UNIVERSDAD 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.

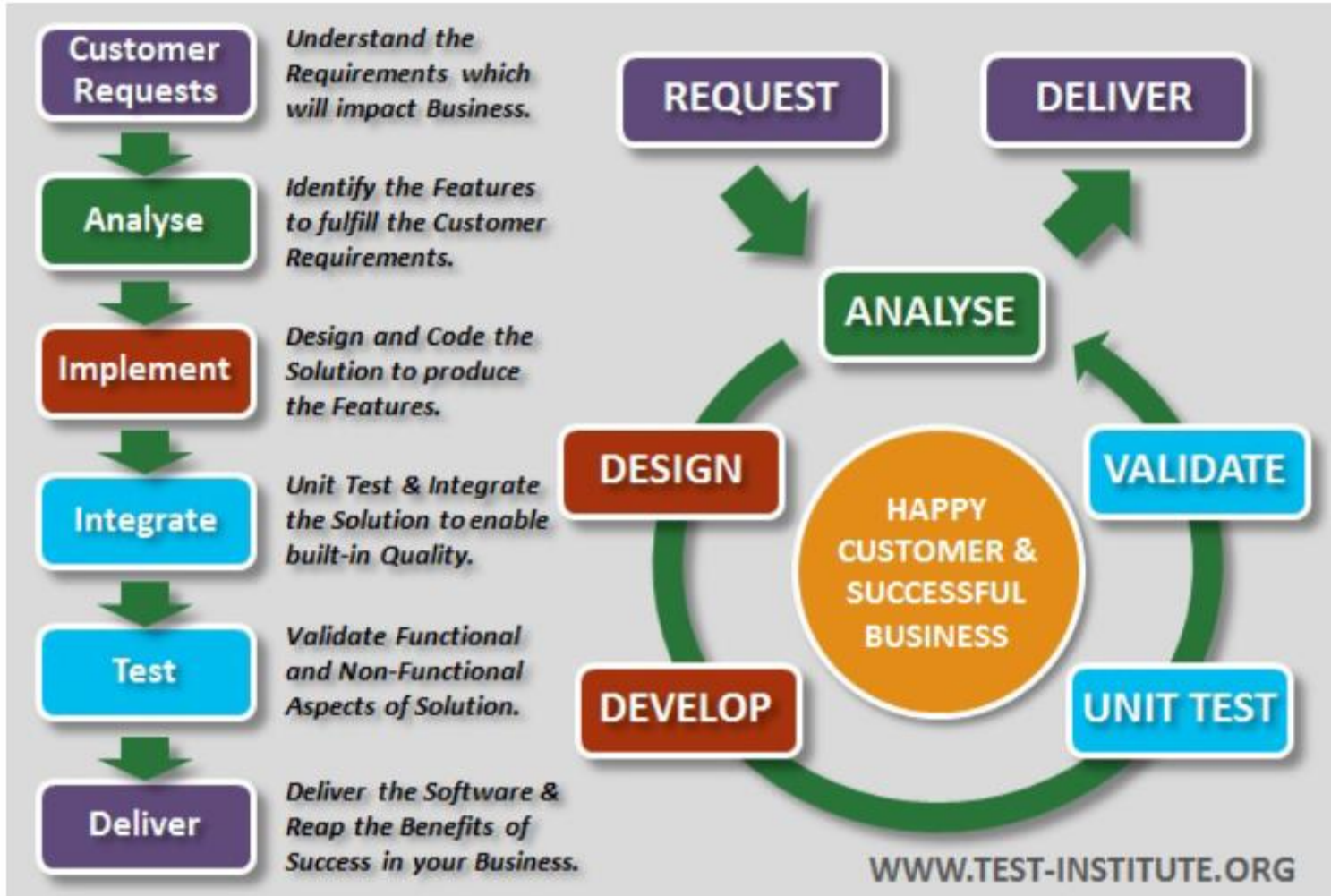
Software Testability

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

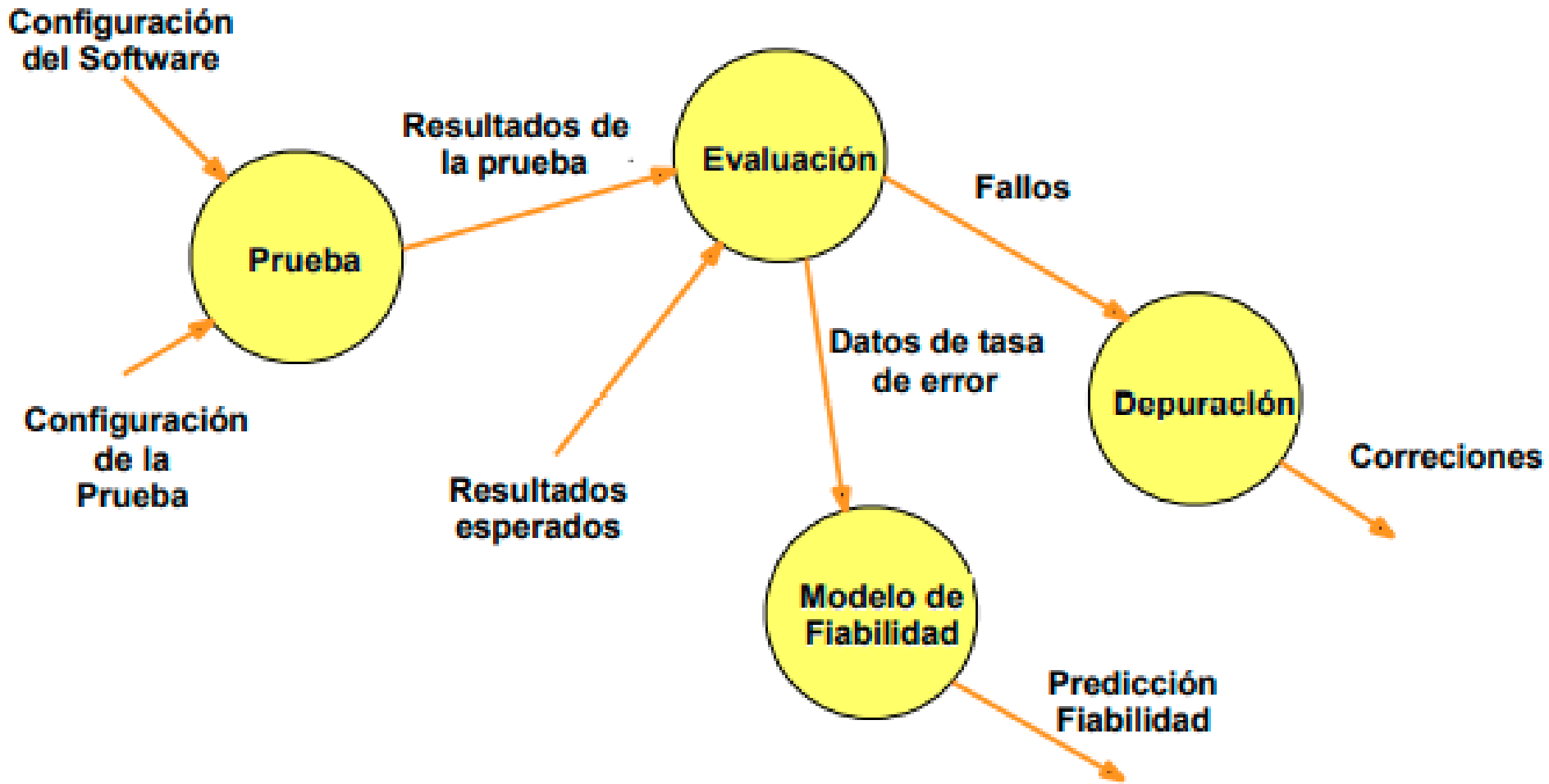


Figura 1. Contexto de la Prueba de Software

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 ahorran 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 ahorran 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 :

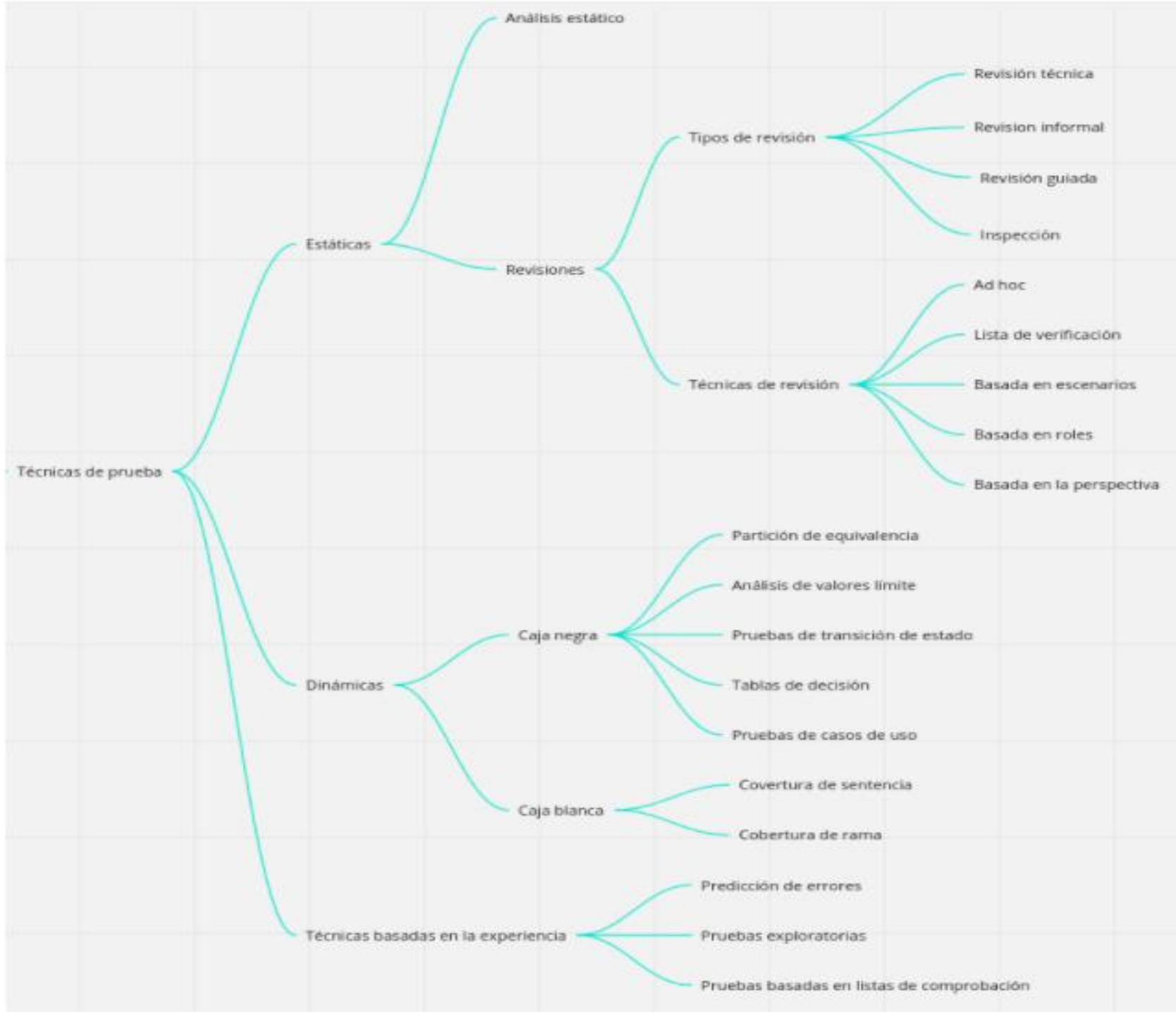
- Se requiere testing efectivo,
  - Unitarias, Integración y sistema.
- Se debe dejar evidencias del testing.

Inspección y Testing :

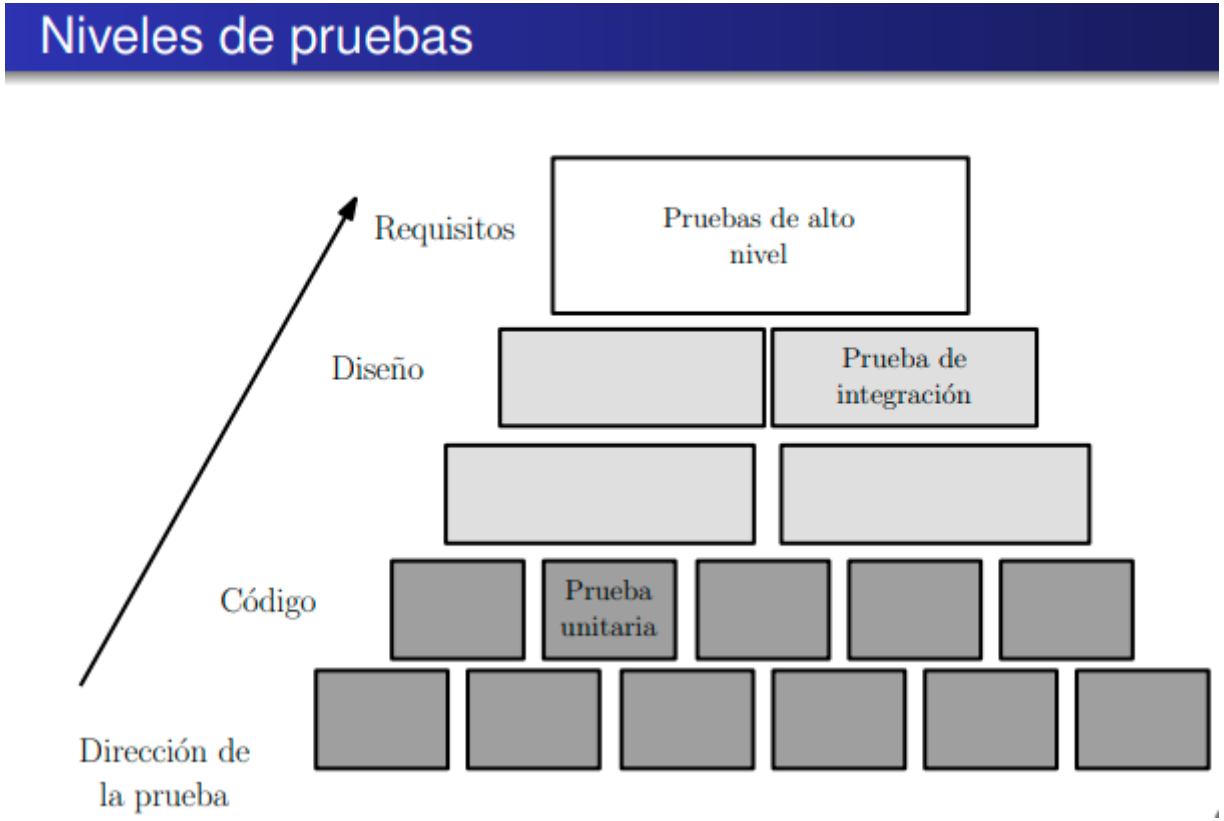
- Si se utilizan herramientas de Inspección, medición y testing,
  - debe configurarse y calibrarse adecuadamente.

Técnicas de pruebas

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


Técnicas de prueba
Estáticas
Análisis estático
Revisiones
Tipos de revisión
Revisión técnica
Revision informal
Revisión guiada
Inspección
Técnicas de revisión
Ad hoc
Lista de verificación
Basada en escenarios
Basada en roles
Basada en la perspectiva
Dinámicas
Caja negra
Partición de equivalencia
Análisis de valores límite
Pruebas de transición de estado
Tablas de decisión
Pruebas de casos de uso
Caja blanca
Covertura de sentencia
Cobertura de rama
Técnicas basadas en la experiencia
Predicción de errores
Pruebas exploratorias
Pruebas basadas en listas de comprobación

Niveles de prueba

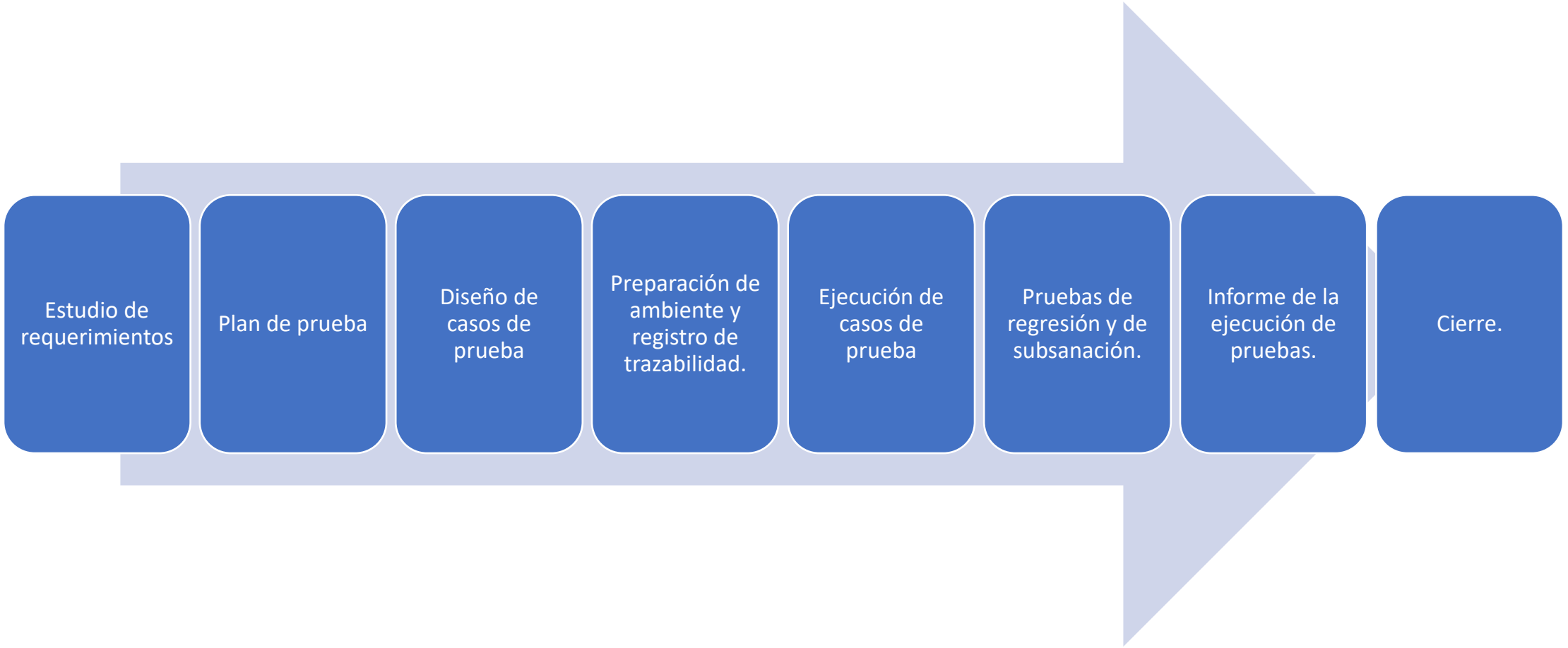


Espectro de los principales niveles y técnicas de pruebas

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

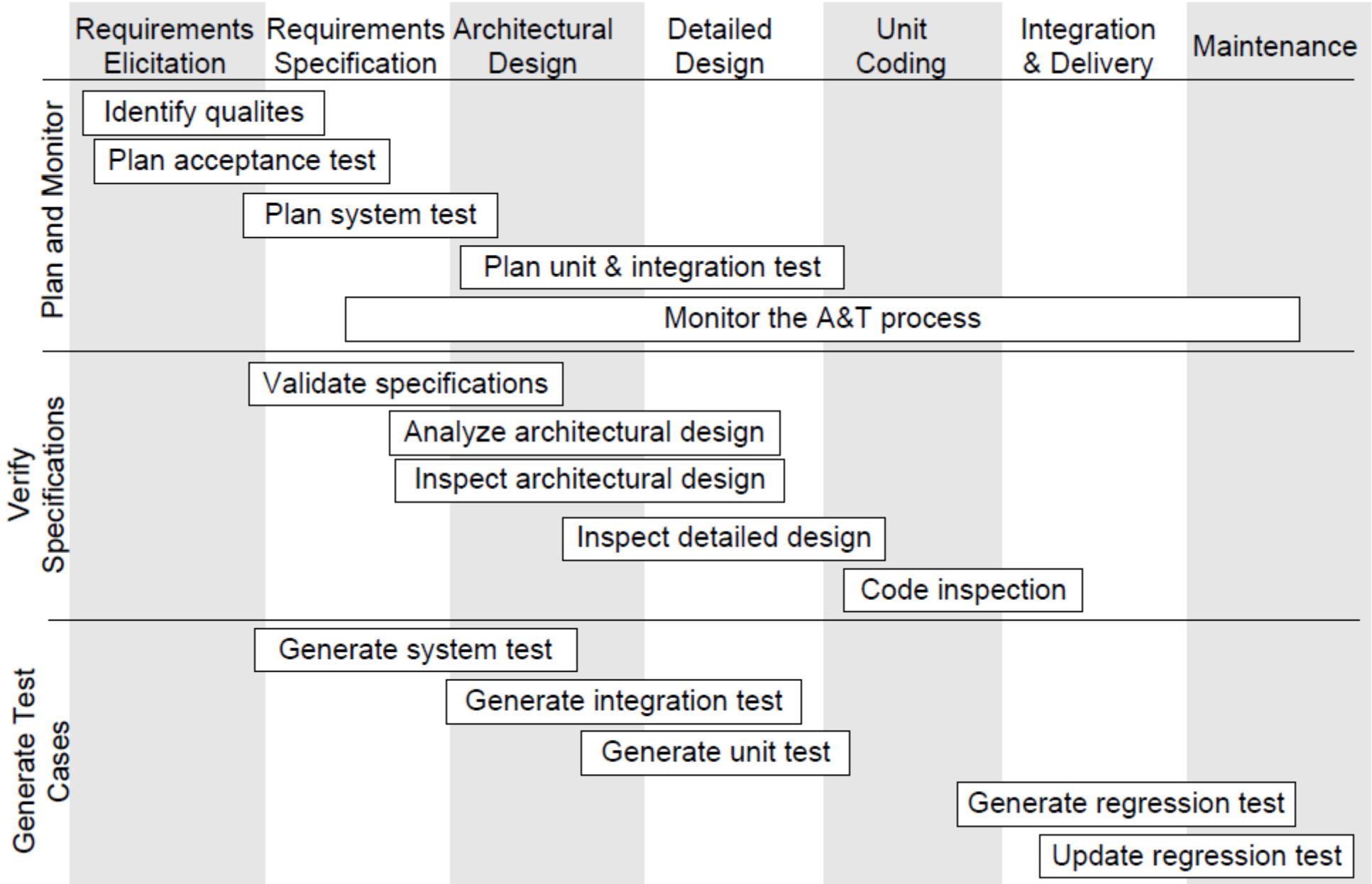
Principales actividades del análisis y testing en el desarrollo de software

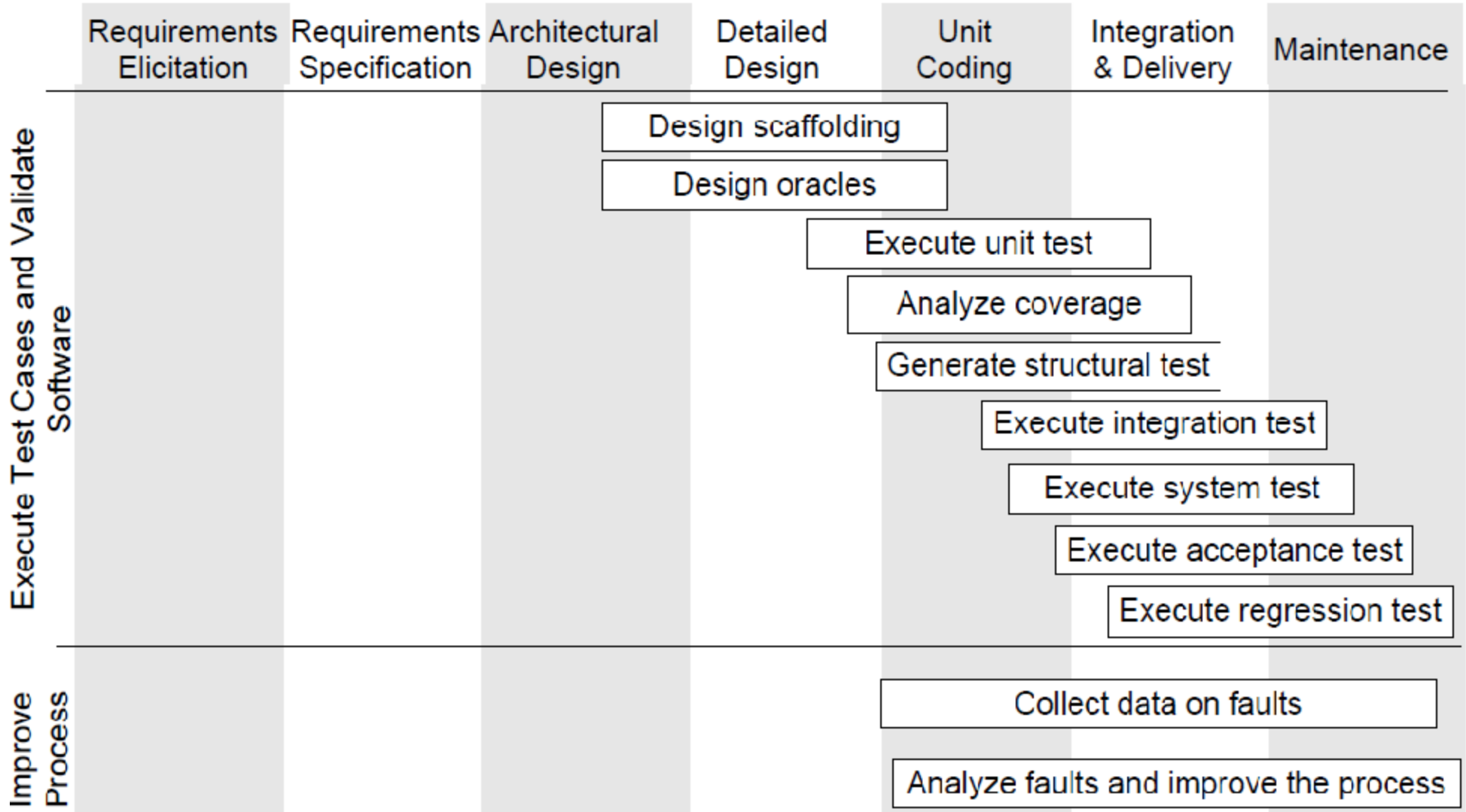
Software Testing Life Cycle



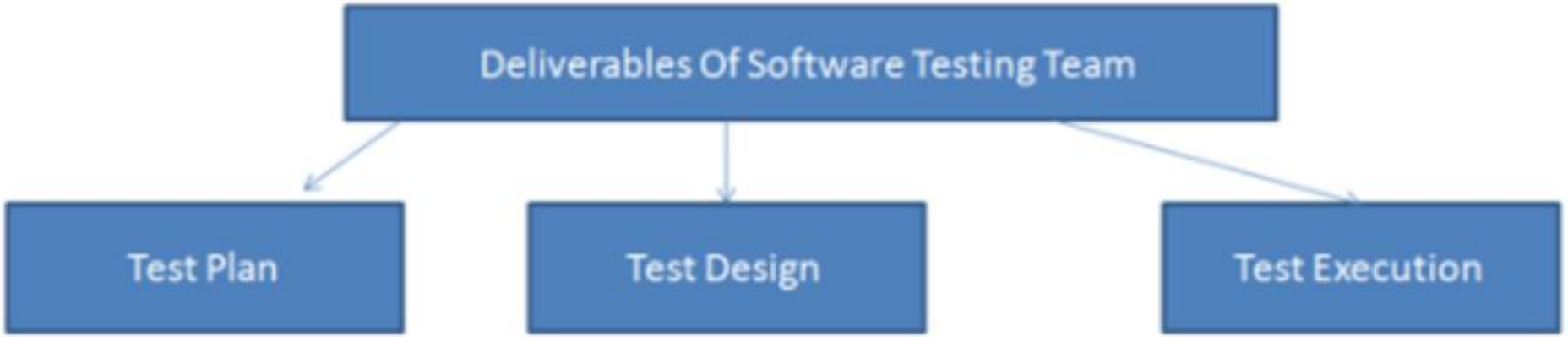
Software Testing Life Cycle

- 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.


Requirements Elicitation
Requirements Specification
Architectural Design
Detailed Design
Unit Coding
Integration & Delivery
Maintenance
Plan and Monitor
Identify qualites
Plan acceptance test
Plan system test
Plan unit & integration test
Monitor the A&T process
Verify Specifications
Validate specifications
Analyze architectural design
Inspect architectural design
Inspect detailed design
Code inspection
Generate Test Cases
Generate system test
Generate integration test
Generate unit test
Generate regression test
Update regression test


Requirements Elicitation
Requirements Specification
Architectural Design
Detailed Design
Unit Coding
Integration & Delivery
Maintenance
Execute Test Cases and Validate Software
Design scaffolding
Design oracles
Execute unit test
Analyze coverage
Generate structural test
Execute integration test
Execute system test
Execute acceptance test
Execute regression test
Improve Process
Collect data on faults
Analyze faults and improve the process

Entregables de testing



Analysis
Strategy
Plan
Tools
Required

Test Cases
Scripts
Scenarios
Environment

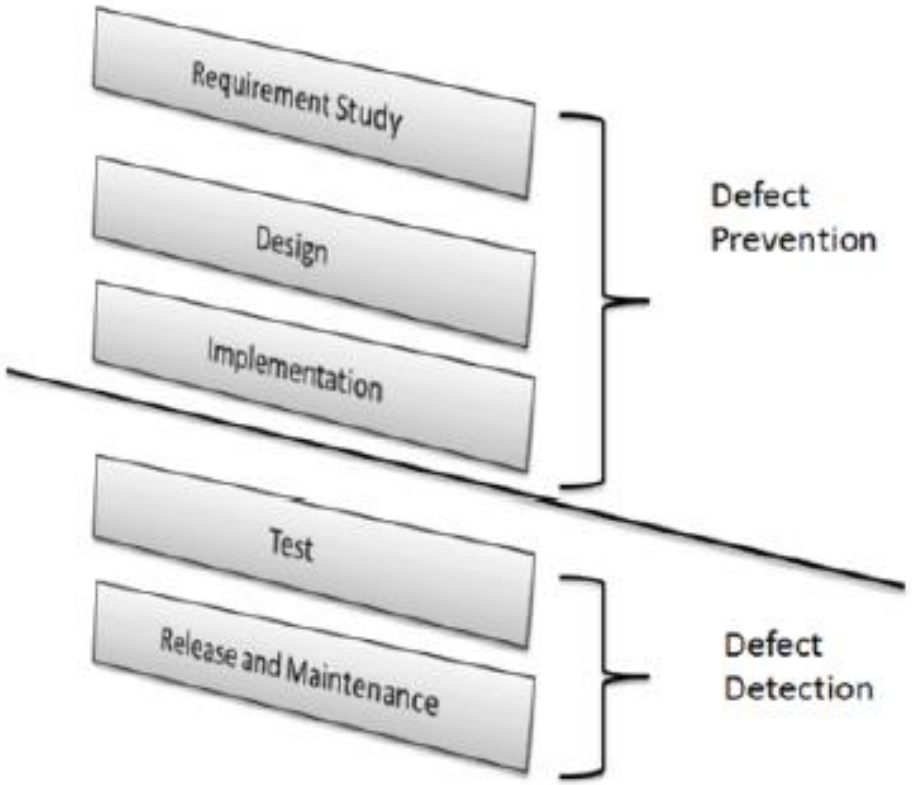
Test Report
Defect Tracking
Report
Defect Analysis

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

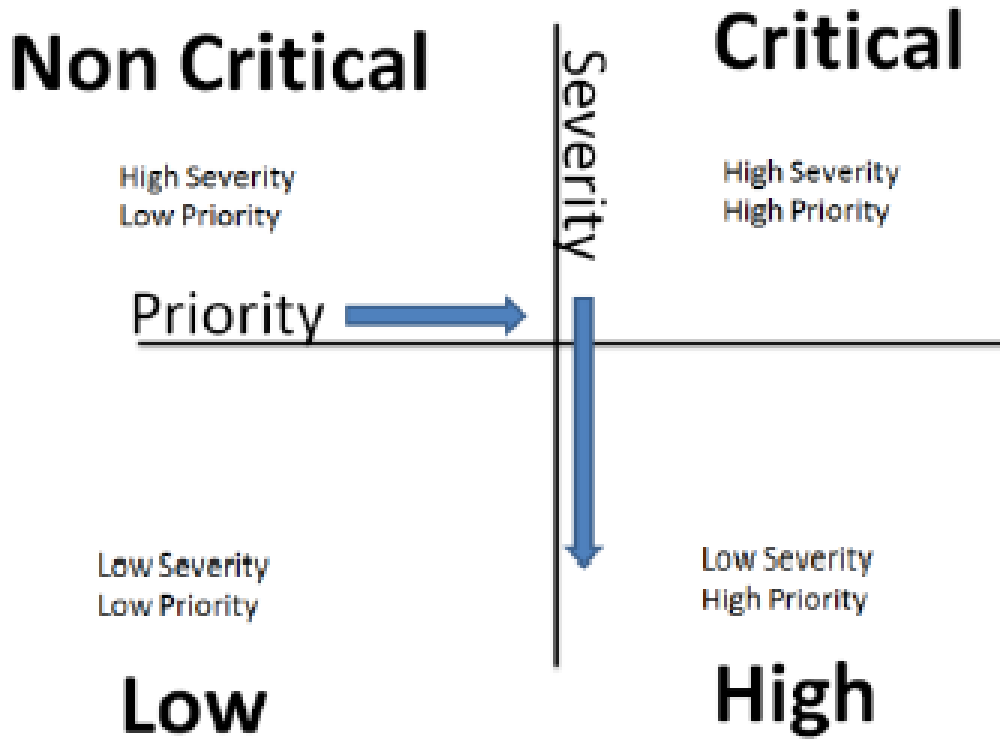
¿Qué es un bug?

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.



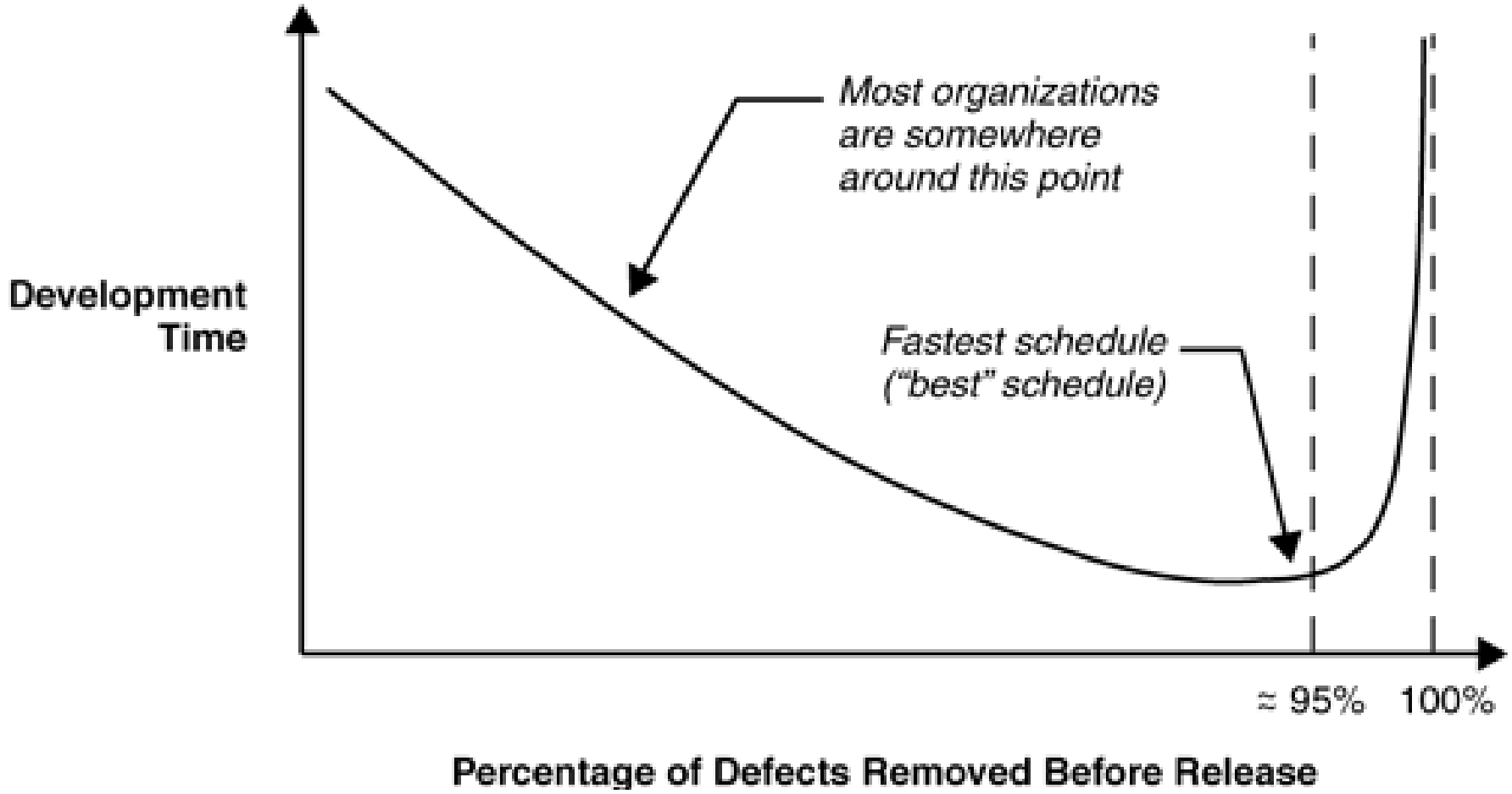
Prioridad, severidad y criticidad

- 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.

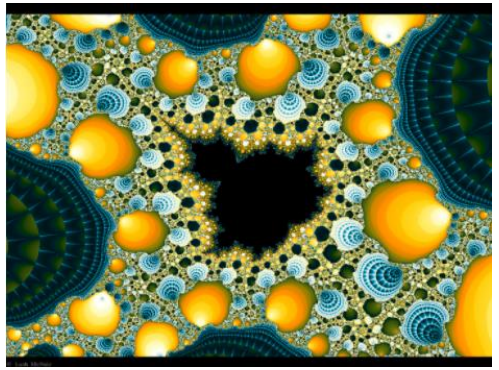


Prioridad, severidad y criticidad

- 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.


Most organizations
are somewhere
around this point
Fastest schedule
("best" schedule)
Development
Time
≈ 95%
100%
Percentage of Defects Removed Before Release

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 limites, 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 por que 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.

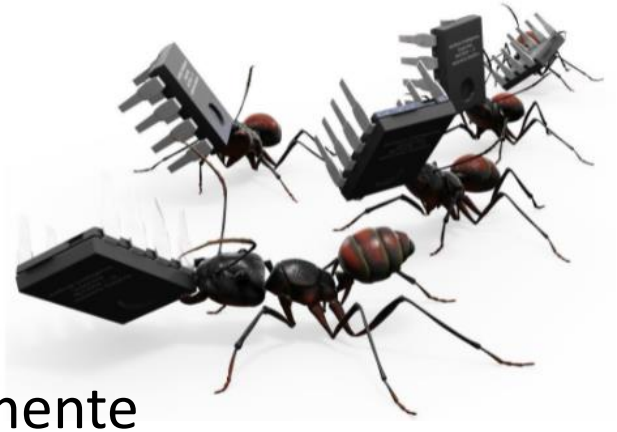
Tipos de defectos

- 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 tenia 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