representative environments
Acceptance	Black Box Testing	Requirements Analysis Phase	Customers Side	Entire product in customer's environment
Beta	Black Box Testing	Client Adhoc Request	Customers Side	Entire product in customer's environment
Regression	Black & White Box Testing	Changed Documentation High-Level Design	Generally Programmers or independent Testers	This can be for any of the above

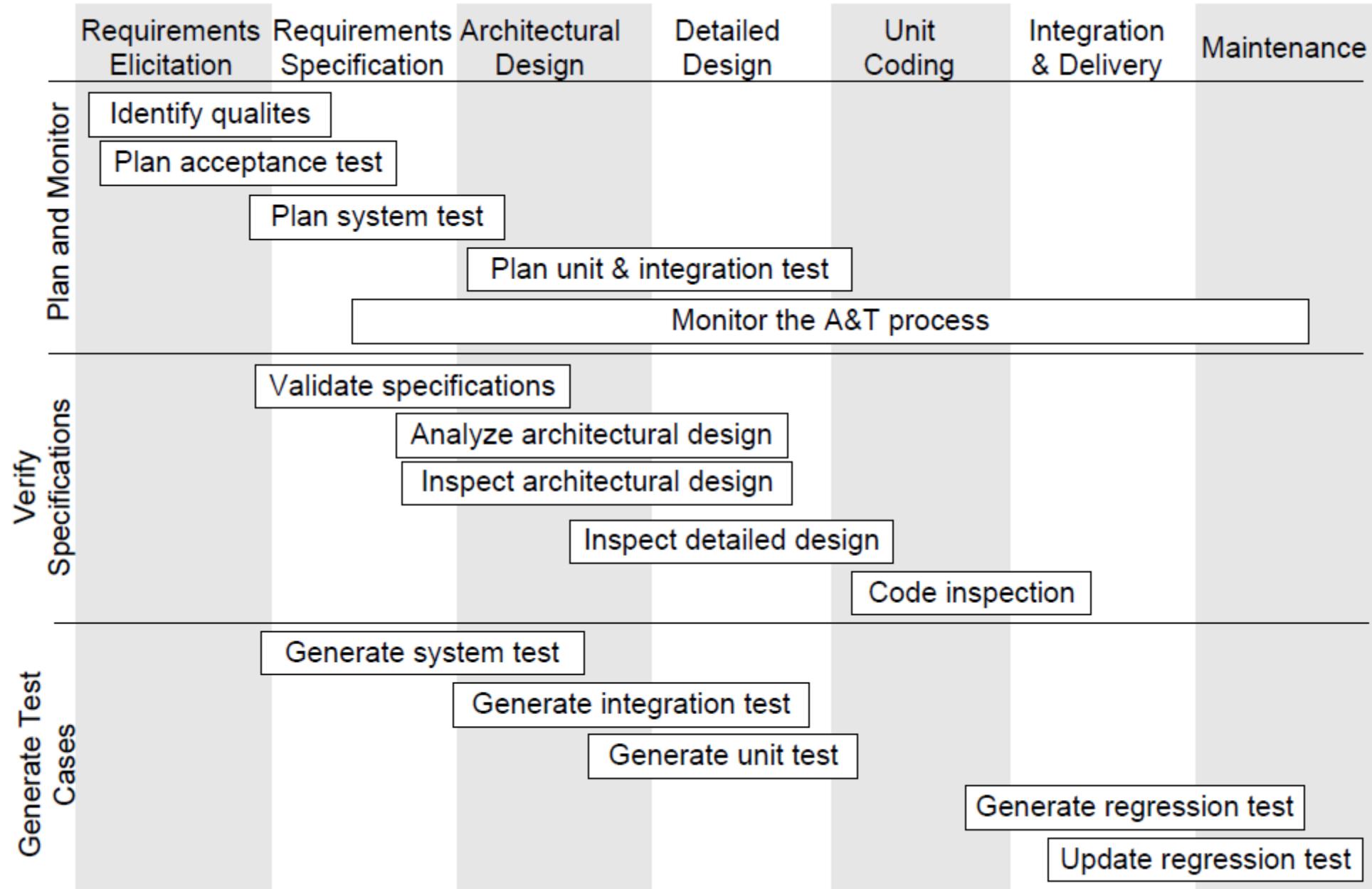
Principales actividades del análisis y testing en el desarrollo de software

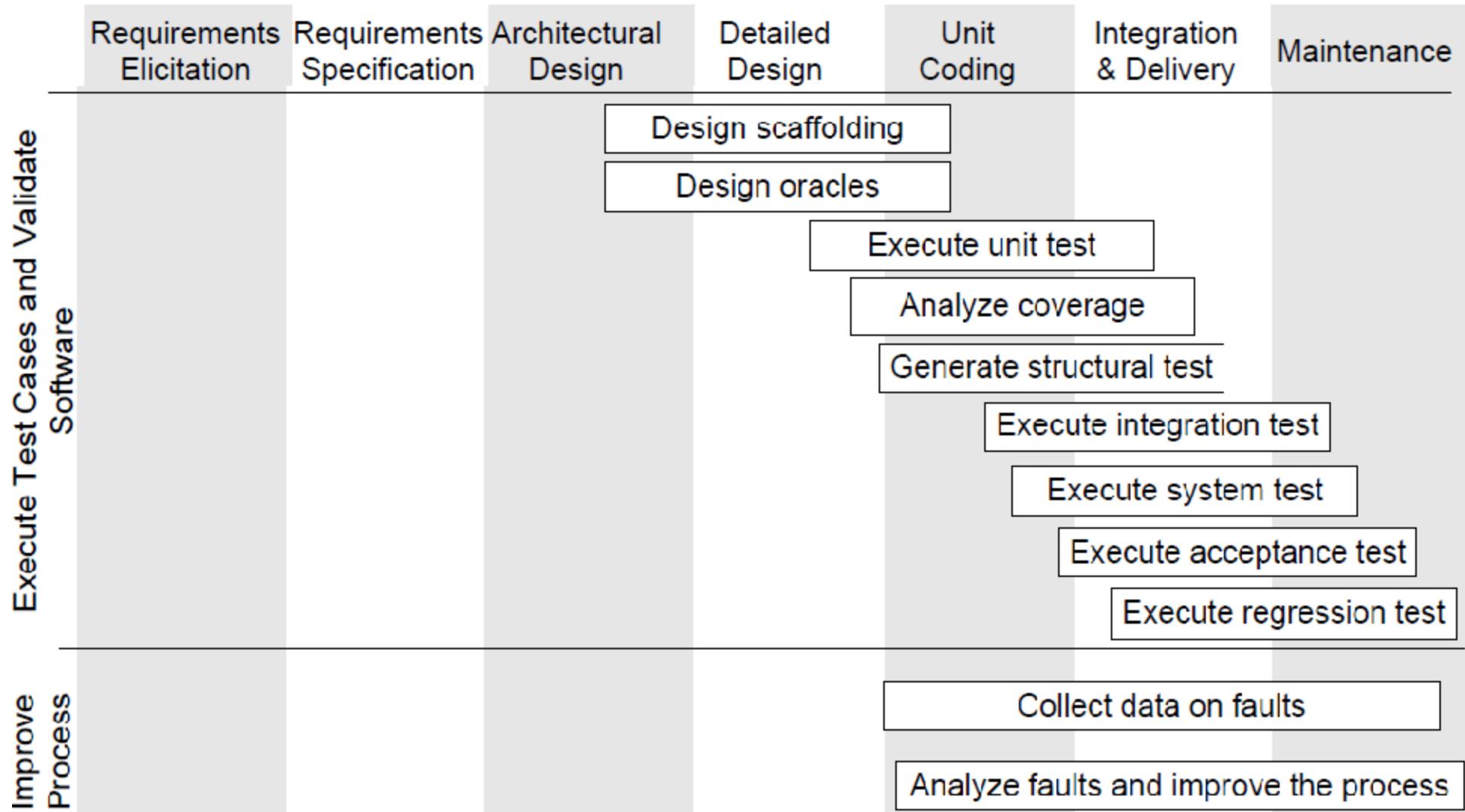
Software Testing Life Cycle



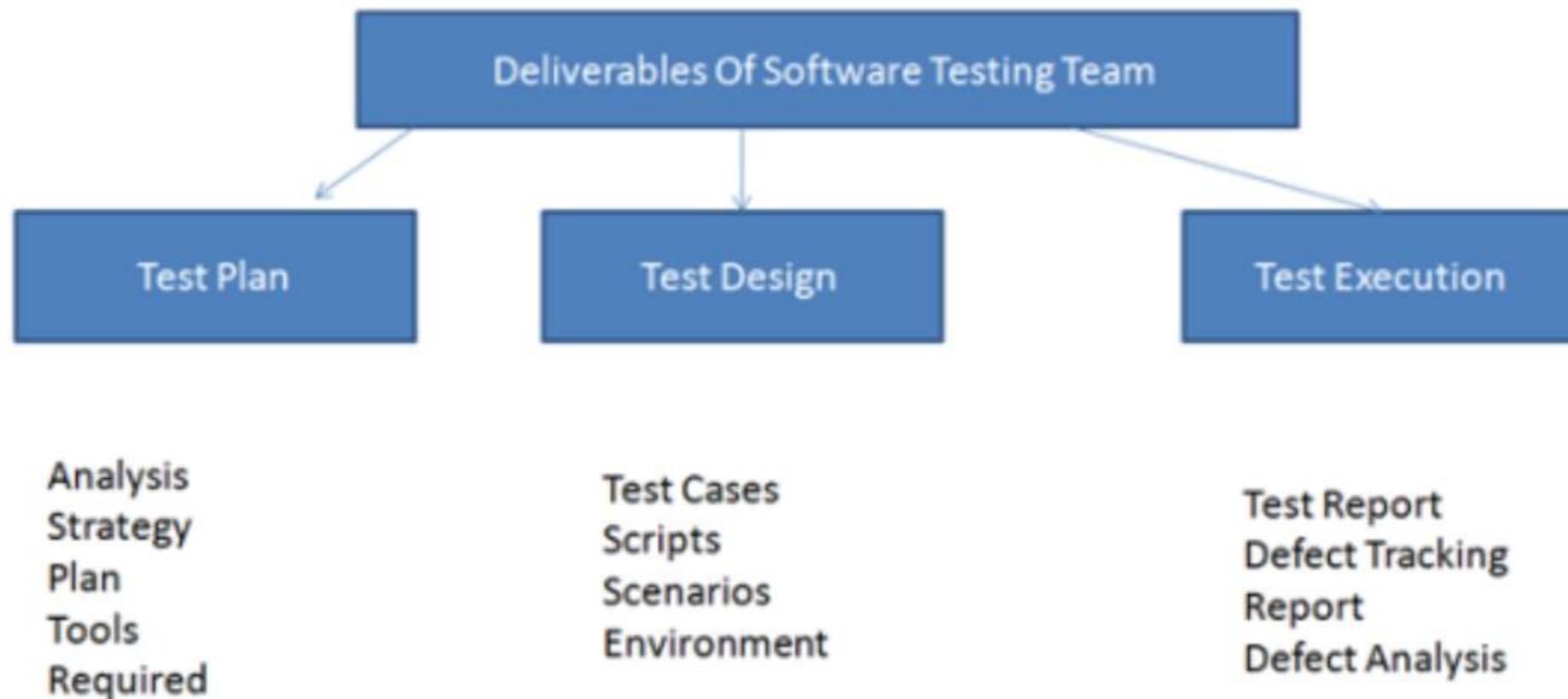
Software Testing Life Cycle

- Estudio de requerimientos: donde se comprende el requerimiento del cliente y se realiza la documentación funcional.
- Plan de prueba: teniendo en cuenta los requisitos del proyecto, se planifican las actividades de prueba y se prepara el documento del plan de prueba.
- Diseño de casos de prueba: los escenarios de prueba se identifican y los casos de prueba se diseñan en consecuencia.
- Se mapean los casos de requisitos y pruebas y se prepara una matriz de trazabilidad de requisitos.
- Ejecución de casos de prueba: se deben ejecutar pruebas y notificar los errores.
- Una vez reparados los defectos, se deben realizar nuevas pruebas y pruebas de regresión.
- Deben prepararse informes de prueba.
- Se completa la actividad de prueba.





Entregables de testing



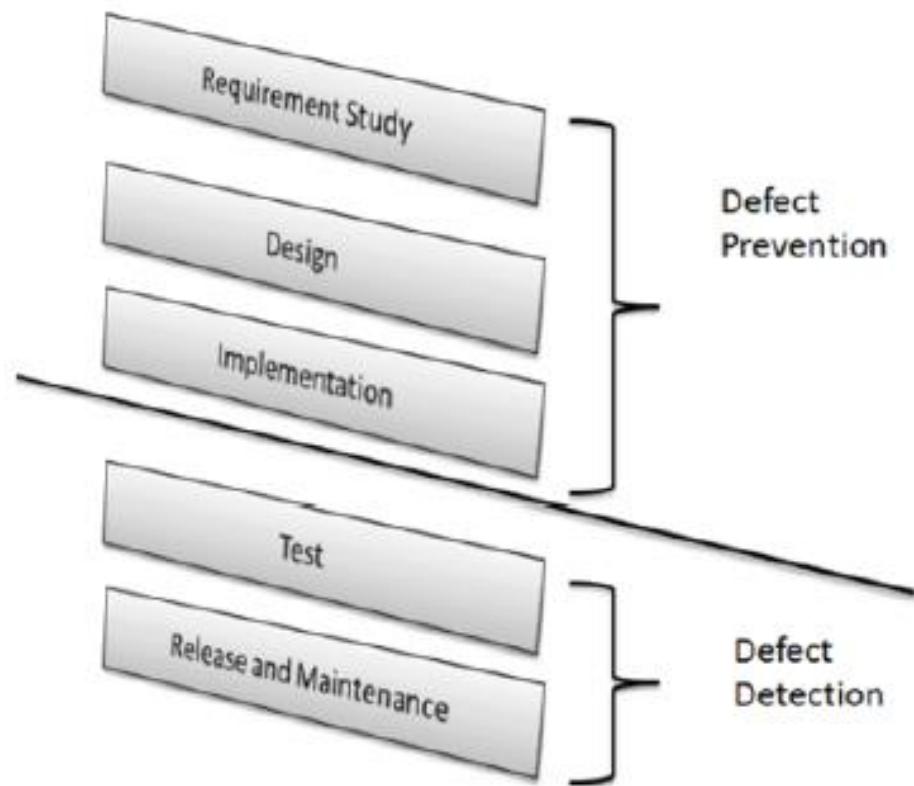
Defectos vs. Fallas



- Un **error** es lo mismo que una equivocación, que por lo general es cometida por una persona, ya sea en el código del software o en algún otro producto de trabajo.
- Cuando se comete un **error**, este introduce un **defecto**. Un **defecto**, es lo mismo que un **problema** o **falta** y también es conocido como **bug** por los desarrolladores.
- Si un fragmento de código que contiene un defecto se ejecuta, es posible que cause un **fallo**, aunque esto no siempre pasa en todas las circunstancias, a veces se requiere de situaciones y datos muy específicos para que ocurra un **fallo**.
- Un fallo detectado se reporta como una incidencia

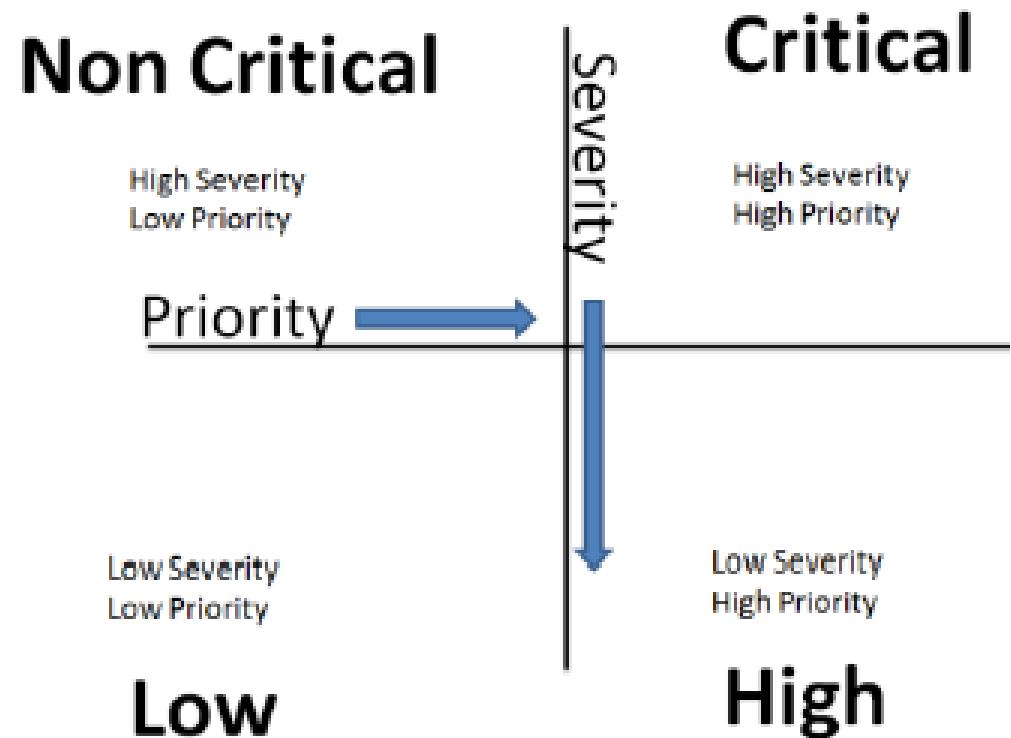
¿Qué es un bug?

Defecto es todo aquello que no se ajusta a las especificaciones de requisitos del software. El defecto generalmente existe en el código del programa o en su diseño. Esto da como resultado una salida incorrecta en el momento de la ejecución del programa.



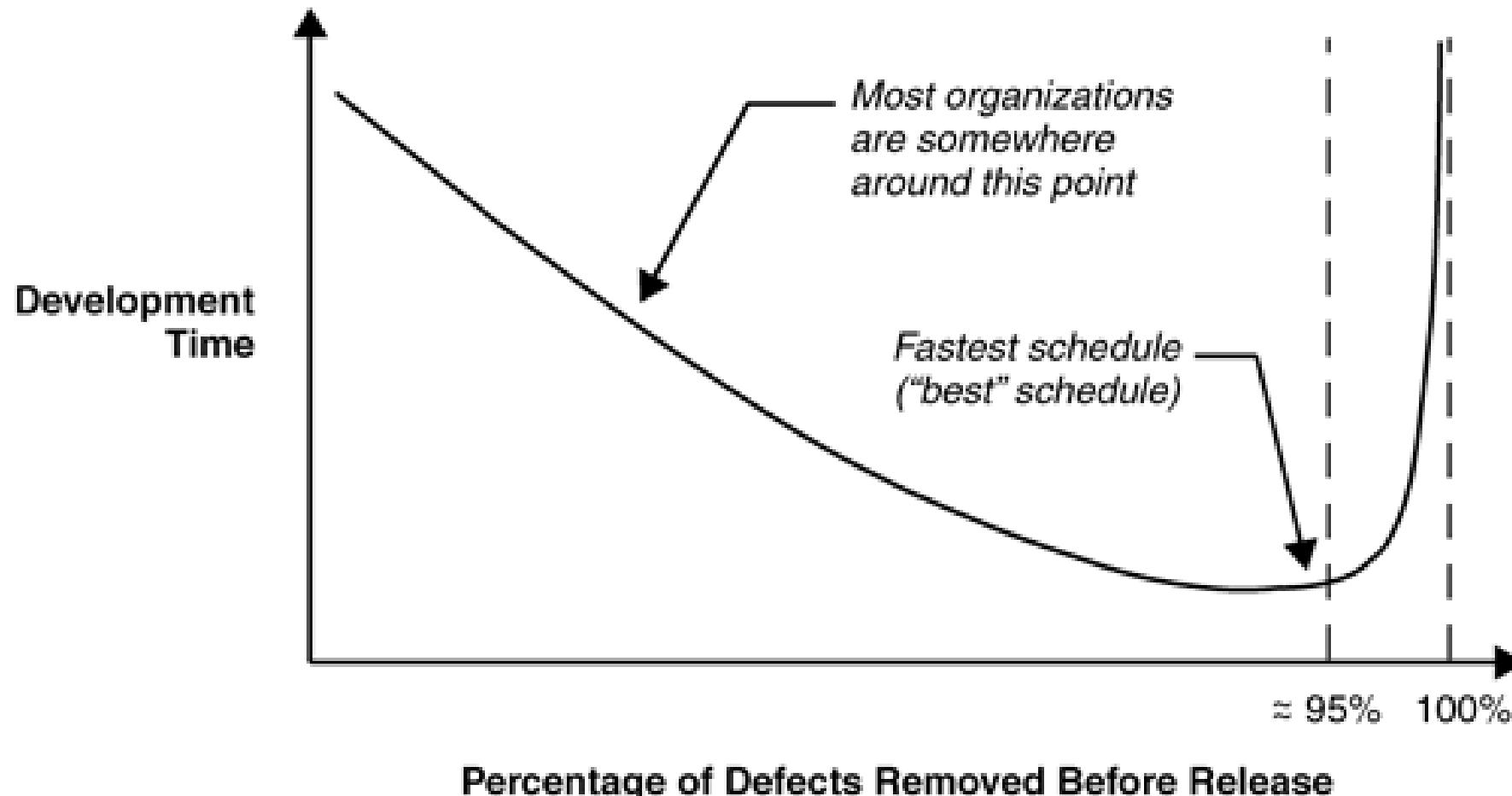
Prioridad, severidad y criticidad

- La severidad define la gravedad del impacto de un defecto/bug en el funcionamiento del sistema.
- La prioridad le dice al desarrollador el orden en el que se deben resolver los defectos.

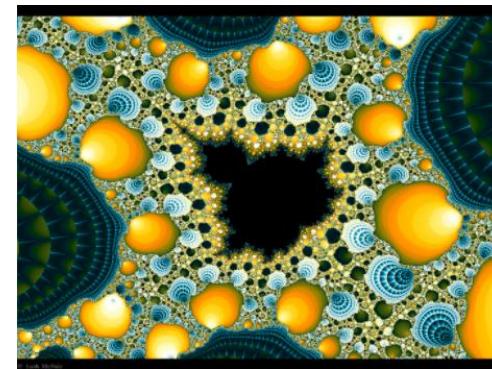


Prioridad, severidad y criticidad

- Si un defecto tiene alta prioridad y alta gravedad, significa que hay un problema en la funcionalidad básica del sistema y el usuario no está en condiciones de usar el sistema. Tales defectos deben subsanarse de inmediato.
- Los defectos que tienen alta prioridad y poca gravedad pueden ser errores de forma importantes (Sintaxis/Opciones etc). Dichos defectos son de baja gravedad pero deben rectificarse inmediatamente y deben considerarse defectos de alta prioridad.
- Defecto de gravedad alta y prioridad baja significa que hay un defecto importante en algún módulo, pero el usuario no lo usa de manera frecuente, por lo que el defecto se puede corregir un poco más tarde.
- Los defectos de baja prioridad y baja gravedad son generalmente de forma (cosmética) y no afectan la funcionalidad del sistema, por lo que dichos defectos se rectifican al final.



Tipos de defectos



- **Heisenbug** es el nombre de uno de estos bichos y tiene su origen en el conocido “Principio de Incertidumbre de Heisenberg”. Heisenberg estableció límites, más allá de los cuales los conceptos de la física clásica no pueden ser empleados. Este principio afirma, por ejemplo, que no se puede determinar simultáneamente la posición y la cantidad de movimiento de una partícula. A veces se expresa esto como que el mismo acto de observar un experimento altera los resultados. Los programadores utilizan el término “heisenbug” para denominar a los errores que desaparecen o alteran su comportamiento al tratar de depurarlos. Esto ocurre porque cuando se intenta encontrar un error dentro de un programa se suele utilizar alguna herramienta -otro programa- o un estado de memoria diferente al habitual, lo que hace que el entorno en que se ejecuta el software bajo prueba no sea el mismo y el error desaparezca como por arte de magia, o “mute”, provocando efectos diferentes.

Tipos de defectos



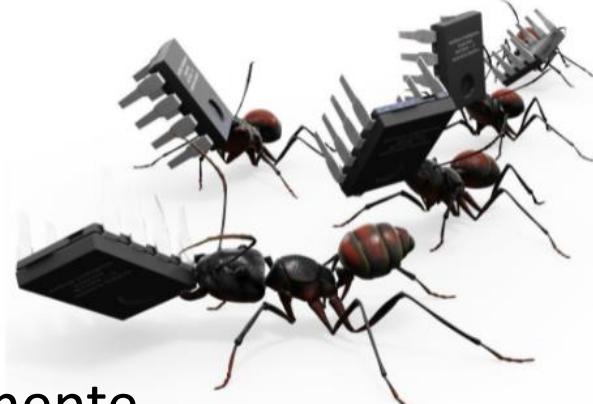
- Los “**Bohrbug**”, denominados así por el modelo atómico de Bohr, es un tipo de error que se encuentra en las antípodas del anterior. Los informáticos utilizan esta denominación para aquellos errores que, no importa lo que se haga, mantienen un comportamiento constante. Otra variedad bastante frecuente es el “**Mandelbug**”, fallos con causas tan complejas que su comportamiento parece ser completamente caótico. La denominación se debe al conocido conjunto fractal descubierto por Benoit Mandelbrot, un monstruo matemático de enorme complejidad. Si estás programando, seguramente no querrás encontrarte con uno de estos.

Tipos de defectos



- Los “**Schroedinbugs**” parecen sacados de una novela de ciencia ficción. Son errores que no aparecen hasta que alguien lee el código y descubre que, en determinadas circunstancias, el programa podría fallar. A partir de ese momento, el maldito “Schroedinbug” comienza aparecer una y otra vez. Parece que algo así no puede existir, sin embargo, cualquiera que haya participado de un desarrollo más o menos grande seguramente ha tenido que lidiar con él. Su nombre se relaciona, por supuesto, con el físico Erwin Rudolf Josef Alexander Schrödinger y su famosa paradoja conocida como “paradoja del gato de Schrödinger”. En este caso, el error no se manifiesta hasta que el observador no sabe que está allí. Raro. Muy raro. Pero real.

Tipos de defectos



- El zoológico informático tiene más criaturas. Algunos no son realmente problemas del software, pero lo parecen. Los programadores llaman “**stole**” al problema que se produce cuando luego de introducir datos que aparentan ser correctos (pero no lo son) se obtiene una (lógica) salida incorrecta. El problema, por supuesto, se encuentra en los datos introducidos, pero como el programador está convencido que estos son correctos, suele comenzar a reescribir partes de su programa tratando de eliminar un bug que, en realidad, no existe.
- El nombre deriva de Aristóteles (Aristotle), de quien mucha gente asumía que debía estar siempre en lo cierto y no cuestionaba sus ideas. Más extraños aun son los denominados “Phase of the Moon bug” (o bugs de fase lunar), errores que parecen depender de factores aleatorios y que la mente del programador los atribuye a los motivos más esotéricos. “El programa falla cuando José está presente”, o “solo falla cuando la luna está en cuarto creciente”. Al igual que todos los anteriores, este bug es muy frecuente.

Tipos de defectos



- El último tipo de bug es quizás uno de los más comunes. Denominado **“fantasma en el código”**, suele esconderse en esas rutinas o subprogramas que rara vez se ejecutan. Su ubicación los hace muy difíciles de identificar durante las pruebas previas al lanzamiento del programa, y puede hacer que un producto fracase estrepitosamente al ser puesto a la venta. Aunque parezca extraño que algo así pueda ocurrir, basta con recordar el problema que tenía el microcódigo de los primeros microprocesadores Intel Pentium, que en determinadas condiciones arrojaban resultados erróneos al dividir dos números. Ese bug logró sobrevivir a todas las pruebas, y explotó cuando el chip ya estaba en la calle.

Actividad



UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA



CURSO:
Métodos Formales de Pruebas - Testing de Software

Tema:
Fundamentos de Pruebas de Software
Semana 3

Docente: Mg. Wilder Inga

Testing Software



Las compañías de software se enfrentan con serios desafíos cada vez más grandes en la prueba de sus productos debido a que cada día **se incrementa la complejidad del software**.

Lo primero y más importante que hay que hacer es reconocer la naturaleza compleja de las pruebas y tomarlas en serio: contratar a las personas más inteligentes que se pueda encontrar, ayudarlas a conseguir y/o brindarles las herramientas y el entrenamiento que requieran para aprender su oficio, y escucharlos cuando hablan acerca de la **calidad del software**. Ignorarlos podría ser el error más costoso para la empresa y el producto.

Testing Software



Los investigadores en pruebas se enfrentan igualmente a estos desafíos. Las compañías están ansiosas por financiar buenas ideas de investigación, pero la demanda fuerte es por más práctica experimental y menos trabajo académico.

Las cuatro fases que estructuran la metodología propuesta en este capítulo no se deben considerar como definitivas y únicas, porque este es un campo en constante desarrollo e investigación en el que están involucrados muchos investigadores y empresas.

EL PROCESOS DE PRUEBA

- Al planificar y ejecutar las pruebas los probadores de software deben considerar: el software y su función de cálculo, las entradas y cómo se pueden combinar, y el entorno en el que el software eventualmente funcionará.
- Este difícil proceso requiere tiempo, sofisticación técnica y una adecuada planificación. Los probadores no solo deben tener buenas habilidades de desarrollo (a menudo las pruebas requieren una amplia cantidad de código), sino también conocimientos en lenguajes formales, teoría de grafos, lógica computacional y algoritmia.



EL PROCESOS DE PRUEBA

Para tener una visión más clara acerca de las dificultades inherentes a las pruebas, es necesario acercarse a ellas a través de la aplicación de cuatro fases:

1. Modelar el entorno del software
2. Seleccionar escenarios de prueba
3. Ejecutar y evaluar los escenarios
4. Medir el progreso de las pruebas

1: Modelar el entorno del software

La tarea del probador es simular la interacción entre el software y su entorno para lo que debe identificar y simular las interfaces que utiliza el sistema. Las interfaces más comunes son:

- Humanas: Interfaz gráfica de usuario GUI, clics, pulsaciones de teclado y entradas desde otros dispositivos.
- De software: APIs, WS, sistema operativo, la base de datos o las librerías, en tiempo de ejecución. Comprobar, no solo las probables, sino también las inesperadas. Por ejemplo, todos los desarrolladores esperan que el sistema operativo guarde los archivos por ellos, pero olvidan que les puede informar que el medio de almacenamiento está lleno, por lo que, incluso, los mensajes de error deben probarse.
- Del sistema de archivos, datos, formato, contenido.
- Las interfaces de comunicación, acceso directo a dispositivos físicos, controladores (IoT), protocolos válidos e inválidos

1: Modelar el entorno del software

Cada entorno único de aplicación puede resultar en un número significativo de interacciones de usuario que se debe probar.

Situaciones que los probadores deben abordar son:

- Usando el sistema operativo un usuario elimina un archivo que otro usuario tenía abierto, ¿qué pasará la próxima vez que el software intente accesar ese archivo?
- Un dispositivo se reinicia en medio de un proceso de comunicación, ¿podrá el software darse cuenta de esto y reaccionar adecuadamente, o simplemente lo dejará pasar?
- Dos sistemas compiten por duplicar servicios desde la API, ¿podrá la API atender correctamente ambos servicios?

1: Modelar el entorno del software

Cuando una interfaz presenta problemas de tamaño o de complejidad infinitos:

- Se suele usar la técnica Boundary Value Partitioning, para seleccionar valores individuales para las variables, en o alrededor de sus fronteras. Por ejemplo, probar los valores máximos, mínimos y cero para un entero con signo es una prueba común, lo mismo que los valores que rodean cada una de estas particiones, por ejemplo, 1 y -1 que rodean la frontera cero. Los valores entre las fronteras se tratan como el mismo número: utilizar 16 o 16.000 no hace ninguna diferencia para el software bajo prueba.

elegir los valores para múltiples variables procesadas simultáneamente y que potencialmente podrían afectar a otras

- Al decidir cómo será la secuencia de entrada, los probadores tienen un problema de generación de secuencia, por lo que deben tratar cada entrada física y cada evento abstracto como símbolos en el alfabeto de un lenguaje formal, definir un modelo de ese lenguaje que les permita visualizar el posible conjunto de pruebas, e indagar cómo se ajusta cada una a la prueba general.

2: Seleccionar escenarios de prueba

Muchos modelos de dominio y particiones de variables representan un número infinito de escenarios de prueba, cada uno de los cuales cuesta tiempo y dinero

Solo un sub-conjunto de ellos se puede aplicar en cualquier programa de desarrollo de software realista, así que ¿cómo hace un probador inteligente para seleccionar ese sub-conjunto? ¿17 es mejor valor de entrada que 34? ¿cuántas veces se debe seleccionar un filename antes de pulsar el botón Open?

los probadores prefieren una respuesta que se refiera a la cobertura de código fuente, o a su dominio de entrada, y se orientan por la cobertura de las declaraciones de código (ejecutar cada línea de código fuente por lo menos una vez), o la cobertura de entradas (aplicar cada evento generado externamente).

En cuanto al código, no son las declaraciones individuales las que interesan a los probadores, sino los caminos de ejecución: secuencias de declaraciones de código que representan un camino de ejecución del software, pero, desafortunadamente, existe un número infinito de caminos.

2: Seleccionar escenarios de prueba

Las pruebas se organizan desde dichos conjuntos infinitos hasta lograr, lo mejor posible, los criterios adecuados de datos de prueba; que se utilizan adecuada y económicamente para representar cualquiera de esos conjuntos

Por ejemplo las pruebas se organizan desde dichos conjuntos infinitos hasta lograr, lo mejor posible, los criterios adecuados de datos de prueba; que se utilizan adecuada y económicamente para representar cualquiera de esos conjuntos

2: Seleccionar escenarios de prueba

Criterios de prueba de los caminos de ejecución

Los criterios adecuados para datos de prueba se concentran en la cobertura de caminos de ejecución o en la cobertura de secuencias de entrada, pero rara vez en ambos. El criterio de selección de caminos de ejecución más común es el de aquellos que cubran las estructuras de control. Por ejemplo: 1) seleccionar un conjunto de casos de prueba que garantice que cada sentencia se ejecute al menos una vez, y 2) seleccionar un conjunto de casos de prueba que garantice que cada estructura de control If, Case, While, ... se evalúe en cada uno de sus posibles caminos de ejecución.

2: Seleccionar escenarios de prueba

Criterios de prueba del dominio de entrada

El criterio para el rango de cobertura del dominio abarca desde la cobertura de una interfaz sencilla hasta la medición estadística más compleja:

- Elegir un conjunto de casos de prueba que contenga cada entrada física.
- Seleccionar un conjunto de casos de prueba que garantice que se recorra cada interfaz de control.

2: Seleccionar escenarios de prueba

Criterios de prueba del dominio de entrada

El criterio de discriminación requiere una selección aleatoria de secuencias de entrada hasta que, estadísticamente, representen todo el dominio infinito de las entradas:

- Seleccionar un conjunto de casos de prueba que tenga las mismas propiedades estadísticas que el dominio de entrada completo.
- Escoger un conjunto de rutas que puedan ser ejecutadas por un usuario típico.

3: Ejecutar y evaluar los escenarios

Debido a que los escenarios de prueba se ejecutan manualmente constituye un trabajo intensivo y, por tanto, propenso a errores, por lo que los probadores deben tratar de automatizarlos tanto como sea posible.

En muchos entornos es posible aplicar automáticamente las entradas a través del código que simula la acción de los usuarios, y existe herramientas que ayudan a este objetivo. Pero la automatización completa requiere la simulación de cada fuente de entrada, y del destino de la salida de todo el entorno operacional.

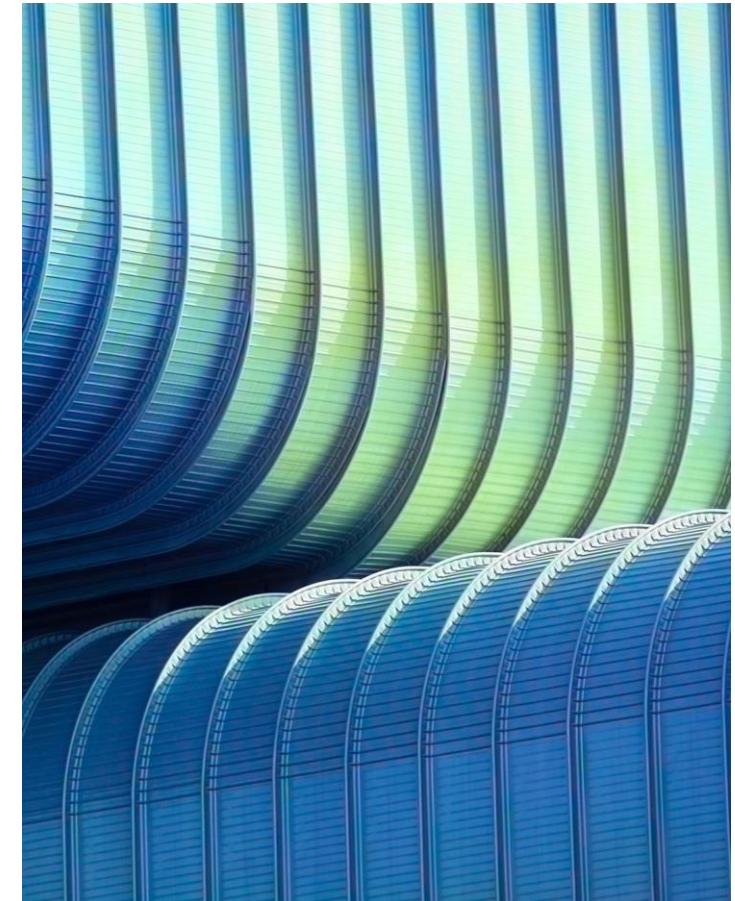
A menudo, los probadores incluyen código para recoger datos en el entorno simulado, como ganchos o seguros de prueba, con los que recogen información acerca de las variables internas, las propiedades del objeto, y otros. Esto ayuda a identificar anomalías y a aislar errores. Estos ganchos se eliminan cuando el software se entrega.



3: Ejecutar y evaluar los escenarios

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar, pero difícil de ejecutar porque es menos automatizada. La evaluación implica la comparación de las salidas reales del software, resultado de la ejecución de los escenarios de prueba, con las salidas esperadas, tal y como están documentadas en la especificación, que se supone correcta, ya que las desviaciones son errores.

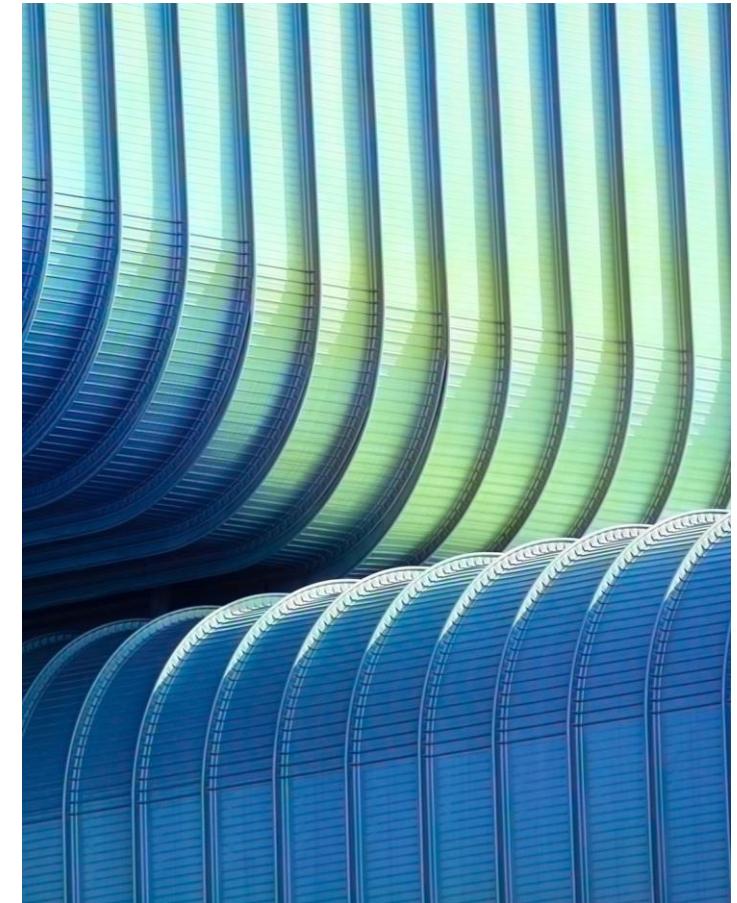
Teóricamente, la comparación (para determinar la equivalencia) de dos funciones arbitrarias computables es irresoluble. Volviendo al ejemplo del editor de texto: si la salida se supone que es resaltar una palabra mal escrita, ¿cómo se puede determinar que se ha detectado cada instancia de errores ortográficos? Tal dificultad es la razón por la que la comparación de la salida real versus la esperada se realiza generalmente por un oráculo humano: un probador que monitorea visualmente la pantalla de salida y analiza cuidadosamente los datos que aparecen.



3: Ejecutar y evaluar los escenarios

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar, pero difícil de ejecutar porque es menos automatizada. La evaluación implica la comparación de las salidas reales del software, resultado de la ejecución de los escenarios de prueba, con las salidas esperadas, tal y como están documentadas en la especificación, que se supone correcta, ya que las desviaciones son errores.

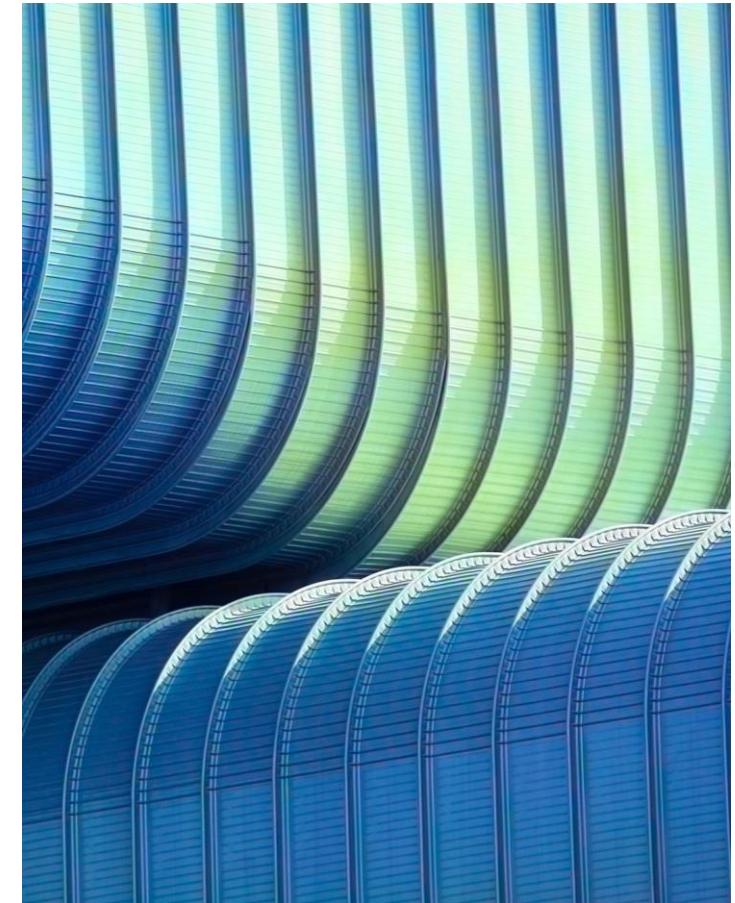
Teóricamente, la comparación (para determinar la equivalencia) de dos funciones arbitrarias computables es irresoluble. Volviendo al ejemplo del editor de texto: si la salida se supone que es resaltar una palabra mal escrita, ¿cómo se puede determinar que se ha detectado cada instancia de errores ortográficos? Tal dificultad es la razón por la que la comparación de la salida real versus la esperada se realiza generalmente por un oráculo humano: un probador que monitorea visualmente la pantalla de salida y analiza cuidadosamente los datos que aparecen.



3: Ejecutar y evaluar los escenarios

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar, pero difícil de ejecutar porque es menos automatizada. La evaluación implica la comparación de las salidas reales del software, resultado de la ejecución de los escenarios de prueba, con las salidas esperadas, tal y como están documentadas en la especificación, que se supone correcta, ya que las desviaciones son errores.

Teóricamente, la comparación (para determinar la equivalencia) de dos funciones arbitrarias computables es irresoluble. Volviendo al ejemplo del editor de texto: si la salida se supone que es resaltar una palabra mal escrita, ¿cómo se puede determinar que se ha detectado cada instancia de errores ortográficos? Tal dificultad es la razón por la que la comparación de la salida real versus la esperada se realiza generalmente por un oráculo humano: un probador que monitorea visualmente la pantalla de salida y analiza cuidadosamente los datos que aparecen.



3: Ejecutar y evaluar los escenarios

Enfoques para evaluar las pruebas

- La formalización consiste en formalizar el proceso de escritura de las especificaciones y la forma cómo, desde ellas, se derivan el diseño y el código. Tanto el desarrollo orientado por objetos, como el estructurado, tienen mecanismos para expresar formalmente las especificaciones, de forma que se simplifique la tarea de comparar el comportamiento real y el esperado.
- Esencialmente, existe dos tipos de código de prueba embebido:
 1. El más simple es el código de prueba que expone algunos objetos de datos internos, o estados, de tal forma que un oráculo externo pueda juzgar su correctitud más fácilmente. Al implementarse, dicha funcionalidad es invisible para los usuarios. Los probadores pueden tener acceso a resultados del código de prueba a través de, por ejemplo, una prueba al API o a un depurador.
 2. Un tipo más complejo de código embebido tiene características de programa de auto-prueba: a veces se trata de soluciones de codificación múltiple para el problema, para chequear o escribir rutinas inversas que deshacen cada operación. Si se realiza una operación y luego se deshace, el estado resultante debe ser equivalente al estado pre-operacional.

3: Ejecutar y evaluar los escenarios

Pruebas de regresión

Después de que los probadores presentan con éxito los errores encontrados, generalmente los desarrolladores crean una nueva versión del software, en la que, supuestamente esos errores se han eliminado. La prueba progres a través de versiones posteriores del software hasta una que se selecciona para entregar. La pregunta es: ¿cuántas re-pruebas, llamadas pruebas de regresión, se necesitan en la versión n, cuando se re-utilizan las pruebas ejecutadas sobre la versión n-1?

3: Ejecutar y evaluar los escenarios

Pruebas de regresión

Cualquier selección puede:

- 1) corregir solo el problema que fue reportado,
- 2) fallar al corregir el problema reportado,
- 3) corregir el problema reportado, pero interrumpir un proceso que antes trabajaba, o
- 4) fallar al corregir el problema reportado e interrumpir un proceso funcional.

Teniendo en cuenta estas posibilidades sería prudente volver a ejecutar todas las pruebas de la versión n-1 en la versión n, antes de probar nuevamente, aunque esta práctica generalmente tiene un costo elevado.

Las nuevas versiones del software a menudo vienen con características y funcionalidades nuevas, además de los errores corregidos, así que las pruebas de regresión les quitarían tiempo a las pruebas del código nuevo.

3: Ejecutar y evaluar los escenarios

Adicionalmente:

Se debería codificar de tal forma que pueda verificarse y validarse adecuadamente.

Dado que los errores los identifican los probadores, pero los diagnostican los desarrolladores, puede suceder:

- 1) Que su reproducción fracase, o
- 2) Que el escenario de prueba se ejecute nuevamente.

Volver a efectuar una prueba no garantiza que se reproduzcan las mismas condiciones originales. Re-ejecutar un escenario requiere conocer con exactitud el estado del sistema operativo y cualquier software que lo acompañe, condiciones del entorno, etc.

Hay que conocer el estado de automatización de la prueba, los dispositivos periféricos y cualquiera otra aplicación de segundo plano, que se ejecute localmente o través de la red y que podría afectar la aplicación bajo prueba.



4: Medir el progreso de las pruebas

¿cuál es el estado de sus pruebas?

- en la práctica, medir las pruebas consiste en contar cosas: el número de entradas aplicadas, el porcentaje de código cubierto, el número de veces que se ha invocado la aplicación, el número de veces que se ha terminado la aplicación con éxito, el número de errores encontrados, y así sucesivamente.
- La interpretación de estas cuentas es difícil: ¿encontrar un montón de errores es buena o mala noticia? ¿un alto número de errores significa que la prueba se ejecutó completamente y que persisten muy pocos errores?; o ¿el software tiene un montón de errores y que, a pesar de que muchos fueron encontrados, otros permanecen ocultos?.

4: Medir el progreso de las pruebas

Los valores de estos conteos pueden dar muy pocas luces acerca de los avances de las pruebas, y muchos probadores alteran estos datos para dar respuesta a las preguntas, con lo que determinan la completitud estructural y funcional de lo que han hecho. Por ejemplo, para comprobar la completitud estructural los probadores pueden hacerse preguntas como:

- ¿He probado para errores de programación común?
- ¿He ejercitado todo el código fuente?
- ¿He forzado a todos los datos internos a ser inicializados y utilizados?
- ¿He encontrado todos los errores sembrados?

Y para probar la completitud funcional:

- ¿He tenido en cuenta todas las formas en las que el software puede fallar y he seleccionado casos de prueba que las muestren y casos que no lo hagan?
- ¿He aplicado todas las posibles entradas?
- ¿Tengo completamente explorado el dominio de los estados del software?
- ¿He ejecutado todos los escenarios que espero que un usuario ejecute?

4: Medir el progreso de las pruebas

Capacidad de prueba

la idea de que el número de líneas de código determine la dificultad de la prueba es obsoleta, la cuestión es mucho más complicada y es donde entra en juego la capacidad de prueba. Si un producto software tiene alta capacidad de prueba:

- 1) Será más fácil de probar y, por consiguiente, más fácil de encontrar sus errores, y
- 2) Será posible monitorear las pruebas, por lo que los errores y disminuyen las probabilidades de que se queden otros sin descubrir.

Una baja capacidad de prueba requerirá muchas más pruebas para llegar a estas mismas conclusiones, y es de esperar que sea más difícil encontrar errores.

4: Medir el progreso de las pruebas

Modelos de fiabilidad

¿Cuánto durará el software en ejecución antes de que falle? ¿Cuánto costará el mantenimiento del software? Sin duda es mejor encontrar respuestas a estas preguntas mientras todavía se tenga el software en el laboratorio de pruebas. Desde un punto de vista funcional los modelos de fiabilidad (modelos matemáticos de escenarios de prueba y datos de errores) están bien establecidos. Con base en cómo se comportó durante las pruebas, estos modelos pretenden predecir cómo se comportará el software en su entorno funcional. Para lograrlo la mayoría de ellos requieren la especificación de un perfil operativo: una descripción de cómo se espera que los usuarios apliquen las entradas.

4: Medir el progreso de las pruebas

Modelos de fiabilidad

Para calcular la probabilidad de una falla estos modelos hacen algunas suposiciones acerca de la distribución de probabilidades subyacentes que regulan las ocurrencias de las anomalías. Tanto los investigadores como los probadores expresan escepticismo acerca de que se pueda ensamblar adecuadamente estos perfiles.

Errores que llegan al usuario

Los desarrolladores conocen la frustración cuando reciben reportes de errores de parte de los usuarios. Cuando esto sucede inevitablemente se preguntan: ¿cómo escaparon esos errores a las pruebas?

Se invirtieron incontables horas en el examen cuidadoso de cientos o miles de variables y sentencias de código, ¿cómo puede un error eludir esta vigilancia?

- 1. El usuario ejecuta un código no probado.** Por falta de tiempo, no es raro que los desarrolladores liberen código sin probar, en el que los usuarios pueden encontrar anomalías.
- 2. El orden en que se ejecuta las declaraciones en el ambiente de uso difiere del que se utilizó durante la prueba.** Este orden puede determinar si el software funciona bien o no.

Errores que llegan al usuario

3. **El usuario aplica una combinación de valores de entrada no probados.** Las posibles combinaciones de valores de entrada, que miles de usuarios pueden hacer a través de una interfaz de software, simplemente son numerosas para que los probadores las apliquen todas y, como deben tomar decisiones difíciles acerca de qué valores de entrada probar, a veces toman las equivocadas.
4. **El entorno operativo de usuario nunca se probó.** Es posible que los probadores tengan conocimiento de dicho entorno, pero que no cuenten con el tiempo suficiente para probarlo. Tal vez no replicaron la combinación de hardware, periféricos, sistemas operativos y aplicaciones del entorno del usuario en el laboratorio de pruebas. Por ejemplo, aunque es poco probable que las empresas que escriben software creen redes de miles de nodos en su laboratorio de pruebas, los usuarios sí lo hacen y, de hecho, lo hacen en sus entornos reales.

Actividad



UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA



CURSO:
Métodos Formales de Pruebas - Testing de Software

Tema:
Métodos formales en las Pruebas de Software
Semana 4

Docente: Mg. Wilder Inga

Los métodos formales

- Los métodos formales permiten al ingeniero de software especificar, desarrollar y verificar un sistema informático mediante la aplicación de una notación matemática rigurosa.
- Utilizando un lenguaje de especificación formal, un método formal proporciona los medios de especificar un sistema de forma que se aseguren, de forma sistemática, la consistencia, la completitud y la corrección.
- Se suelen basar en notaciones matemáticas similares a las del álgebra de conjuntos y la lógica

Propósitos de los métodos formales

- Se introducir rigor en todas las fases de desarrollo de software, con lo que es posible evitar que se pasen por alto cuestiones críticas;
- Proporcionar un método estándar de trabajo a lo largo del proyecto; y
- Constituir una base de coherencia entre las muchas actividades relacionadas y, al contar con mecanismos de descripción precisos y no ambiguos, proporcionar el conocimiento necesario para realizarlas con éxito.

Los Métodos Formales En La Ingeniería Del Software

Conceptos

El concepto de los métodos formales involucra una serie de técnicas lógicas y matemáticas con las que es posible especificar, diseñar, implementar y verificar los sistemas de información.

La importancia de los métodos formales en la Ingeniería del Software se incrementó en el siglo XXI: se desarrollaron nuevos lenguajes y herramientas para especificar y modelar formalmente, y se diseñaron metodologías maduras para verificar y validar

Desde hace varias décadas se utiliza técnicas de notación formal para modelar los requisitos, principalmente porque estas notaciones se pueden verificar fácilmente y porque, de cierta forma, son más comprensibles para el usuario final

Propósitos de los métodos formales

1. Sistematizar e introducir rigor en todas las fases de desarrollo de software, con lo que es posible evitar que se pasen por alto cuestiones críticas;
2. Proporcionar un método estándar de trabajo a lo largo del proyecto; y
3. Constituir una base de coherencia entre las muchas actividades relacionadas y, al contar con mecanismos de descripción precisos y no ambiguos, proporcionar el conocimiento necesario para realizarlas con éxito.

Uso de los métodos formales

1. Las políticas de los requisitos. En un sistema seguro se convierten en las principales propiedades de seguridad que éste debe conservar, es decir, el modelo de políticas de seguridad formal, como confidencialidad o integridad de datos.
2. La especificación. Es una descripción matemática basada en el comportamiento del sistema, que utiliza tablas de estado o lógica matemática. No describe normalmente al software de bajo nivel, pero sí su respuesta a eventos y entradas, de tal forma que es posible establecer sus propiedades críticas.
3. Las pruebas de correspondencia entre la especificación y los requisitos. Es necesario demostrar que el sistema, tal como se describe en la especificación, establece y conserva las propiedades de las políticas de los requisitos. Si están en notación formal se puede diseñar pruebas rigurosas manuales o automáticas.
4. Las pruebas de correspondencia entre el código fuente y la especificación. Aunque muchas técnicas formales se crearon inicialmente para probar la correctitud del código, pocas veces se logra debido al tiempo y costo implicados, pero pueden aplicarse a los componentes críticos del sistema.
5. Pruebas de correspondencia entre el código máquina y el código fuente. Este tipo de pruebas raramente se aplica debido al costo y a la alta confiabilidad de los compiladores modernos.

Principales Métodos Formales utilizados en el desarrollo de software

- Métodos formales basados en Lógica de Primer Orden: Z, B, VDM, Object-Z, Z++ y VDM++.
- Métodos formales basados en Formalismos Algebraicos: HOSA (Hidden Order Sorted Algebras), TROLL, OBLOG, Maude y AS-IS (Algebraic Specifications with Implicit States).
- Métodos formales basados en Redes de Petri: CO-OPN (Concurrent Object-Oriented Petri Nets)
- Métodos formales basados en Lógica Temporal: TRIO, OO-LTL y ATOM.
- Métodos Semiformales: Syntropy, Statemate, UML y OCL (Object Constraint Language)

Definition 1. A finite automaton M is defined by a 5-tuple $(\Sigma, Q, q_0, F, \delta)$, where

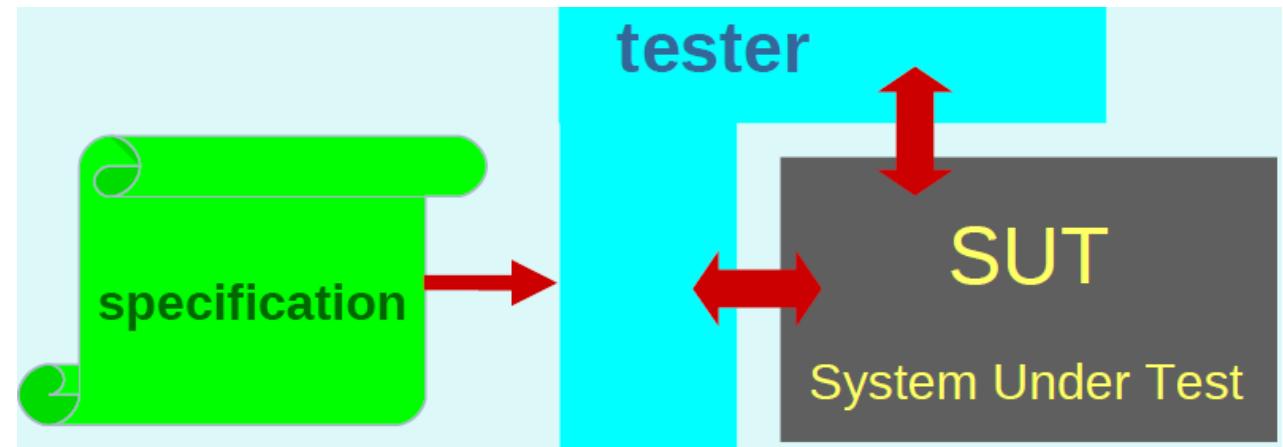
- Σ is the set of symbols representing input to M
- Q is the set of states of M
- $q_0 \in Q$ is the start state of M
- $F \subseteq Q$ is the set of final states of M
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function

Especificación en lógica formal de las funciones de insertar y borrar elementos en una estructura pila

- **L.push(e:stelement)**: Inserta e después del último elemento insertado
 - **PRE**: $\exists L \wedge L \neq \{\text{NULL}\} \wedge L = \{\text{PRF}\} \wedge \text{tamaño}(L) = n \wedge \neg \text{existe}(L, \text{clave}(e))$
 - **POST**: $L = \{\text{PRF}, e\} \wedge \text{tamaño}(L) = n + 1$
 - **PRE**: $\exists L \wedge L = \{\text{NULL}\}$
 - **POST**: $L = \{e\} \wedge n = 1$
- **L.pop()**: borra el elemento recientemente insertado
 - **PRE**: $\exists L \wedge L = \{\text{PRF}, e\} \wedge \text{tamaño}(L) = n$
 - **POST**: $L = \{\text{PRF}\} \wedge \text{tamaño}(L) = n - 1$

Testing

Comprobar o medir algunas características de calidad de un objeto en ejecución mediante la realización de experimentos de forma controlada



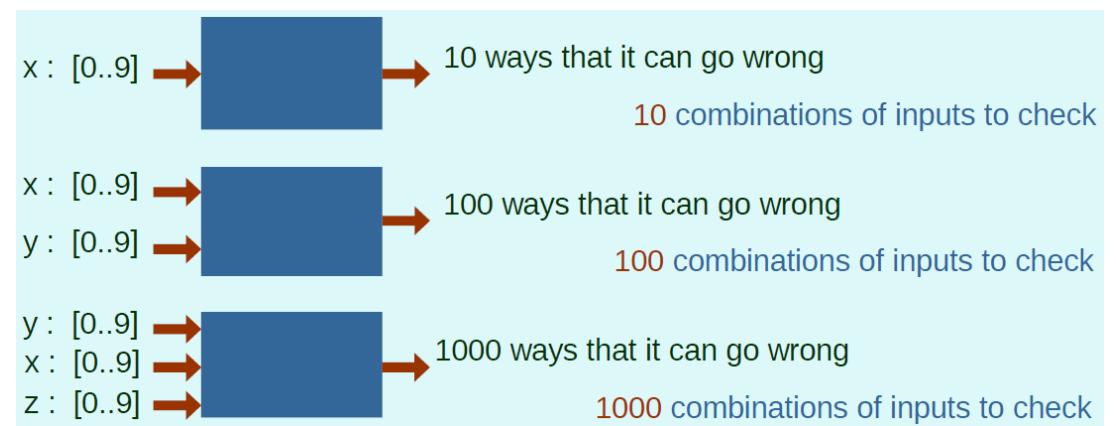
¿Por qué Model-Based Testing?

- Aumento de la complejidad y búsquedas
 - El esfuerzo de prueba crece exponencialmente con el tamaño del sistema
 - Las pruebas no pueden seguir el ritmo del desarrollo
- Más abstracción
 - Menos detalle
 - Desarrollo basado en modelos; UML, MDA, Simulink/Matlab de OMG
- Comprobando la calidad
 - En la práctica: pruebas - ad hoc, demasiado tarde, caras, mucho tiempo
 - Investigación: verificación formal - pruebas, verificación de modelos, . . . con un impacto práctico decepcionante

Software bugs / errors cost US economy yearly:
\$ 59.500.000.000 (www.nist.gov)
\$ 22 billion could be eliminated...

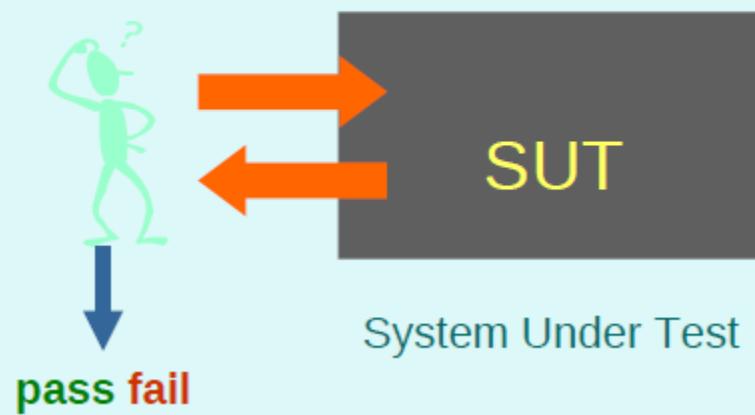
Testing Complexity

- El aumento de la complejidad y el tamaño del esfuerzo de prueba de sistemas crece exponencialmente con el tamaño del sistema. Las pruebas no pueden seguir el ritmo del desarrollo.
- Es necesaria la automatización de las pruebas. Las pruebas basadas en modelos son una de las técnicas

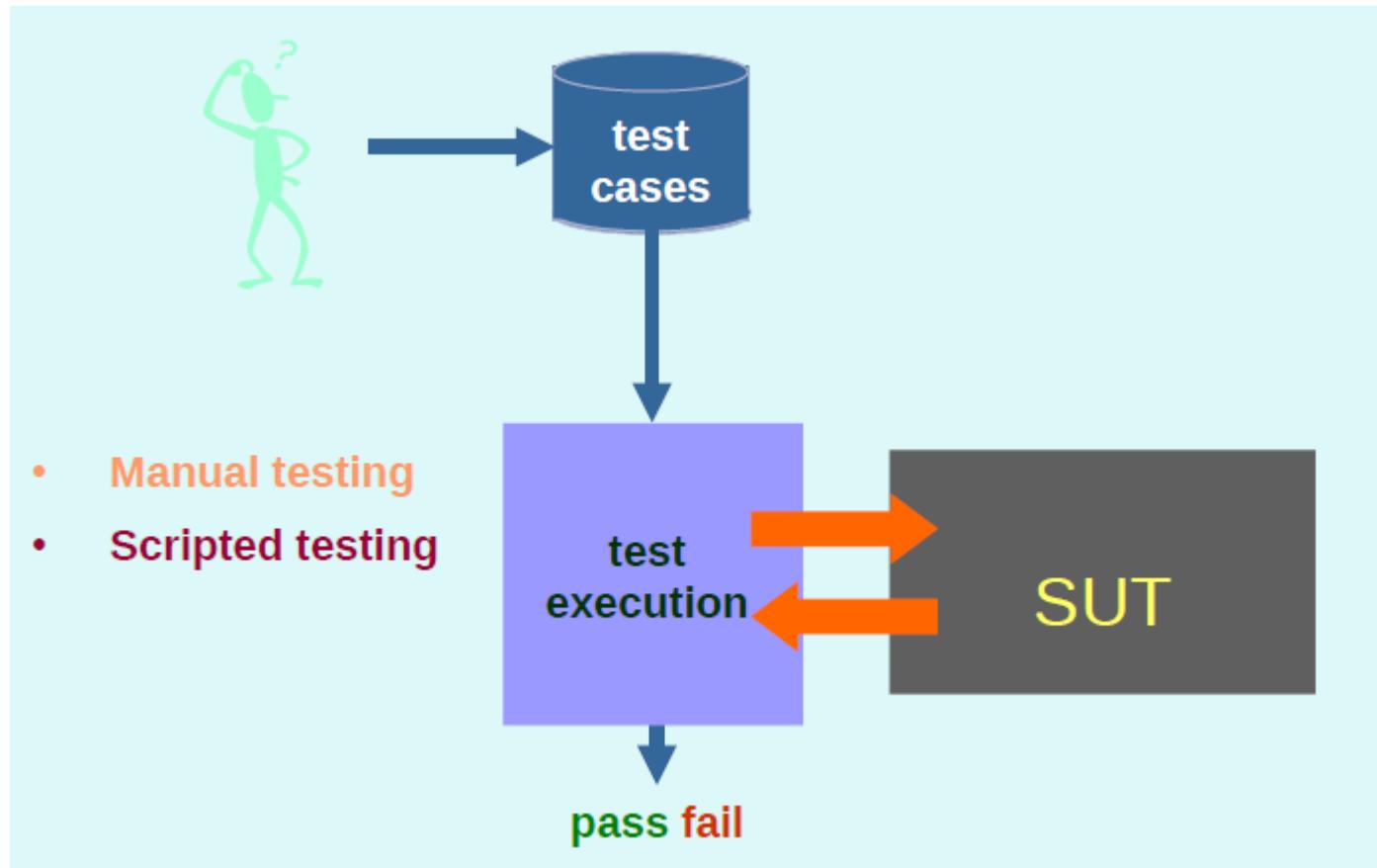


Developments in Testing 1

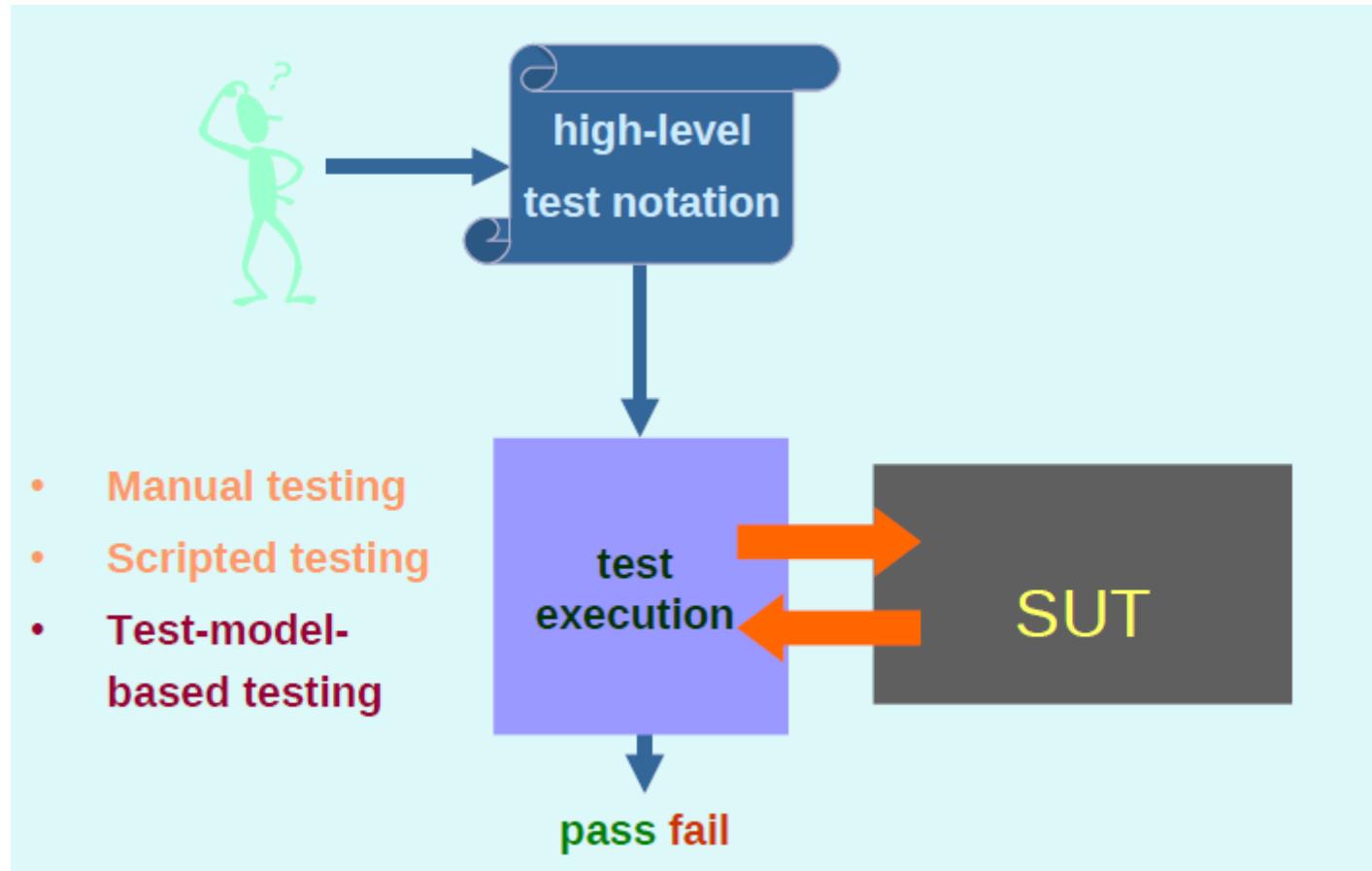
1. Manual testing



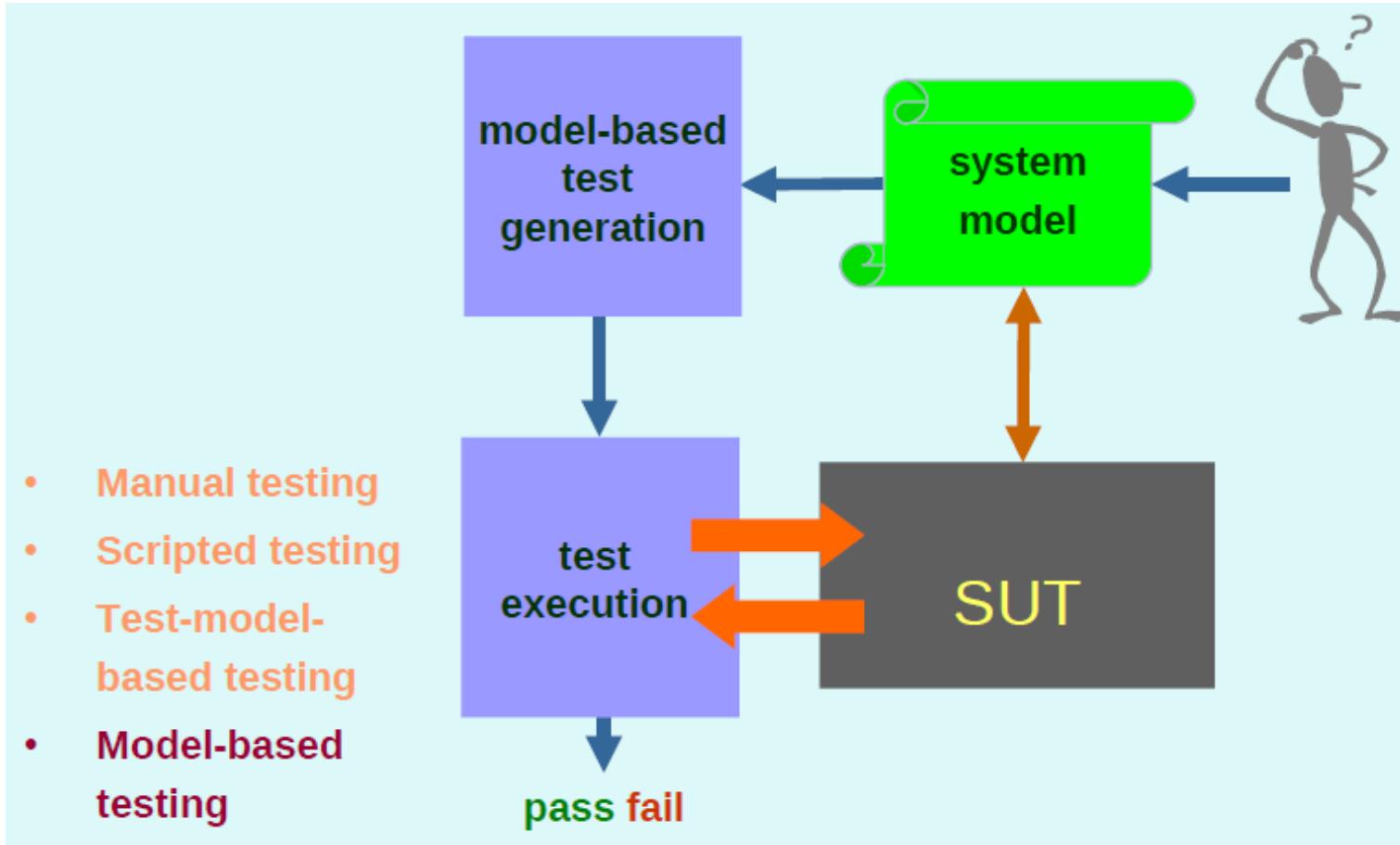
Developments in Testing 2



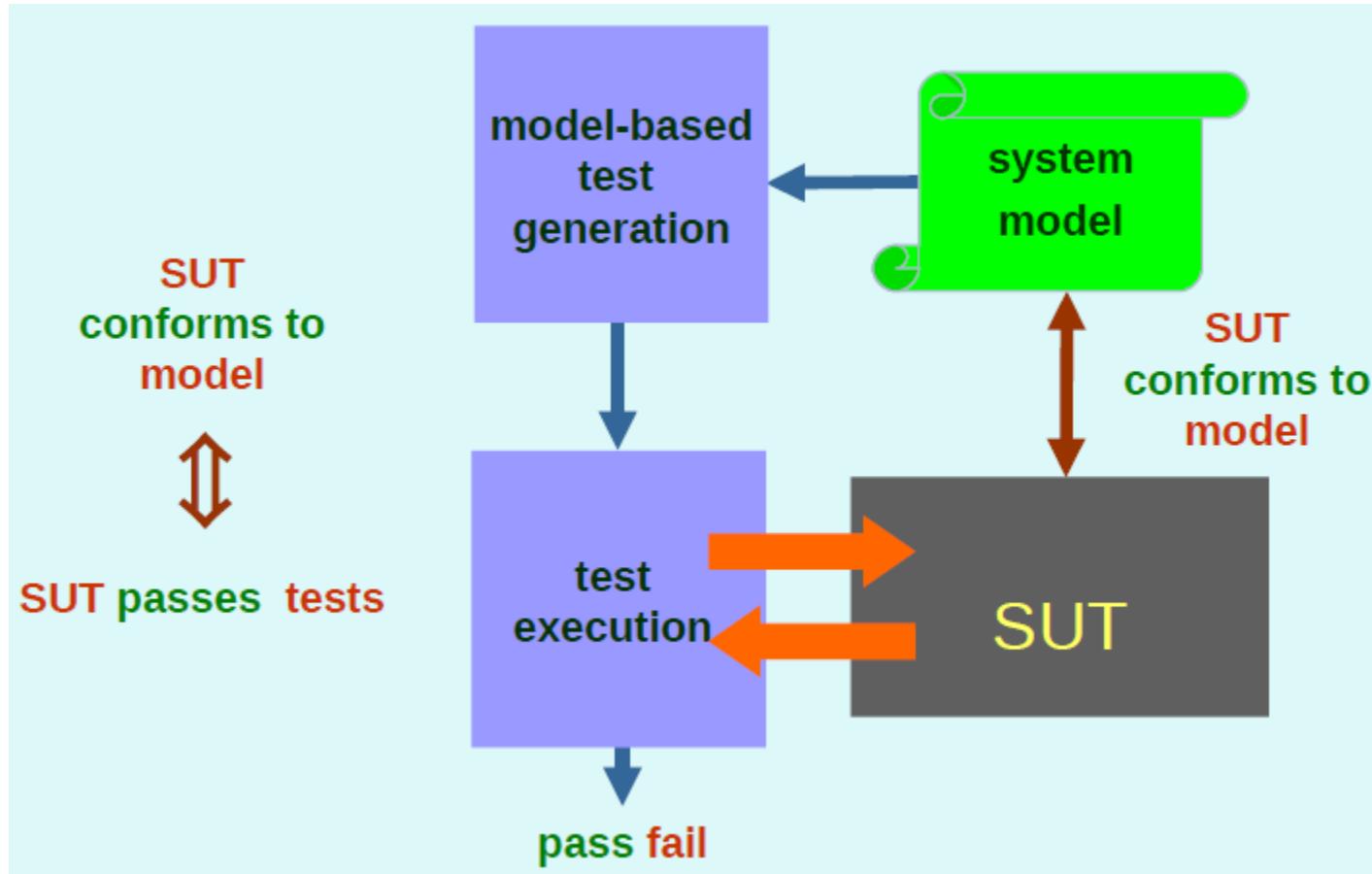
Developments in Testing 3



Developments in Testing 4



Model-Based Testing



Model-Based Testing

- Pruebas basadas en modelos -> Prueba de SUT con respecto a un modelo / especificación (formal)-> Modelo de estado/transición, condiciones previas/posteriores, CSP, Promela, UML, n.º de especificación,
- Promesa: mejores pruebas, más rápidas y más baratas
 - Generación algorítmica de pruebas y oráculos de prueba: herramientas
 - Base formal e inequívoca para las pruebas
 - Medición de la integridad de las pruebas
 - Mantenimiento de pruebas a través de la modificación del modelo
 - Potencial para combinar la práctica y la teoría

Utilidad de modelos formales

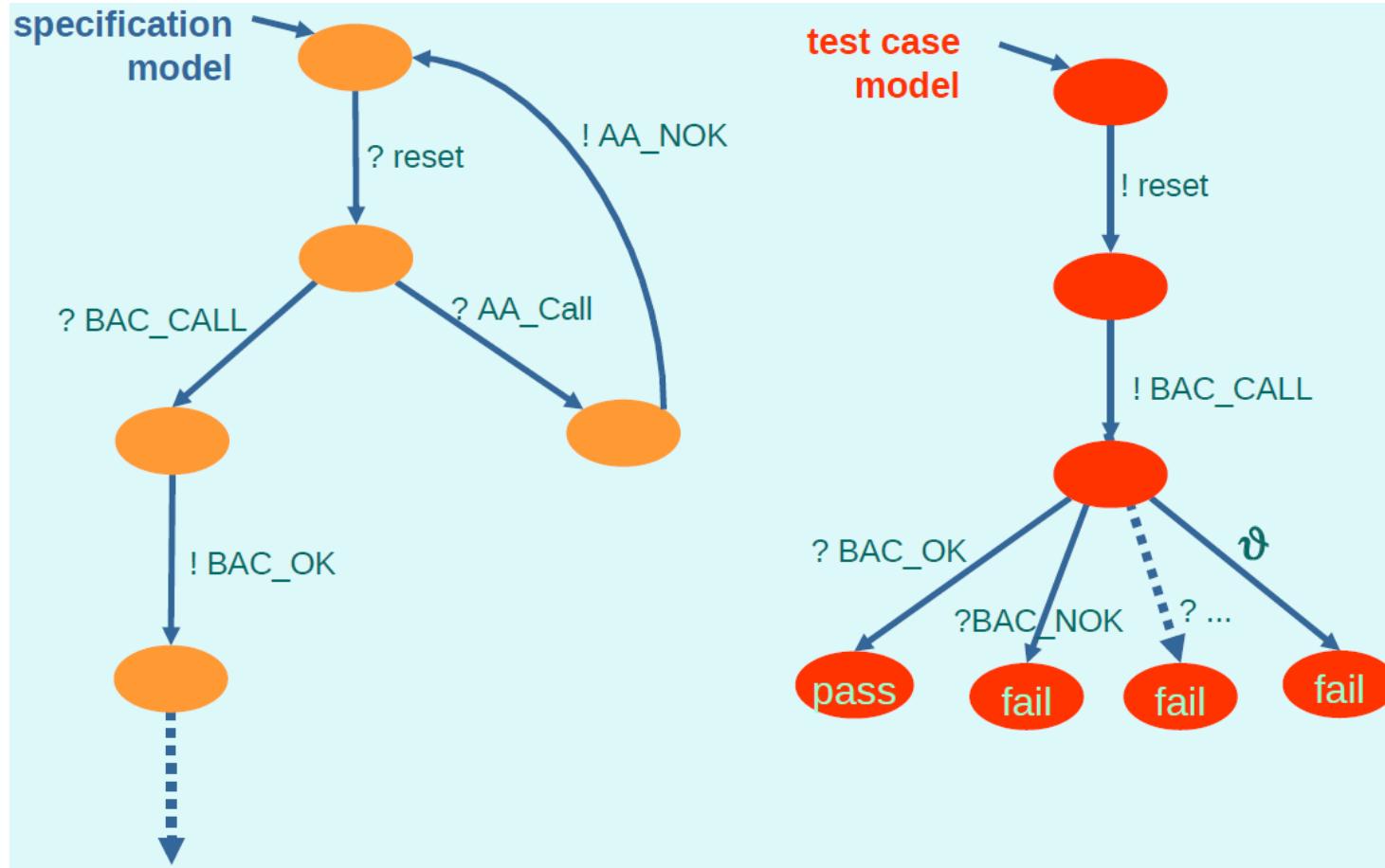
- Elaborar un modelo revela errores
- Simulación, paso a paso a través del modelo
- La comprobación del modelo pasa por todos los estados del modelo.
- Demostración de teoremas sobre el modelo
- Generación de código a partir del modelo.

Enfoques para Model-Based Testing

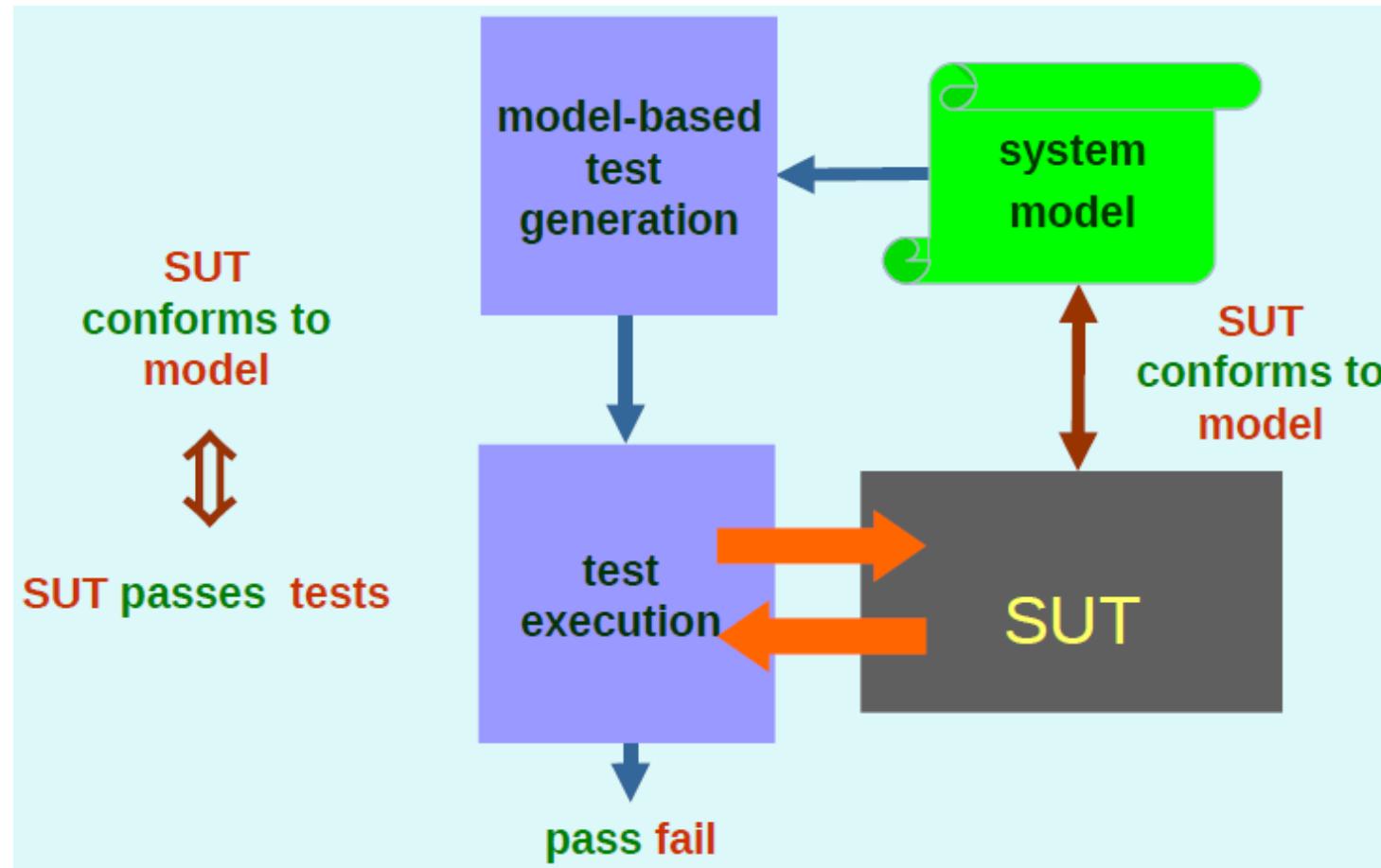
Varios paradigmas de modelado:

- Máquina de estados finitos
- Pre/Post Condiciones
- Sistemas de transición (LTS)
- Programas como Funciones
- Pruebas de tipos de datos abstractos
- Autómatas cronometrados
-

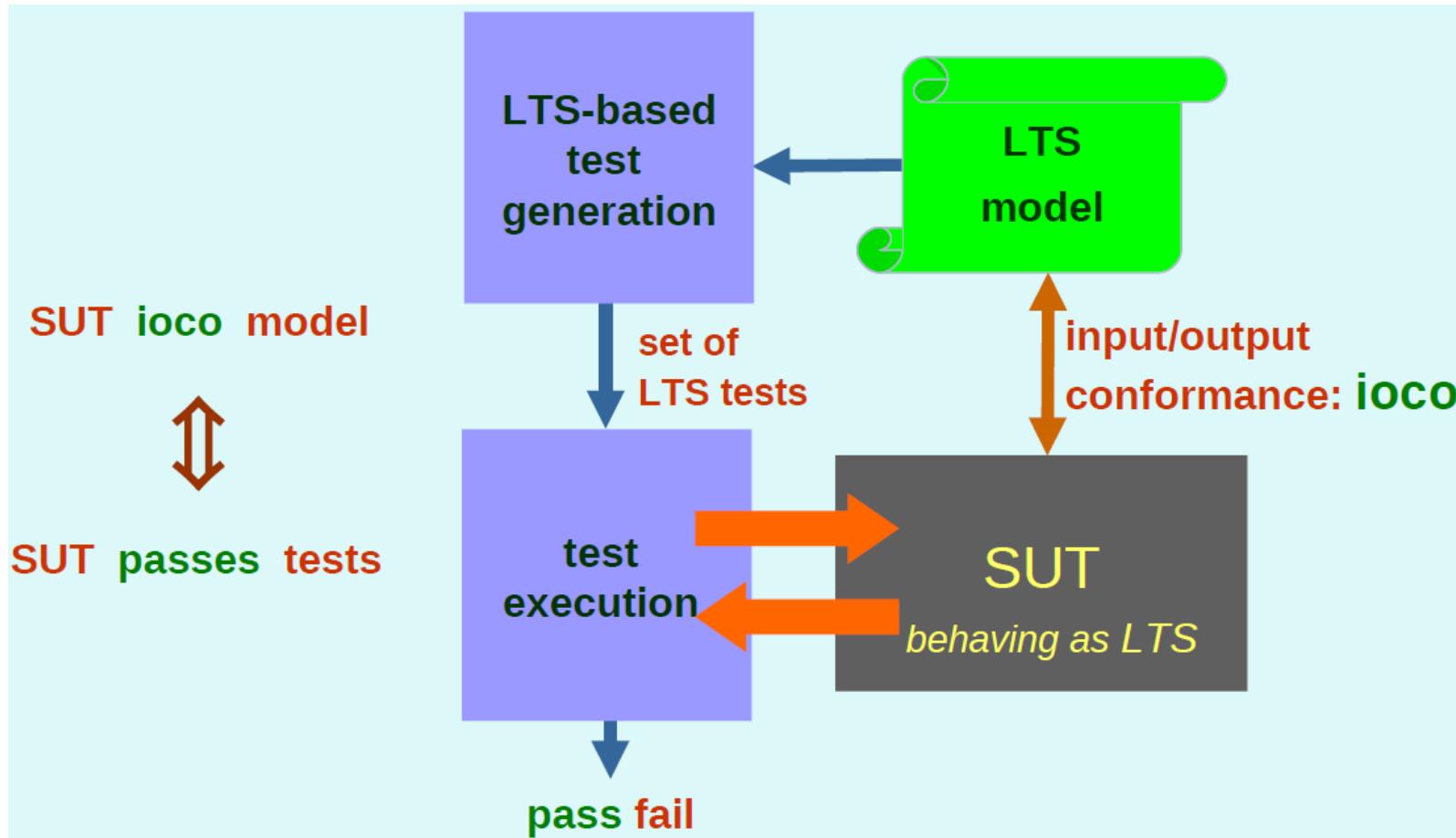
Model vs Model of Test Cases



Model-Based Testing



Model-Based Testing



Model-Based Testing Tools

Some Tools for Model-Based testing:

- | | | |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none">• AETG• Agatha• Agedis• Autolink• Conformiq• Cooper• Cover• STG• TestGen (Stirling) | <ul style="list-style-type: none">• QuickCheck• Reactis• RT-Tester• SaMsTaG• Smartesting• Spec Explorer• Statemate• TestGen (INT)• TorXakis | <ul style="list-style-type: none">• Test Composer• TGV• TorX• Uppaal-Tron• Tveda• Gotcha• Phact/The Kit |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Actividad



UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

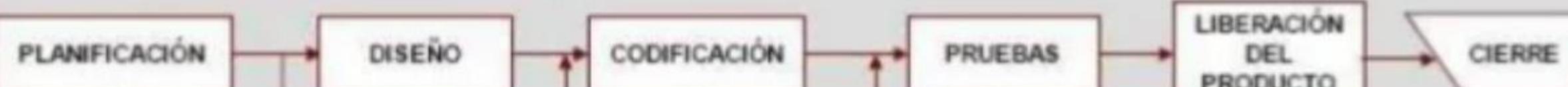


CURSO:
Métodos Formales de Pruebas - Testing de Software

Tema:
Diseño de casos de prueba
Semana 6

Docente: Mg. Wilder Inga

CICLO DE VIDA GENÉRICO DEL SOFTWARE



ESTRATEGIA
DE PRUEBAS

DISEÑO DE
PRUEBAS

Pruebas
unitarias

Pruebas
integración

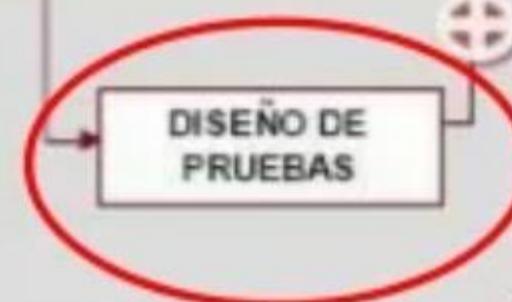
Pruebas
sistema

GESTIÓN DE
PUESTA EN
MARCHA

UAT

PRUEBAS DE RENDIMIENTO

PRUEBAS DE SEGURIDAD



Diseño de casos de prueba

- Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollado para conseguir un objetivo particular o condición de prueba. Ejemplo: verificar el cumplimiento de un requisito específico
- En la ingeniería de software, mediante el caso de prueba o test case el analista determinará, si una aplicación o una característica de esta es parcial o completamente satisfactoria.

Diseño de casos de prueba

- Para desarrollar software de calidad y libre de errores, el plan de pruebas y los casos de prueba son muy importantes.
- Un caso de prueba bien diseñado tiene gran posibilidad de llegar a resultados más fiables y eficientes, mejorar el rendimiento del sistema, y reducir los costos en tres categorías:
 - ✓ 1) productividad — menos tiempo para escribir y mantener los casos—;
 - ✓ 2) capacidad de prueba —menos tiempo para ejecutarlos—; y
 - ✓ 3) programar la fiabilidad —estimaciones más fiables y efectivas— (Perry, 1995).

Diseño de casos de prueba

- El proceso de escribir casos de prueba y establecer su estándar es un logro especial muy dinámico, y es necesario que se enseñe, aplique, controle, mida y mejore continuamente.
- Para llevar a cabo un caso de prueba, es necesario definir los siguientes pasos:
 - ✓ Definir escenarios
 - ✓ Identificar condiciones de entrada
 - ✓ Definir valores de entrada
 - ✓ Generar casos de prueba

Diseño de casos de prueba

- Para llevar a cabo un caso de prueba, es necesario definir los siguientes pasos:
 - Definir escenarios
 - Identificar condiciones de entrada
 - Definir valores de entrada
 - Generar casos de prueba

Definir escenarios

- Consiste en identificar todos los escenarios (caminos) a probar de un caso de uso: flujo básico, sub flujos y flujos alternativos.

Identificar condiciones de entrada

Las condiciones de entrada son parte del dominio de valores de entrada. Se pueden identificar condiciones de entrada con estados válidos (V) y no válidas (NV); asimismo se consideran condiciones de entrada con el estado que no se aplica (N/A) para un determinado escenario.

Existen los siguientes tipos de condiciones de entrada:

- Miembro de un conjunto
- Lógico
- Valor
- Rango

Identificar condiciones de entrada

Ejemplo: Considérese una aplicación bancaria, donde el usuario puede conectarse al banco por Internet y realizar una serie de operaciones bancarias. Una vez accedido al banco con las siguientes medidas de seguridad (clave de acceso), la información de entrada del procedimiento que gestiona las operaciones concretas a realizar por el usuario requiere las siguientes entradas:

- **Código de banco:** En blanco o número de tres dígitos el primer debe ser mayor que 1.
- **Código de sucursal:** Número de cuatro dígitos, el primero de ellos mayor a 0.
- **Número de cuenta:** Número de cinco dígitos.
- **Clave personal:** Valor alfanumérico de cinco posiciones.
- **Orden:** Este valor se seleccionará de una lista desplegable, según la orden que se deseé realizar. Puede estar en “Seleccione Orden” o una de las dos cadenas siguientes: “Talonario” o “Movimientos”
- En el caso “Talonario” el usuario recibirá un talonario de cheques, mientras que en “Movimientos” recibirá los movimientos del mes en curso. Si no se especifica este dato, el usuario recibirá los dos documentos.

Identificar condiciones de entrada

A continuación, se muestra una tabla con estados de las condiciones de entrada por cada resultado esperado.

ID CP	Escenario	CONDICIONES DE ENTRADA					Resultado esperado
		Código de banco	Código de sucursal	Número de cuenta	Clave personal	Orden	
CP1	Escenario 1	V	V	V	V	V	Mensaje "Envio de talonarios"
CP2	Escenario 1	V	V	V	V	V	Mensaje "Envio de movimientos "
CP3	Escenario 1	V	V	V	V	V	Mensaje "Envio de talonarios y movimientos"
CP4	Escenario 2	NV	V	V	V	V	Mensaje "Código de banco incorrecto"
CP5	Escenario 3	V	NV	V	V	V	Mensaje "Código de sucursal incorrecto"
CP6	Escenario 4	V	V	NV	V	V	Mensaje "Número de cuenta incorrecto"
CP7	Escenario 5	V	V	V	NV	V	Mensaje "Clave incorrecta"

Definir valores de entrada

Pueden usarse varias técnicas para identificar los valores de los datos de entrada, la técnica de particiones o clases de equivalencias es una de ellas.

Las clases de equivalencia se identifican examinando cada condición de entrada (normalmente una frase en la especificación) y dividiéndola en dos o más grupos.

Se definen dos tipos de clases de equivalencia:

- Clases Válidas: Entradas válidas al programa.
- Clases no Válidas: Valores de entrada erróneos.

Definir valores de entrada - ejemplo

- A continuación, se muestra las clases de equivalencia para el caso de gestión bancaria anterior.

Sec.	Condición de Entrada	Tipo	Clases Válidas		Clases No Válidas	
			Entrada	Código	Entrada	Código
1	Código de banco	Lógico (puede estar o no)	En blanco	CEV<01>	Un valor no numérico	CENV<01>
		Si está, es Rango	200 <= Código de banco <= 999	CEV<02>	Código de banco < 200 Código de banco > 999	CENV<02> CENV<03>
2	Código de sucursal	Rango	1000 <= Código de sucursal <= 9999	CEV<03>	Código de sucursal < 1000 Código de sucursal > 9999	CENV<04> CENV<05>
3	Número de cuenta	Valor	Cualquier número de 5 dígitos	CEV<04>	Número de más de cinco dígitos Número de menos de cinco dígitos	CENV<06> CENV<07>
4	Clave personal	Valor	Cualquier cadena de caracteres alfanuméricos de 5 posiciones	CEV<05>	Cadena de más de cinco posiciones Cadena de menos de cinco posiciones	CENV<08> CENV<09>
5	Orden	Miembro de un conjunto, con comportamiento distinto	Orden = "Seleccione Orden"	CEV<06>		
			Orden = "Talonario"	CEV<07>		
			Orden = "Movimientos"	CEV<08>		

Generar casos de prueba

En esta última etapa, se generan los casos de pruebas. Para ello, se considera como referencia la tabla de condiciones de entrada, indicando en cada caso de prueba las clases de equivalencia creadas.

Por ejemplo, para el caso bancario se tendría lo siguiente:

ID CP	Clases de equivalencia	CONDICIONES DE ENTRADA					Resultado esperado
		Código de banco	Código de sucursal	Número de cuenta	Clave personal	Orden	
CP1	CEV<02>, CEV<03>, CEV<04>, CEV<05>, CEV<07>	200	1000	10000	Aaaaa	"Talonario"	Mensaje "Envio de talonarios"
CP2	CEV<01>, CEV<03>, CEV<04>, CEV<05>, CEV<08>	820	9999	99999	Zzzzz	"Movimientos"	Mensaje "Envio de movimientos "
CP3	CEV<02>, CEV<03>, CEV<04>, CEV<05>, CEV<06>	999	1001	12345	A1b2c	"Seleccione Orden"	Mensaje "Envio de talonarios y movimientos"
CP4	CENV<01>, CEV<03>, CEV<04>, CEV<05>, CEV<07>	30A	1989	12345	1a2b3	"Seleccione Orden"	Mensaje "Código de banco incorrecto"
CP5	CENV<04>, CEV<03>, CEV<04>, CEV<05>, CEV<07>	210	999	12345	1a2b3	"Seleccione Orden"	Mensaje "Código de sucursal incorrecto"
CP6	CENV<07>, CEV<03>, CEV<04>, CEV<05>, CEV<07>	210	1989	123	1a2b3	"Seleccione Orden"	Mensaje "Número de cuenta incorrecto"
CP7	CENV<09>, CEV<03>, CEV<04>, CEV<05>, CEV<07>	210	1989	12345	"	"Seleccione Orden"	Mensaje "Clave incorrecta"

Componentes de un caso de prueba

Un caso de prueba es un conjunto de acciones con resultados y salidas previstas basadas en los requisitos de especificación del sistema; sus componentes son:

1. Propósito: de la prueba o descripción del requisito que se está probando
2. Método: o forma como se probará
3. Versión: o configuración de la prueba, versión de la aplicación en prueba, el hardware, el software, el sistema operativo, los archivos de datos, entre otros
4. Resultados: acciones y resultados esperados o entradas y salidas
5. Documentación: de la prueba y sus anexos.

Componentes de un caso de prueba

En cada nivel de la prueba, estos componentes deben probarse utilizando casos de prueba para pruebas de unidad, pruebas de integración, pruebas del sistema, pruebas Alpha y Beta,..., además, son útiles para las pruebas de rendimiento, pruebas funcionales y pruebas estructurales.

Factores de calidad de los casos de prueba

Un caso de prueba debe cumplir con los siguientes factores de calidad:

1. Correcto. Ser apropiado para los probadores y el entorno. Si teóricamente es razonable, pero exige algo que ninguno de los probadores tiene, se caerá por su propio peso.
2. Exacto. Demostrar que su descripción se puede probar.
3. Económico. Tener sólo los pasos o los campos necesarios para su propósito.
4. Confiable y repetible. Ser un experimento controlado con el que se obtiene el mismo resultado cada vez que se ejecute, sin importa qué se pruebe.
5. Rastreable. Saber qué requisitos del caso de uso se prueban.
6. Medible. Este es un ejercicio muy útil para quienes escriben pruebas, para verificar constantemente dónde están, si pierden alguno de los elementos, o si no se cumple un estándar.

Formato de los casos de prueba

1. Paso a paso. Este formato se utiliza en:

- La mayoría de las reglas de procesamiento
- Casos de prueba únicos y diferentes Interfaces GUI
- Escenarios de movimiento en interfaces diferentes
- Entradas y salidas complicadas para representar en matrices.

Formato de los casos de prueba

2. Matrices. Sus usos más productivos son:

- Formularios con información muy variada, mismos campos, valores y archivos de entrada diferentes
- Mismas entradas, diferentes plataformas, navegadores y configuraciones
- Pantallas basadas en caracteres
- Entradas y salidas con mejor representación matricial.

Formato de los casos de prueba

- 3. Scripts automatizados. La decisión de utilizar software para automatizar las pruebas depende de la organización y del proyecto que se esté probando. Existen algunas cuestiones técnicas que deben concretarse y que varían de una herramienta a otra. El beneficio real de automatizar las pruebas se obtiene en la fase de mantenimiento del ciclo de vida del software; en ese momento, los scripts se pueden ejecutar repetidamente, incluso sin supervisión, y el ahorro en tiempo y dinero es sustancial (Moller and Paulish, 1993)

Formato de los casos de prueba

- Un caso de prueba paso a paso tiende a ser más verbal, y el de matrices más numérico. A menudo, la ruta más productiva es utilizarlos todos. Los dos primeros se utilizan para las pruebas unitarias, de integración y del sistema; y el automatizado, para pruebas de regresión (Voas, 1993).

Mitos y realidades de los casos de prueba

Mito	Los casos de prueba paso a paso toman mucho tiempo para escribirse. No lo podemos permitir.
Realidad	Puede o no que tomen más tiempo para escribirse, pero su detalle los hace resistentes y fáciles de mantener; además, son necesarios para probar adecuadamente algunas de las funciones.
Mito	Una matriz es siempre la mejor opción. Hagámosla trabajar.
Realidad	Un problema persistente es armar una matriz con la información adecuada de la configuración. Frecuentemente se omite dicha información, o peor aún, si las configuraciones o clases de entrada son diferentes no se pueden forzar dentro de una matriz como grupo similar, ya que no se han probado todos.
Mitos	La alta tecnología es la mejor. Si es posible automatizar los casos de prueba, se debe hacer.
Realidad	La decisión de utilizar pruebas automatizadas debe basarse en muchos factores.
Mito	No tenemos tiempo para escribir los casos de prueba manuales. Vamos a automatizarlos.
Realidad	Automatizar los casos de prueba toma más tiempo que los otros dos tipos.

Plantilla para casos de prueba

Proyecto No.:	Página No.:
Nombre del Proyecto:	
Caso No.:	Ejecución No.:
Nombre del Caso:	Nombre: Estado de la prueba:
Marca/Subsistema/Módulo/Nivel/Función/Código de la Unidad bajo prueba:	Requisito No.:
Escrito por: Fecha:	Nombre: Ejecutado por: Fecha:
Descripción del caso de prueba (propósito y método):	
Configuración de la prueba para (H/W, S/W, N/W, datos, pre-requisitos de prueba, seguridad y tiempo):	

Paso	Acción	Resultados esperados	Pasado/Fallido
1			
2			
...			

Plantilla para matrices

Checklist para la calidad de un caso de prueba

Atributo	Lista de chequeo	S/N
Calidad	Correcto. Es apropiado para los probadores y el entorno	
	Exacto. Su descripción se puede probar	
	Económico. Tiene sólo los pasos o los campos necesarios para su propósito	
	Confiable y repetible. Se obtiene el mismo resultado sin importa qué se pruebe	
	Rastreable. Se sabe qué requisito se prueba	
	Medible. Retorna al estado de la prueba sin valores en su estado	
Estructura y de capacidad de prueba	Tiene nombre y número	
	Tiene un propósito declarado que incluye qué requisito se está probando	
	Tiene una descripción del método de prueba	
	Especifica la información de configuración –entorno, datos, pre-requisitos de prueba, seguridad–	
	Tiene acciones y resultados esperados	
	Guarda el estado de las pruebas, como informes o capturas de pantalla	
	Mantiene el entorno de pruebas limpio	
	No supera los 15 pasos	
	La matriz no demora más de 20 minutos para probarse	
	El script automatizado tiene propósitos, entradas y resultados esperados	
	La configuración ofrece alternativas a los pre-requisitos de la prueba cuando es posible	
	El escenario de aplicación se relaciona con otras pruebas	
Administración de la configuración	Emplea convenciones de nomenclatura y numeración	
	Guarda en formatos especificados los tipos de archivo	
	Su versión coincide con el software bajo prueba	
	Incluye objetos de prueba necesarios para el caso, tales como bases de datos	
	Almacena con acceso controlado	
	Realiza copias de seguridad en red	
	Archiva por fuera del sitio	

Actividad - Individual

Diseño de pruebas Partiendo de los grupos establecidos. Realizar:

- Cada integrante escoge dos requisitos del proyectos del curso (definidos o por definir).
- Definir escenarios para los dos requisitos
- Identificar condiciones de entrada
- Definir datos de entrada usando Clases de Equivalencia (revise el video de apoyo en Classroom).
- Realizar casos de prueba. Mínimo 5 por cada requisito.