

Unidad 4: Aseguramiento de calidad de Proceso y de Producto

Conceptos generales sobre calidad

Importancia de trabajar para y con Calidad. Ventajas y Desventajas.

Calidad

Formalmente se puede definir como el cumplimiento de los requerimientos funcionales y de performance explícitamente definidos, de los estándares de desarrollo explícitamente documentados y de las características implícitas esperadas del desarrollo de software profesional.

Todos los aspectos y características de un producto o servicio que se relacionan con la habilidad de alcanzar las necesidades manifiestas o implícitas. Está relacionado con MIS necesidades y MIS expectativas (se relaciona con las necesidades implícitas ya que no son expresadas).

Está directamente relacionada con la persona (es subjetiva a quién la analice), las circunstancias, el contexto, la edad en un momento de tiempo dado (puede para mí algo no tener calidad, pero para otra persona sí).

- **Concepto de Calidad en el Software:** La calidad del software se define como el grado en que un producto cumple con los requerimientos funcionales y de rendimiento establecidos, los estándares de desarrollo documentados y las expectativas implícitas del usuario.

- En otras palabras, un software es de calidad si:

- **Funciona correctamente** (cumple lo que promete).
- **Rinde bien** (es eficiente y estable).
- **Está bien construido** (sigue estándares profesionales).
- **Satisface las expectativas del usuario**, incluso las que no se mencionan explícitamente.

- **Ejemplo:** Un sistema de turnos médicos puede cumplir con las funciones pedidas (registrar pacientes, asignar turnos), pero si tarda mucho en cargar o tiene una interfaz confusa, los usuarios lo perciben como de baja calidad.

La calidad es subjetiva: depende de la persona, el contexto y el momento. Lo que para un usuario joven es intuitivo, puede ser confuso para uno mayor.

Gestión de calidad

La gestión de calidad del software para los sistemas de software tiene tres intereses fundamentales:

- A nivel de organización, la gestión de calidad se ocupa de establecer un marco de proceso y estándares de organización que conducirán a software de mejor calidad.
- A nivel del proceso, la gestión de calidad implica la aplicación de procesos específicos de calidad y la verificación de que continúen dichos procesos planeados.
- A nivel del proyecto, la gestión de calidad se ocupa también de establecer un plan de calidad para un proyecto.

- **Gestión de la Calidad del Software:** La gestión de calidad abarca tres niveles complementarios:

1. **A nivel organizacional:**

Se establecen marcos de trabajo y estándares globales que guían a todos los proyectos.
□ Ejemplo: usar ISO 9001 o CMMI como base de procesos internos.

2. **A nivel de proceso:**

Se verifica que los procesos de desarrollo sigan los pasos definidos y se mantengan en el tiempo.
□ Ejemplo: realizar revisiones de código o auditorías internas para asegurar que se cumplen las buenas prácticas.

3. **A nivel de proyecto:**

Se crea un plan de calidad específico que define:

- Qué atributos se buscan (rendimiento, usabilidad, seguridad, etc.).
- Cómo se medirán esos atributos.

□ Ejemplo: definir que el sistema debe responder en menos de 2 segundos y que esa métrica se testeará en cada versión.

Aseguramiento de calidad

Es la definición de procesos y estándares que deben conducir a la obtención de productos de alta calidad y, en el proceso de fabricación, a la introducción de procesos de calidad.

El espíritu del aseguramiento de la calidad del SW es actuar como medida preventiva a diferencia del testing que llega tarde.

El equipo QA debe ser independiente del equipo de desarrollo para que pueda tener una perspectiva objetiva del software.

La planeación de calidad es el proceso de desarrollar un plan de calidad para un proyecto. El plan de calidad debe establecer las cualidades deseadas de software y describir cómo se valorarán. Por lo tanto, define lo que realmente significa software de "alta calidad" para un sistema particular.

Humphrey (1989), en su libro referente a la gestión del software, sugiere un bosquejo de estructura para un plan de calidad. Éste incluye:

- Introducción del proyecto;

- **Planes del producto:** fechas de entrega críticas y las responsabilidades para el producto;
- **Descripciones de procesos;**
- **Metas de calidad:** Las metas y los planes de calidad para el producto, incluyendo una identificación y justificación de los atributos esenciales de calidad del producto;
- **Riesgos y gestión de riesgos.**

Aunque los estándares y procesos son importantes, los **administradores de calidad** deben enfocarse también a **desarrollar una “cultura de calidad”** en la que todo responsable del desarrollo del software se comprometa a lograr un alto nivel de calidad del producto. Deben exhortar a los **equipos a asumir la responsabilidad** de la calidad de su trabajo y desarrollar nuevos enfoques para el mejoramiento de la calidad.

Aseguramiento de la Calidad (QA – Quality Assurance) El Aseguramiento de Calidad (QA) busca **prevenir errores antes de que ocurran**, definiendo procesos y estándares de trabajo.

A diferencia del **testing**, que detecta errores después del desarrollo, el QA actúa antes y durante el proceso.

Principales ideas:

- QA establece los procesos y métodos para obtener un producto confiable.
- El equipo QA debe ser **independiente del desarrollo**, para mantener objetividad.
- El **plan de calidad** define las metas, atributos de calidad y cómo se van a evaluar.

Según Humphrey (1989), un **plan de calidad** debe incluir:

1. Introducción del proyecto.
2. Planes del producto (fechas, responsables).
3. Descripción de procesos.
4. Metas y atributos de calidad.
5. Riesgos y su gestión.

Ejemplo: Antes de desarrollar un e-commerce, QA puede definir que se use un estándar de codificación (PEP8 en Python), revisiones por pares, pruebas unitarias automáticas y seguimiento de incidencias en Jira.

Cultura de Calidad Más allá de normas y procesos, se busca **crear una cultura de calidad**:

- Cada integrante debe asumir la **responsabilidad de su propio trabajo**.
- Se debe fomentar la **mejora continua** y la búsqueda de nuevas formas de optimizar la calidad.

Ejemplo: Un equipo que revisa sus propios errores en retrospectivas de sprint y ajusta su flujo de trabajo está construyendo una cultura de calidad.

Problemas en la calidad

- Atrasos en las entregas -> Relacionado a calidad del **proyecto**.
- Requerimientos no claros -> Relacionado a calidad del **producto**.
- Software que no hace lo que debería -> Relacionado a calidad del **producto**.
- Costos excedidos -> Relacionado a calidad del **proyecto**.
- Trabajo fuera de hora -> Relacionado a calidad del **proyecto**.
- Fenómeno 90-90 (90% hecho 90% restante) -> Relacionado a calidad del **proyecto**.
- No se aplica SCM. -> Relacionado a calidad del **producto**.



Un software de calidad no solo debería cumplir con los requerimientos, sino que también debería poder entregarse a tiempo, no exceder costos, etc.

Problema	Tipo de Calidad afectada	Ejemplo
Atrasos en entregas	Calidad del proyecto	El sistema se entrega fuera del cronograma.
Requerimientos no claros	Calidad del producto	El software no hace lo que el cliente esperaba.
Software incorrecto	Calidad del producto	Funciones que no cumplen su objetivo.
Costos excedidos	Calidad del proyecto	Se gasta más de lo presupuestado.
Trabajo fuera de hora	Calidad del proyecto	Mala planificación y sobrecarga del equipo.
“Fenómeno 90-90”	Calidad del proyecto	Cuando parece estar listo pero aún falta la mitad.
Falta de control de versiones (SCM)	Calidad del producto	Pérdida de trazabilidad o errores al combinar código.

Un **software de calidad** debe cumplir los requerimientos, pero también **respetar tiempos, costos y mantener estabilidad** durante su ciclo de vida.

Aseguramiento de calidad de Proceso y de Producto



Calidad en el desarrollo de Software

Los costos excedidos, retrasos en la entrega, falta de cumplimiento de los compromisos, requerimientos no claros hacen que la percepción de nuestro cliente sea mala. El **proyecto es el medio que permite alcanzar el producto**. Si el **medio no tiene calidad, difícilmente se pueda lograr un producto de calidad**.

Para que un **SW sea de calidad** debe satisfacer:

- Las **expectativas del cliente**;
- Las **expectativas del usuario**;
- Las **necesidades de la gerencia**;
- Las **necesidades del equipo de desarrollo y mantenimiento**;
- Las **expectativas de otros interesados**.

Calidad en el desarrollo de software

El **proceso** influye directamente en la calidad del producto.

Si el proceso es desorganizado, el producto resultante probablemente sea deficiente.

Un software de calidad debe satisfacer las necesidades de:

- **Clientes** (rentabilidad, cumplimiento del contrato).
- **Usuarios** (usabilidad, confiabilidad).
- **Gerencia** (cumplimiento de plazos y costos).
- **Equipo de desarrollo** (claridad técnica, mantenibilidad).
- **Otros interesados** (inversores, auditores, soporte, etc.).

Ejemplo: En una app bancaria, el cliente espera seguridad y cumplimiento legal, los usuarios quieren rapidez y simplicidad, y el equipo técnico busca un sistema mantenable. Un buen proceso de QA debe equilibrar todas esas perspectivas.

Principios de Calidad

- La calidad NO se inyecta, debe estar **embebida**: la calidad es algo que se concibe **desde el momento cero**, desde requerimientos en adelante. Lo que se hace con el **testing** es controlarla.
- Es una **responsabilidad de todos** los involucrados en el proyecto.
- **Las personas son la clave para lograrla**: se hace mucho hincapié en la **capacitación**, para brindar herramientas y conocimientos a los involucrados. "Si usted cree que la capacitación es cara entonces pruebe con la ignorancia". Hacer **software** es una actividad humano-intensiva
- Se necesita **sponsor a nivel gerencial**, pero se puede empezar por uno.
- Se debe **liderar con el ejemplo**.
- **No se puede controlar lo que no se mide, debemos usar métricas**.
- Simplicidad, **empezar por lo básico**.
- Debe **planificarse el aseguramiento de calidad**.
- El **aumento de las pruebas no aumenta la calidad**. La calidad se obtiene y se inyecta a lo largo del proyecto, no al final.
- Debe ser **razonable para mi negocio**.
- **Prevención mejor que corrección**: prevenir es menos costoso, ya que corregir demanda un costo muy alto asociado al retrabajo.

En la actualidad, el trabajo con calidad es **fundamental** en la producción del software ya que:

- Es un aspecto competitivo, que **permite sobrevivir en el mercado internacional**. Las empresas certifican

estándares de calidad (de proceso no de producto) para poder competir en el mismo.

- **Retiene a los clientes** e incrementa los beneficios. Siempre que hablamos de calidad, en el fondo nos referimos al cumplimiento de las expectativas del cliente en todos los sentidos.
- Determina un equilibrio del costo-efectividad. Trabajar con calidad reduce los costos y el retrabajo, ya que se asume que las fallas serán menores.

Las metodologías ágiles hablan constantemente de calidad de SW y se asume que los equipos ya están capacitados y orientados a procesos de calidad.

La disciplina de Gestión de la Administración de Configuración nos da el contexto de base para poder trabajar con calidad.

Principios de la Calidad del Software La calidad **no se agrega al final del desarrollo**, sino que **debe estar integrada desde el inicio**. Esto implica que cada etapa del ciclo de vida (análisis, diseño, codificación, pruebas y mantenimiento) debe contemplar la calidad como un objetivo.

Principios clave:

1. **La calidad no se inyecta, se diseña desde el comienzo.**
 - Se construye desde los **requerimientos**, no con pruebas al final.
 - □ Ejemplo: No sirve de nada hacer miles de tests si los requerimientos están mal definidos.
2. **Es responsabilidad de todos.**
 - Cada integrante (analista, desarrollador, tester, líder, cliente) influye en la calidad.
 - □ Ejemplo: Si un analista documenta mal un requisito, el error se propaga a todo el proyecto.
3. **Las personas son la clave.**
 - La calidad depende del conocimiento, la motivación y la capacitación del equipo.
 - □ Frase clave: "Si cree que la capacitación es cara, pruebe con la ignorancia."
 - □ Ejemplo: Un programador mal entrenado puede generar errores costosos aunque siga el proceso correcto.
4. **Se necesita apoyo gerencial (sponsor).**
 - La dirección debe respaldar las iniciativas de calidad con recursos y tiempo.
5. **El liderazgo por el ejemplo.**
 - Los líderes deben **modelar comportamientos de calidad**, no solo exigirlos.
6. **"No se puede controlar lo que no se mide."**
 - Las métricas son esenciales para evaluar y mejorar.
 - □ Ejemplo: medir defectos por módulo, porcentaje de cobertura de tests o cumplimiento de hitos.
7. **Simplicidad y foco.**
 - Comenzar con procesos simples y expandirlos gradualmente.
8. **Planificación del aseguramiento de calidad.**
 - El QA no debe improvisarse: necesita un plan definido y revisado.
9. **Más pruebas ≠ más calidad.**
 - Aumentar las pruebas sin mejorar el proceso no garantiza calidad.
 - □ Ejemplo: Testear un sistema mal diseñado solo confirma los errores más rápido.
10. **Debe ser razonable para el negocio.**
 - El nivel de calidad debe equilibrarse con el costo y el contexto del proyecto.
11. **Prevención antes que corrección.**
 - Corregir errores después es mucho más caro que prevenirlos.
 - □ Ejemplo: Detectar una mala especificación en la etapa de análisis cuesta poco; corregirla en producción puede costar miles.

Importancia actual de trabajar con calidad

Hoy en día, **trabajar con calidad es un factor competitivo clave** para las empresas de software.

Ventajas principales:

1. **Competitividad internacional:**
 - Las certificaciones (como ISO 9001, CMMI o SPICE) demuestran procesos confiables, permitiendo competir globalmente.
 2. **Fidelización y satisfacción del cliente:**
 - Un cliente satisfecho tiende a renovar contratos y recomendar la empresa.
 - □ Ejemplo: Una empresa que entrega software estable y fácil de mantener gana confianza y futuros proyectos.
 3. **Eficiencia y reducción de costos:**
 - Menos retrabajo = menor costo total.
 - □ Ejemplo: Implementar buenas prácticas de QA puede reducir los defectos en producción hasta un 50%.
- En las **metodologías ágiles**, la calidad es un valor intrínseco: se asume que el equipo está capacitado, colabora continuamente y busca mejorar el proceso en cada iteración (por ejemplo, a través de retrospectivas).
- Además, la **Gestión de la Configuración (SCM)** proporciona la base para controlar versiones, cambios y garantizar integridad, trazabilidad y consistencia del producto.

Calidad para Quién – Visiones de Calidad

A la calidad se la puede analizar desde distintas perspectivas o visiones:

- **Visión del usuario:** esto tiene que ver con las **expectativas** que tiene el **usuario** para con el producto, y si **este las satisface**. Esta es quizás la más complicada de establecer, porque está en la **cabeza de los usuarios**, lo cual es una razón más por la que el **principio de comunicación** constante es crucial, para ir validando en todo momento si estamos en el camino correcto. El **agilismo** trata de incluir la visión de calidad del usuario

a través del rol de **Product Owner**, el cual es ocupado por una persona con capacidad de decisión del cliente.

- **Visión del producto:** esto se asocia al nivel de **satisfacción de los requerimientos** particulares de cada producto.
- **Visión del proceso (manufactura):** si el **proceso de desarrollo** utilizado es el correcto para el **producto** que se desea desarrollar, es decir, el proceso **aporta valor** al producto y no produce desperdicios.
- **Visión del valor:** encontrar un equilibrio en la **relación costo-beneficio**, para obtener siempre el mayor valor para el cliente posible y, obviamente generar ganancias con el desarrollo del software.
- **Visión Trascendental:** esta visión es un tanto **utópica**, ya que se asocia a objetivos **difíciles de alcanzar**. Por ejemplo 0 defectos en el producto, pero son aquellos objetivos que **motivan a seguir en busca de la mejora continua**.



La **calidad** es un concepto **subjetivo** que tiene que ver con, si para uno **cumple con las necesidades y expectativas**. Las expectativas son **implícitas** ya que son las cosas que quieras que abarque tu producto o servicio, pero no están dichas en ningún lado. Por ejemplo, que contenga una interfaz agradable el sistema, pero si no ocurren estas expectativas empiezan las disconformidades.

La **calidad subjetiva** de un sistema de software se basa principalmente en sus **características no funcionales**. Esto refleja la experiencia práctica del usuario: Si la funcionalidad del software no es lo que se esperaba, entonces los usuarios con frecuencia solos le darán la vuelta para realizar lo que necesitan y encontrarán otras formas de hacer lo que quieren. Sin embargo, si el software no es fiable o es muy lento, entonces es prácticamente imposible que los usuarios logren sus objetivos con el sistema.

Con demasiada frecuencia, la **causa real de los problemas** en la calidad del software no es una gestión deficiente, procesos inadecuados o capacitación de escasa calidad. Más bien, es el hecho de que **las organizaciones** deben **competir para sobrevivir**. Una empresa, puede estimar el esfuerzo requerido o prometer la **entrega rápida** de un sistema en plazos de tiempo ajustados, para lograr un acuerdo con el cliente. En consecuencia, al no haber suficiente tiempo para el desarrollo, es probable que el **software entregado tenga funcionalidades reducidas** o niveles más bajos de fiabilidad o rendimiento, por lo tanto, esto conlleva a que la **calidad** del software se vea **afectada**, en ese intento de lograr cerrar un acuerdo de negocio con el cliente que necesita el software.

Visiones de la Calidad – ¿Calidad para quién? La calidad no es un concepto único: depende del punto de vista desde el que se analice.

1. Visión del Usuario

- **Enfocada en las expectativas del usuario final.**
- **Lo importante es si el software satisface lo que el usuario esperaba.**
- **Ejemplo:** Un usuario puede considerar que una app no tiene calidad si es difícil de usar, aunque funcione bien.
- **En Agile**, esta visión se representa a través del **Product Owner**, quien actúa como voz del cliente y valida la calidad funcional en cada entrega.

2. Visión del Producto

- **Se evalúa la calidad según el cumplimiento de los requerimientos técnicos y funcionales.**
- **Ejemplo:** El sistema cumple todos los requisitos del pliego, aunque no sea muy intuitivo para el usuario.

3. Visión del Proceso (manufactura)

- **Se centra en el proceso de desarrollo:** si está bien diseñado y controlado, produce resultados consistentes.
- **Ejemplo:** Seguir un proceso Scrum disciplinado o un pipeline DevOps bien estructurado es señal de calidad de proceso.

4. Visión del Valor

- **Busca el equilibrio entre costo y beneficio**, maximizando el valor entregado al cliente.
- **Ejemplo:** No tiene sentido desarrollar un sistema perfecto pero inviable económicamente; debe ser “lo suficientemente bueno” para cumplir el objetivo.

5. Visión Trascendental

- **Se refiere a la calidad ideal o utópica, como alcanzar “cero defectos”.**

- Aunque sea inalcanzable, **sirve como guía para la mejora continua.**
 - □ Ejemplo: Adoptar un enfoque de mejora continua (Kaizen) para reducir defectos versión tras versión.
- Subjetividad de la Calidad** La calidad es **subjetiva**, ya que depende de las **necesidades y expectativas** de cada usuario.
- Las **expectativas implícitas** (no escritas) también definen la percepción de calidad.
 - Ejemplo: Que el sistema tenga una interfaz atractiva y tiempos de respuesta rápidos.
 - Si no se cumple, el usuario percibe "baja calidad" aunque las funciones estén correctas.

Además, la **calidad subjetiva** se asocia principalmente con **características no funcionales**, como:

- Rendimiento
- Confiabilidad
- Usabilidad
- Seguridad

□ Si el sistema **no es confiable o es lento**, el usuario no podrá usarlo efectivamente, aunque cumpla sus funciones básicas.

□ **Factores que afectan la calidad del software**

Muchas veces, la pérdida de calidad no se debe a falta de conocimiento, sino a **presiones del negocio**.

□ **Ejemplo práctico:**

Una empresa promete entregar un sistema en 3 meses (por razones comerciales), aunque sabe que el desarrollo real requiere 5. El resultado suele ser:

- Funcionalidades incompletas.
- Bajo rendimiento.
- Menor fiabilidad.

Esto demuestra que **la prisa y las estimaciones irreales deterioran la calidad**, ya que sacrifican tiempo y validación para "cerrar el trato".

□ **Conclusión general La calidad del software:**

- Es **un valor estratégico**, no un lujo.
- Se logra **desde el inicio del proyecto**, no con pruebas al final.
- Involucra **personas, procesos y cultura organizacional**.
- Se evalúa **desde múltiples perspectivas** (usuario, producto, proceso, valor y trascendencia).
- Requiere **prevención, medición, capacitación y compromiso** para sostenerla en el tiempo.

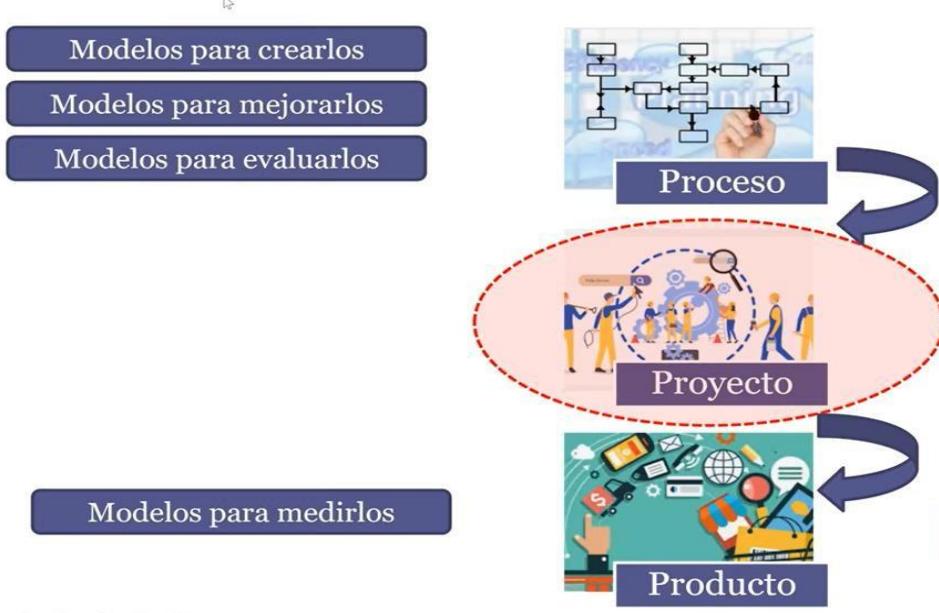
Calidad en el Software

La gente necesita **definir un proceso** para llevar a cabo el software, y generalmente en las organizaciones se basa en modelos para **tomar de referencia**.

Si quiero **funcionar con mejora continua** tengo **modelos** para **mejorar esos procesos**. También tenemos **modelos de evaluación** que ven el grado de adherencia del proceso al modelo que se tomó de referencia. Por ejemplo, si tomo la ISO 9001, si llamo a una auditoría de ISO esa auditoría debería decirte todas las **diferencias** que se tienen en ese **proceso definido respecto de lo que deberías tener**.

Los **procesos** se instancian en los **proyectos**. Los procesos son solo una indicación de cómo hacer las cosas, pero es el **proyecto** que al ejecutarse se **insertan actividades para ir regulando si lo que estoy haciendo es lo que se había pensado**.

Tenemos las **revisiones técnicas** que se hacen entre pares, no se somete a la persona a evaluación sino el producto. Se puede hacer sobre cualquier artefacto que queramos: requerimientos, etc. Luego tenemos las **auditorías** que son realizadas por **personas externas** (los empíricos están muy en contra porque creen que los equipos son capaces de reconocer y corregir por sí mismos, esto puede verse en la retrospectiva).



El producto que tengo que someter a evaluación es el que se va generando en el contexto del proyecto, por lo que tengo que **insertar tareas para asegurar la calidad del producto**. Ahí es donde aparecen varias técnicas como **revisiones técnicas** (realizado entre pares, con el propósito de detectar

tempranamente el defecto), **auditorías de configuración funcional** (la que valida una versión de la línea base), **auditorías de configuración física** (la que verifica una versión de la línea base), **Testing** (es para controlar la calidad cuando el producto ya está listo).

Los modelos de calidad dicen: hace tu proceso de manera que tenga calidad para vos (empresa), pero este proceso debe ser compatible con lo que yo te digo (modelo de referencia).

Calidad en el Software La calidad del software **no depende solo del producto final**, sino también del **proceso seguido para desarrollarlo**. Por eso, las organizaciones definen **procesos estandarizados** basados en **modelos de referencia**, que sirven como guías para asegurar que el desarrollo cumpla con buenas prácticas y que pueda **mejorarse de forma continua**.

Procesos y modelos de referencia

- **Un proceso de desarrollo** indica cómo deben hacerse las cosas: cómo se analizan requerimientos, cómo se programan, cómo se prueban, etc.
- Estos procesos se **instancian en proyectos concretos**, es decir, cada proyecto los aplica con sus propias particularidades.

Ejemplo: Una empresa puede definir un proceso basado en **ISO 9001 o CMMI**.

Luego, cada proyecto (por ejemplo, un sistema bancario o una app médica) aplica ese proceso adaptado a su contexto.

Evaluación y mejora del proceso

Para garantizar que el proceso se cumpla, existen **modelos de evaluación** que miden su grado de adherencia al modelo de referencia.

- Si una empresa se somete a una **auditoría ISO 9001**, el auditor analiza qué tanto se cumplen los requisitos del estándar y señala las **no conformidades** (diferencias entre lo definido y lo que realmente se hace).

Ejemplo: El proceso documentado indica que debe haber revisión de código entre pares, pero en la práctica no se realiza → **no conformidad**.

Esa diferencia se usa para mejorar el proceso (mejora continua).

Revisões y auditorías

Durante el desarrollo se realizan distintas actividades de control de calidad:

Tipo de control	Quién lo realiza	Propósito	Ejemplo
Revisión técnica	Entre pares (compañeros)	Detectar errores tempranos en cualquier artefacto (código, requerimientos, etc.)	Un desarrollador revisa el diseño de otro antes de implementarlo.
Auditoría funcional	Externa (control de configuración)	Validar que la versión cumple con las funciones esperadas según la línea base	Verificar que la versión entregada cumple con las especificaciones aprobadas.
Auditoría física	Externa (control de configuración)	Comprobar que los elementos físicos (archivos, documentación, builds) coinciden con lo que se declara en la línea base	Verificar que el ejecutable entregado corresponde con la documentación oficial.
Testing	Equipo QA	Controlar la calidad cuando el producto ya está listo	Pruebas unitarias, de integración, funcionales o de aceptación.

Importante: Las revisiones técnicas son **preventivas** (buscan evitar defectos).

El testing es **detectivo** (encuentra defectos ya presentes).

Postura ágil frente a las auditorías

En metodologías ágiles, los equipos tienden a **autoevaluarse** a través de la **retrospectiva**, en lugar de depender de auditorías externas.

El objetivo es que el equipo sea capaz de **reconocer y corregir** sus errores sin necesidad de control externo, promoviendo autonomía y mejora continua.

Modelos de calidad y su compatibilidad

Los modelos de calidad (ISO, CMMI, etc.) no imponen un único proceso, sino que proponen **principios y criterios**.

Cada empresa debe adaptar su proceso para que sea **compatible** con ese modelo, pero manteniendo su propio estilo de trabajo.

Ejemplo: Una startup puede implementar integración continua y control de versiones con GitHub, cumpliendo con los principios ISO, aunque no use toda la estructura documental tradicional.

Calidad del producto

Definimos anteriormente que la **calidad** es el nivel de cumplimiento o **adecuación a las necesidades** (explícitas e implícitas) y para el caso del producto estas deberían estar **explicitadas** en los requerimientos. Es por esto que los **requerimientos son tan importantes**, porque si no están o están **mal definidos**, no tenemos contra qué **validar**. Esto quiere decir que para que los requerimientos nos sirvan estos tienen que ser **medibles** (verificables) y **objetivos** (no ambiguos).

La **gestión de calidad** en productos son **acciones** sistemáticas de las organizaciones para **lograr calidad**, las cuales requieren **equilibrio** entre:

- **Calidad programada:** los alcances del producto planificados.
- **Calidad realizada:** lo que realmente se ha desarrollado del producto.
- **Calidad necesaria:** **mínimas características** que el producto debe tener, para que este satisfaga los requerimientos.

En el **equilibrio** obtenemos las **expectativas de calidad** que esperamos del producto y todo lo que esté **por fuera** de esta es desperdicio o insatisfacción.



Calidad del Producto de Software La calidad del producto se refiere al **grado en que el software cumple con las necesidades y expectativas** (explicadas e implícitas) del cliente y los usuarios.

Esto depende directamente de la **calidad de los requerimientos**: si son claros, medibles y no ambiguos, se podrá validar si el producto los cumple.

Importancia de los requerimientos Los requerimientos deben ser:

- **Medibles**: se puede comprobar objetivamente su cumplimiento.

Ejemplo: "El sistema deberá responder en menos de 2 segundos".

- **Objetivos**: sin ambigüedades ni interpretaciones.

Ejemplo: "El sistema deberá ser fácil de usar" → esto es subjetivo; debería medirse con una encuesta o estándar de usabilidad.

Si los requerimientos no están bien definidos, **no hay base de validación**. No se puede determinar si el producto "tiene calidad" porque no se sabe qué debía cumplir.

Equilibrio de la calidad del producto La gestión de calidad del producto **busca un equilibrio entre tres dimensiones**:

Tipo de calidad	Descripción	Ejemplo
Calidad programada	Lo que se planificó o prometió entregar	Se planea que el sistema tenga login, registro y dashboard.
Calidad realizada	Lo que efectivamente se desarrolló	Se entrega login y registro, pero el dashboard tiene errores.
Calidad necesaria	Lo mínimo indispensable para que el producto sea útil y funcional	El login debe funcionar correctamente; sin eso, el sist no sirve.

El punto ideal es donde la calidad realizada se alinea con la programada y cubre la calidad necesaria.

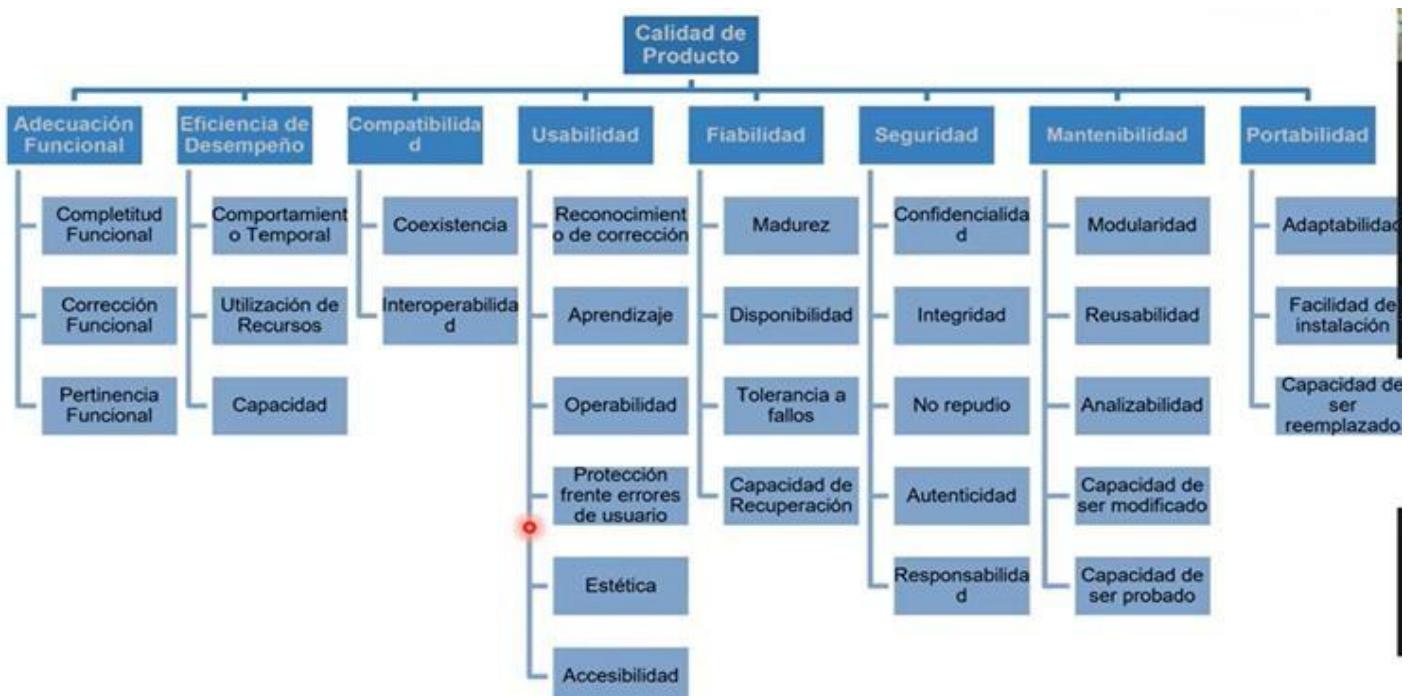
Todo lo que queda por fuera (exceso o defecto) genera **desperdicio o insatisfacción**.

Ejemplo: Si se agrega una funcionalidad que el cliente no pidió, eso puede ser desperdicio (tiempo y costo sin valor agregado).

Modelos de calidad de Producto

Al variar los requerimientos para cada producto en particular, **no existen modelos de calidad que acrediten para un determinado software qué nivel de calidad posee**. Si existen modelos como el Modelo de Barbacci, Calidad de Software (MCCALL) o la ISO 25.010 que permite **medir** epifenómenos como los **RNF** del software, para certificar calidad en ciertos aspectos (no los vemos en esta materia).

ISO 25000 -RNF



Modelos de Calidad de Producto

No existe un único modelo que determine el nivel de calidad de un software específico, porque cada producto tiene distintos requerimientos.

Sin embargo, existen **modelos de referencia** que ayudan a **evaluar y medir la calidad** en aspectos generales.

Principales modelos

1. Modelo ISO 25000 (antes ISO/IEC 9126 – SQuaRE)

Define un marco para evaluar la **calidad interna, externa y en uso** del software, basado en **Requerimientos No Funcionales (RNF)**.

Incluye características como:

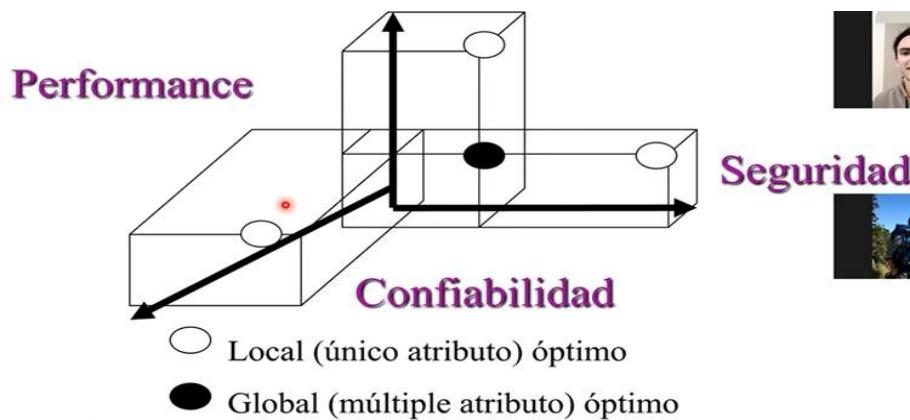
- **Fiabilidad (Reliability)** → el sistema no falla con frecuencia.
- **Usabilidad (Usability)** → facilidad de aprendizaje y uso.
- **Rendimiento (Performance Efficiency)** → tiempos de respuesta, uso de recursos.
- **Mantenibilidad (Maintainability)** → facilidad de corregir y actualizar.
- **Seguridad (Security)** → protección contra accesos no autorizados.
- **Portabilidad (Portability)** → facilidad para adaptarse a otros entornos.

Ejemplo:

Un sistema puede ser funcionalmente correcto pero tener baja calidad si su tiempo de respuesta es alto o si no es seguro.

Modelo de Barbacci / SEI

Busca el **equilibrio entre la Perfomance, la Confiabilidad y la Seguridad** para evaluar la calidad del producto.



2. Modelo de Barbacci / SEI

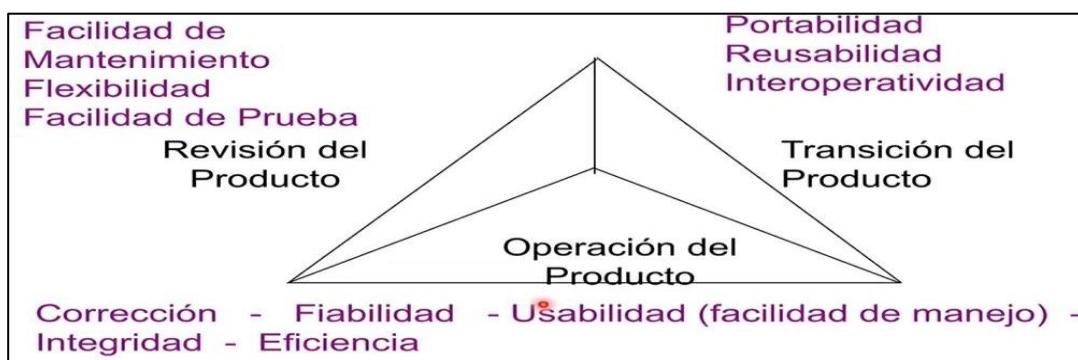
Propone una visión estructurada de la **arquitectura de calidad del software**, considerando atributos como:

- Rendimiento,
- Modificabilidad,
- Disponibilidad,
- Seguridad,
- Testeabilidad.

Se utiliza más en contextos de **evaluación arquitectónica** (cómo las decisiones técnicas impactan la calidad final).

Calidad del SW Mccall

Tiene tres grandes **agrupadores** unos relacionados a la **capacidad de verificación del producto** (revisión del producto), otras con el **producto en uso** (operación del producto) y otras con los factores que influyen que el producto **pueda crecer en el tiempo** (transición del producto)



3. Modelo de McCall (1977)

Uno de los primeros modelos formales de calidad de software. Busca el equilibrio entre tres dimensiones clave:

Dimensión	Qué mide	Ejemplo
Performance (Desempeño)	Eficiencia y rapidez del software	Que el sistema procese 1000 transacciones por minuto.
Reliability (Confiabilidad)	Estabilidad y frecuencia de fallos	Que el sistema no se caiga más de una vez al mes.
Security (Seguridad)	Protección de datos y accesos	Que los usuarios no puedan acceder a información ajena.

Agrupa los factores en tres categorías:

1. **Operación del producto:** mide cómo se comporta el software en uso (confiabilidad, eficiencia, usabilidad).
2. **Revisión del producto:** mide la capacidad del software de ser probado o verificado (testeabilidad, modularidad).
3. **Transición del producto:** mide su capacidad de adaptarse o crecer (portabilidad, reusabilidad, mantenibilidad).

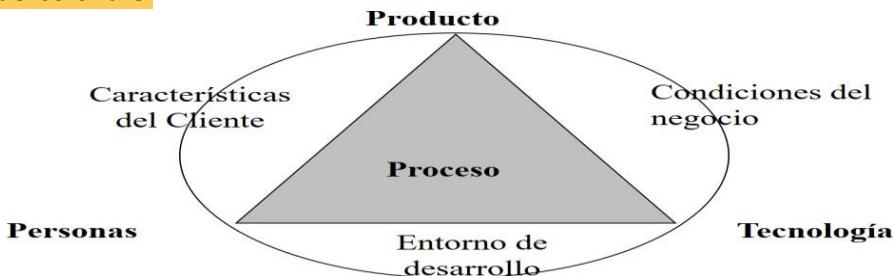
□ Ejemplo práctico: Un sistema de facturación que funciona rápido y sin errores (Performance + Reliability), pero es difícil de modificar (baja Mantenibilidad) → tiene buena calidad operativa, pero pobre calidad de transición.

□ Conclusión general

- La calidad del software combina proceso + producto:
un buen proceso genera productos más predecibles y confiables.
- Los modelos de referencia (ISO, McCall, Barbacci) ayudan a estructurar y medir la calidad desde distintos enfoques.
- La prevención temprana de errores mediante revisiones, auditorías y buenas prácticas reduce costos y mejora la percepción del cliente.
- Finalmente, la calidad del producto no depende solo de cumplir requerimientos, sino de equilibrar las expectativas reales del usuario, los objetivos del negocio y la sustentabilidad técnica del sistema.

Calidad del Proceso

- No existe en la realidad un proceso que sirva para todas las organizaciones y proyectos en general.
- La calidad de proceso nunca es un fin en sí mismo, lo que realmente nos interesa es la calidad de producto.
- Proceso con calidad → Producto con calidad
- Se insiste tanto en la calidad del proceso, debido a que es el único factor controlable para mejorar la calidad del software:



- El avance de las tecnologías que se utilizan para construirlo es ajeno a la organización y no es posible controlarlo.
- Las características y necesidades del cliente tampoco se pueden modificar.
- Las personas involucradas en el proceso de desarrollo son difíciles de controlar también.

Se aplica calidad en el producto y en el proceso. No se habla de aseguramiento de calidad en el proyecto, dado que esta se encuentra implícita por el hecho de que el proyecto implementa un proceso. Para garantizar la calidad se realizan auditorías.

La calidad del producto se realiza mediante actividades de revisiones técnicas y de auditorías. Estas son las herramientas principales para el seguimiento.

Para obtener calidad en el producto se debe definir un proceso que debe ser conocido por todos los involucrados en el equipo, donde además de tener tareas técnicas (requisitos, análisis, diseño, etc.) se incorporan disciplinas transversales como SCM, QA, Planificación de Proyectos y su Seguimiento.

Objetivos de QA:

- Realizar los controles apropiados del software y de su proceso de desarrollo.
- Asegurar el cumplimiento de los estándares y procedimientos para el software y el proceso.
- Asegurar que los defectos en el producto, proceso o estándares son informados a la gerencia para que puedan ser solucionados.

Calidad del Proceso La calidad del proceso se refiere al conjunto de prácticas, estándares y actividades que aseguran que el desarrollo de software se realice de manera controlada, eficiente y orientada a obtener productos de calidad.

No existe un proceso universal que sirva para todas las organizaciones o proyectos. Cada organización debe definir su propio proceso en función de su contexto, tamaño, cultura y objetivos.

Es importante destacar que la calidad del proceso no es un fin en sí mismo, sino un medio para lograr la calidad del producto.

Proceso con calidad → Producto con calidad

Importancia del proceso en la calidad del software

La insistencia en lograr procesos de calidad se debe a que es el único factor realmente controlable dentro del desarrollo. Otros elementos —como las tecnologías disponibles, las características del cliente o las habilidades del equipo humano— son difíciles o imposibles de controlar directamente.

Por lo tanto, mejorar la calidad del proceso se convierte en la vía más efectiva para asegurar la calidad del software.

Aseguramiento de Calidad (QA)

En los proyectos no se habla de "aseguramiento de calidad del proyecto", ya que esta se encuentra implícita por el hecho de que el proyecto implementa un proceso definido.

Para garantizar la calidad, se realizan auditorías y revisiones que permiten verificar tanto el producto como el proceso.

Objetivos del QA:

- Realizar los controles apropiados del software y de su proceso de desarrollo.
- Asegurar el cumplimiento de los estándares y procedimientos establecidos.
- Detectar e informar defectos en el producto, proceso o estándares, para su corrección por parte de la gerencia.

Estándares de Gestión de Calidad del Software:

Estándares de Producto: Definen las **características** que todos los componentes del **producto deberían exhibir**. Por ejemplo: estilos/buenas prácticas de programación, estándares de documentación.

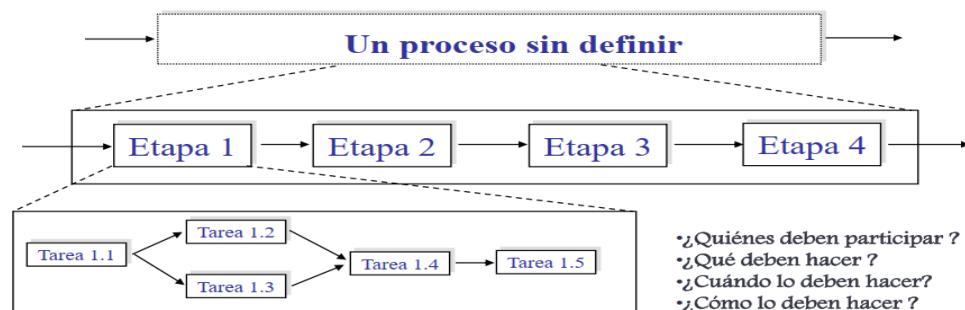
Estándares de Proceso: Establecen los **procesos que deben seguirse** durante el desarrollo, por ejemplo: incluyen definiciones de requerimientos, diseño, validación, etc. Definen como deben ser implementados los procesos de software.

Los estándares permiten unificar criterios y mantener coherencia en la construcción de software.

- **Estándares de Producto:** Definen las **características** que todos los componentes del **producto deben tener** (por ejemplo, convenciones de codificación o normas de documentación).
- **Estándares de Proceso:** Establecen los **pasos que deben seguirse durante el desarrollo** (por ejemplo, cómo definir requerimientos, diseñar, validar o probar).

Definición de Procesos

- Lo primero que se debe realizar en una organización para lograr tener un proceso de calidad, es **definirlo** de forma explícita y comunicarlo a todo el equipo, para que así todos tengan una base y el conocimiento de la forma de trabajar. Se deben definir aspectos como **etapas, subetapas, roles, qué se debe hacer, en qué momentos se deben hacer, cómo lo deben hacer, etc.**



- Dentro de esa definición, la cual plantea las **etapas para construir la parte técnica** (Ingeniería de Requerimientos, Análisis, Diseño, Implementación, Prueba y Despliegue), también se deben **incorporar disciplinas transversales** a ellas, como la **Planificación y Seguimiento de Proyectos, SCM y QA**, independientemente de la metodología adoptada (tradicional o empírica).
- El proceso definido y adoptado, **debe aportar valor al producto** (es posible utilizar **modelos de mejora de proceso** como Kanban) y **utilizarlos** en los distintos proyectos de forma **ADAPTADA**.
- La **adaptación** de los procesos se da en **aspectos negociables** del mismo. Por ejemplo, el realizar Testing no es algo negociable y que se pueda eliminar del mismo.

¿Cómo es un proceso para Construir Software?



El primer paso hacia un proceso de calidad es **definirlo explícitamente y comunicarlo** a todo el equipo. Esto incluye detallar:

- Etapas y subetapas del desarrollo.
- Roles y responsabilidades.
- Qué debe hacerse, cuándo y cómo.

Además de las tareas técnicas (requerimientos, análisis, diseño, implementación, pruebas y despliegue), se deben incorporar **disciplinas transversales** como:

- Planificación y seguimiento de proyectos.
- Gestión de la configuración (SCM).
- Aseguramiento de calidad (QA).

El proceso debe **aportar valor al producto**, y puede mejorarse continuamente con modelos como **Kanban** o prácticas de **mejora continua**.

Debe aplicarse de forma **adaptada a cada proyecto**, ajustando solo los aspectos negociables (por ejemplo, no se puede eliminar el testing).

Procesos definidos

En los **procesos definidos** se considera que el proceso es el **único factor controlable**, por lo tanto, se asume que la **mejora de la calidad continua** se realiza sobre el proceso. Si el proceso cuenta con calidad, entonces el **producto será de calidad**.

Todos los modelos de calidad están basados en **procesos definidos**, por lo que asumen que la calidad del producto se obtiene si se tiene un proceso de calidad para construir el producto.

*En los **procesos definidos**, se considera que la calidad del producto depende directamente de la calidad del proceso. La mejora continua se logra **optimizando el proceso**, bajo la premisa de que un proceso con calidad genera productos con calidad.*

*Todos los **modelos de calidad** (como CMMI, ISO, etc.) se basan en esta idea.*

Procesos empíricos

Los **procesos empíricos** están **basados en la experiencia**, por lo que **no consideran que la calidad en el proceso determine la calidad en el producto**. Por otra parte, **no están de acuerdo con las auditorías**, ya que asumen que los **equipos son autoorganizados**. Sin embargo, la **calidad** en estos procesos se lleva a cabo mediante **revisiones técnicas y la constante inspección y adaptación del trabajo**.

Según los empíricos, siempre que las personas hagan lo que tengan que hacer el producto tendrá calidad.

*Por otro lado, los **procesos empíricos** (como los ágiles) se basan en la **experiencia y adaptación constante**, sin depender estrictamente de un proceso documentado.*

*No suelen aceptar auditorías externas, ya que confían en la **autoorganización del equipo** y en la **inspección y adaptación continua** del trabajo.*

*En este enfoque, la calidad se garantiza mediante **revisiones técnicas, retrospectivas y mejora continua**, más que mediante controles formales.*

Según los empíricos, si las personas hacen lo que deben hacer y se mantienen los valores ágiles, el producto resultará con calidad.

Actividades relacionadas con el Aseguramiento de la Calidad del Software.

Administración de la calidad de Software

Busca **asegurar** que se **alcancen los niveles requeridos de calidad** para el **producto** de SW, **definiendo** **estándares y proceso** de calidad apropiados (que deben ser respetados por todos). El fin en sí de la Administración de Calidad de SW es **generar una cultura** de calidad y que esta sea vista como una responsabilidad de todos en el equipo.

La idea es **insertar** en las actividades **acciones tendientes a detectar lo más temprano posible oportunidades de mejora** sobre el **producto** y sobre el **proceso**. Hay que tener ciertas **consideraciones** respecto al Reporte del Grupo de Aseguramiento de Calidad (GAC):

- No debería reportar la gente que hace calidad al mismo gerente que reporta los proyectos (porque le quita independencia y libertad).
 - El **grupo de aseguramiento de calidad** debería **depender** del gerente general.
- Cuando sea posible, el grupo de aseguramiento de calidad debería reportar a alguien realmente interesado en el aseguramiento de calidad.

El **Aseguramiento de la Calidad del Software (SQA)** comprende el conjunto de actividades sistemáticas y planificadas que buscan garantizar que los procesos y productos de software cumplan con los estándares y niveles de calidad requeridos.

El propósito principal es **promover una cultura de calidad**, entendiendo que esta es responsabilidad de **todos los miembros del equipo**, y no solo del área de control o testing.

Administración de la Calidad de Software: La **Administración de la Calidad de Software** tiene como objetivo asegurar que se alcancen los niveles de calidad definidos para el producto, estableciendo **estándares, procesos y procedimientos**

apropriados que deben ser respetados por toda la organización.

Su fin último es **instalar una cultura de calidad** que promueva la mejora continua y la detección temprana de oportunidades de mejora tanto en el **producto** como en el **proceso**.

Consideraciones sobre el Grupo de Aseguramiento de Calidad (GAC)

- El equipo de calidad **no debe reportar** al mismo gerente que lidera los proyectos, para preservar la independencia y objetividad.
- Idealmente, el **GAC debería depender de la gerencia general**.
- Siempre que sea posible, el grupo debería reportar a una figura realmente comprometida con la calidad.

Entre las actividades que desempeña la administración de calidad, se encuentran:

- **Aseguramiento de calidad:** define estándares, procesos, procedimientos y modelos de calidad sobre los cuáles se van a realizar las comparaciones.
- **Planificación de Calidad:** se debe planificar la calidad, qué actividades se van a llevar a cabo, cuándo. Se selecciona los estándares aplicables para nuestro proyecto en particular y se los modifica si fuera necesario. Esta actividad abarca determinar los productos de SW que se desea que tengan calidad y determinar sus atributos de calidad más significativos.
- **Control de Calidad:** es la ejecución de lo planificado, y se revisa en qué situación está el proyecto. Asegura que los procedimientos y estándares son respetados por el equipo de desarrollo de SW. Existen dos enfoques para el control de calidad:
 - Tipos de revisiones de calidad:
 - Inspecciones para remoción de defectos (producto);
 - Revisiones para evaluación de progreso (producto y proceso);
 - Revisiones de Calidad (producto y estándares).
- **Evaluaciones de SW** automáticas y mediciones.

Actividades de la Administración de Calidad

1. **Aseguramiento de Calidad**

Define los estándares, procesos, procedimientos y modelos de calidad que servirán como referencia para la evaluación del software.

2. **Planificación de la Calidad**

Implica planificar las actividades de calidad, determinando qué se va a evaluar, cuándo y cómo.

Incluye la selección y adaptación de los estándares aplicables al proyecto, así como la identificación de los **atributos de calidad más significativos** para los productos de software involucrados.

3. **Control de Calidad**

Consiste en la ejecución de lo planificado y en la verificación de que los procedimientos y estándares definidos se estén cumpliendo.

Se busca detectar desviaciones y tomar medidas correctivas.

Existen dos enfoques principales:

- **Revisiones de Calidad:**

Método central de validación de procesos y productos. Consisten en el examen detallado de artefactos, documentación o entregables, con el fin de detectar posibles defectos o áreas de mejora.

Tipos de revisiones:

- Inspecciones para remoción de defectos (producto)
- Revisiones para evaluación del progreso (producto y proceso)
- Revisiones de calidad (producto y estándares)

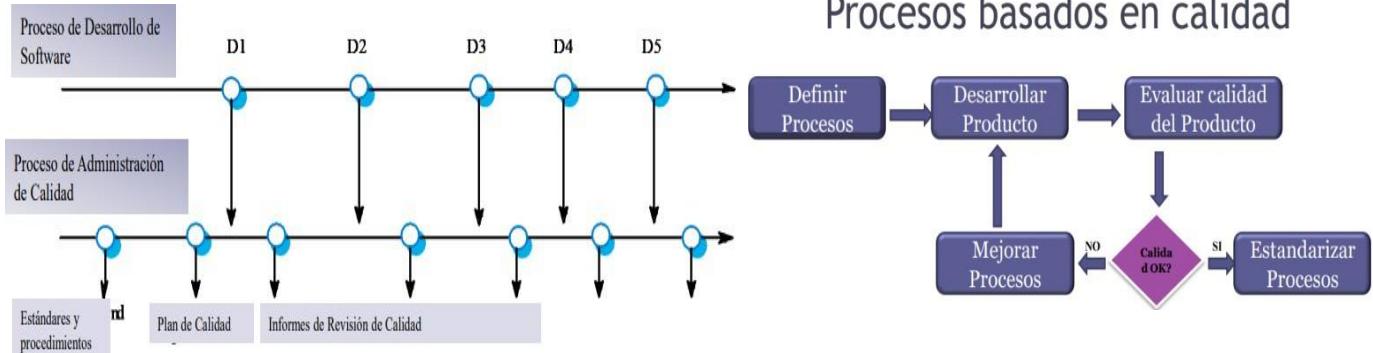
- **Evaluaciones automáticas y mediciones:**

Uso de herramientas automáticas y métricas cuantitativas para evaluar el cumplimiento de los estándares de calidad.

Entre las **funciones del Aseguramiento de Calidad de SW**:

- Prácticas de aseguramiento de calidad.
- Evaluación de la planificación del proyecto de SW.
- Evaluación de requerimientos.
- Evaluación del proceso de diseño.
- Evaluación de las prácticas de programación
- Evaluación del proceso de integración y prueba del software
- Evaluación de los procesos de planificación y control de proyectos.
- Adaptación de los procedimientos de calidad para cada proyecto.

Procesos basados en calidad



Funciones del Aseguramiento de Calidad de Software

Entre las principales funciones del SQA se encuentran:

- Implementación de prácticas de aseguramiento de calidad.
- Evaluación de la planificación del proyecto de software.
- Evaluación de requerimientos, diseño y prácticas de programación.
- Evaluación del proceso de integración y prueba del software.
- Evaluación de los procesos de planificación y control de proyectos.
- Adaptación de los procedimientos de calidad a las particularidades de cada proyecto.

Calidad de procesos en la práctica: ¿cómo garantizamos calidad de proceso en la práctica? Cumpliendo con algunas de las siguientes tareas:

- Definir proceso estándares tales como:
 - Cómo deberían conducirse las revisiones;
 - Cómo debería realizarse la administración de configuración, etc.
- Monitorear el proceso de desarrollo para asegurar que los estándares sean respetados.
- Reportar en el proceso a la Administración De Proyectos y al responsable del SW.
- No use prácticas inapropiadas simplemente porque se han establecido los estándares.

Calidad de Procesos en la Práctica Garantizar la calidad del proceso en la práctica implica realizar tareas concretas de control, definición y seguimiento:

- Definir procesos estándar, tales como:
 - Cómo deben conducirse las revisiones.
 - Cómo debe realizarse la administración de la configuración.
- Monitorear continuamente el proceso de desarrollo para asegurar que los estándares se cumplan.
- Reportar los resultados de estas verificaciones tanto a la administración de proyectos como a los responsables del software.
- Evitar la aplicación mecánica de prácticas inapropiadas solo por cumplir con los estándares: la calidad debe ser razonable, contextual y aportar valor.

Principales Modelos de Calidad existentes (CMMI – SPICE – ISO) y sus métodos de evaluación.

Lineamientos para la implementación de modelos de calidad en las organizaciones.

Modelos para la mejora de procesos

La mejora continua de procesos significa comprender los procesos existentes y cambiarlos para incrementar la calidad del producto o reducir los costos y el tiempo de desarrollo. Los procesos definidos están de acuerdo con esta definición, ya que consideran que la calidad del producto final depende de la calidad del proceso.

Los modelos NO te dicen cómo hacer las cosas. Son modelos descriptivos

El propósito de los modelos de mejora es analizar el proceso que tiene la organización y armar un proyecto cuyo resultado, en lugar de ser un producto, es un proceso mejorado que se vuelve de nuevo a la organización con la idea de que se produzca un producto mejor.

La mejora continua de procesos consiste en analizar, comprender y optimizar los procesos existentes dentro de una organización con el objetivo de aumentar la calidad del producto o reducir costos y tiempos de desarrollo.

Estos modelos parten de una premisa fundamental:

La calidad del producto final depende directamente de la calidad del proceso que lo genera.

Sin embargo, es importante aclarar que los modelos no te dicen cómo hacer las cosas —no son guías prescriptivas—, sino descriptivas, es decir, establecen buenas prácticas y niveles de madurez que sirven como referencia para que cada organización adapte su propio método.

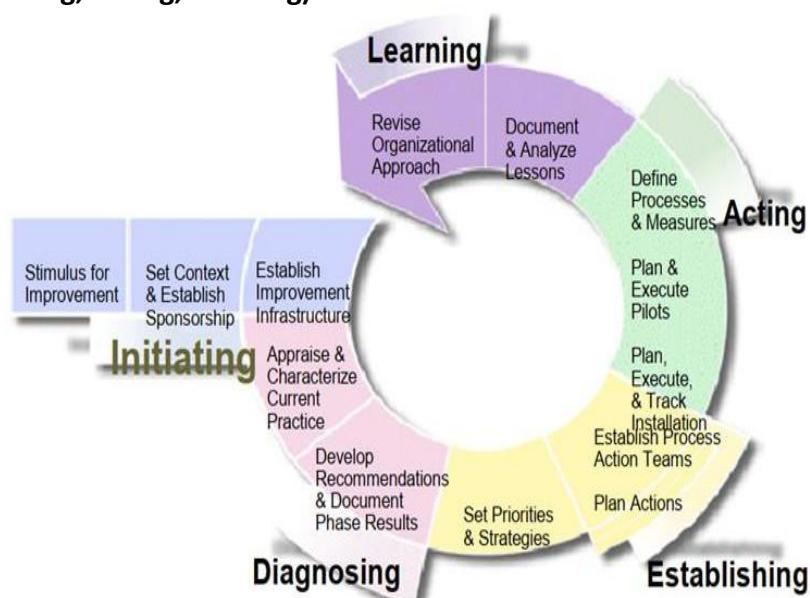
Ejemplo: Una empresa puede usar un modelo de referencia como CMMI o SPICE para evaluar qué tan maduros son sus procesos de desarrollo (si están documentados, medidos, estandarizados, etc.) y luego aplicar mejoras basadas en los resultados de esa evaluación.

El propósito de estos modelos es **crear proyectos de mejora** cuyo resultado no es un producto, sino **un proceso optimizado**, que luego se reutiliza dentro de la organización para generar productos de mayor calidad.

Modelo IDEAL (Initiating, Diagnosis, Establishing, Acting, Learning)

Nos da el contexto para crear un proyecto cuyo resultado va a ser un proceso definido. Es un modelo cíclico que mejora el proceso existente en una organización.

En el inicio empieza buscando un sponsor (apoyo en la organización (económico)). Esto es importante porque una mejora de proceso nunca es crítica, siempre hay algo más importante, entonces si no tengo un aval para ejecutar este tipo de proyectos, suelen no terminar exitosamente. Debemos entonces analizar dónde estamos y dónde queremos ir (se le llama análisis de brecha).



Se trata de un modelo de mejora de procesos que sirve como guía para inicializar, planificar e implementar acciones de mejora. Su nombre se debe a las 5 fases que lo componen:

- **Inicialización:** se reconocen las necesidades de cambio en la organización, razones para iniciar, determinar metas buscadas al proponer un cambio en el proceso. Se requiere que la organización apoye esta decisión de mejora (sponsoreo).
- **Diagnóstico:** Establecer la madurez actual de la organización y los riesgos asociados al proceso de mejora (revisar estado actual y futuro de la org.).
- **Establecimiento:** Durante la fase se elabora un plan detallado con acciones específicas, entregables y responsabilidades para el programa de mejora basado en los resultados del diagnóstico y en los objetivos que se quieren alcanzar. Para elaborar el plan se parte de definir las prioridades para el esfuerzo de mejora.
- **Ejecutar/Acción:** Efectuar los cambios y reunir información para aprender de la mejora. Se implementan las acciones planeadas en un proyecto piloto (no en todos los procesos de la organización, ya que es una prueba). Si la solución es satisfactoria para la org, se implanta en la empresa.
- **Aprendizaje:** busca garantizar que el próximo ciclo sea más efectivo. Durante la misma se revisa toda la información recolectada en los pasos anteriores y se evalúan los logros y objetivos alcanzados para lograr implementar el cambio de manera más efectiva y eficiente en el futuro. Extrapolar la mejora al resto de proyectos en caso de que sea exitosa la ejecución o realizar correcciones en caso de que fracase el proyecto piloto.

El modelo IDEAL es un modelo cíclico de mejora de procesos, desarrollado por el SEI (Software Engineering Institute), que sirve como guía práctica para planificar, implementar y evaluar proyectos de mejora dentro de una organización.

Su nombre proviene de las cinco fases que lo componen:

1. **Initiating (Inicialización)**
 - Se reconocen las necesidades de cambio y se definen los objetivos de mejora.
 - Es fundamental contar con un sponsor (apoyo institucional o económico) para que el proyecto tenga respaldo.
 - Se realiza un análisis de brecha entre la situación actual y la deseada.

Ejemplo: la dirección detecta que los proyectos fallan por falta de control de versiones, por lo que decide iniciar un plan para mejorar la gestión de configuración.
2. **Diagnosis (Diagnóstico)**
 - Se analiza la madurez actual de los procesos y los riesgos del proyecto de mejora.
 - Se identifican debilidades y oportunidades.

Ejemplo: descubrir que no existen revisiones técnicas formales o métricas de calidad.
3. **Establishing (Establecimiento)**
 - Se elabora un plan detallado de acciones, entregables, responsables y plazos, basado en el diagnóstico.

- Se priorizan las áreas más críticas para la mejora.

Ejemplo: implementar un proceso de control de cambios y establecer capacitaciones para el equipo.

4. Acting (Ejecución / Acción)

- Se implementan los cambios **en un proyecto piloto**, para evaluar su efectividad antes de aplicarlo a toda la organización.
- Se recopilan datos y experiencias.

Ejemplo: probar el nuevo proceso de revisión de código en un solo equipo antes de extenderlo a todos.

5. Learning (Aprendizaje)

- Se analizan los resultados del piloto, identificando lecciones aprendidas.
- Si la mejora fue exitosa, se **extrapolala al resto de la organización**; si no, se corrige el enfoque.

Ejemplo: documentar buenas prácticas y estandarizarlas como procedimiento oficial.

□ Este modelo fomenta la **mejora continua**, ya que al finalizar un ciclo, los aprendizajes alimentan el siguiente, asegurando una evolución constante del proceso.

Modelo SPICE (Software Process Improvement Capability Evaluation)

Es un modelo creado para la mejora de procesos software (ISO 15504). Es una adaptación para la evaluación de procesos de desarrollo software por niveles de madurez, dividido en 6 niveles.

Es un modelo dual, teórico. Tiene 2 partes:

- **Modelo de Calidad**
- **SPICE + IDEAL**: son ambos modelos de mejora de proceso. Estos modelos de mejora se usan para crear Proyectos de Mejora para una organización. El resultado de este proyecto va a ser un proceso definido que se usará para hacer Proyectos.

Este modelo establece conjuntos predefinidos de procesos con objeto de definir un camino de mejora para una organización. En concreto, establece **6 niveles de madurez para clasificar a las organizaciones**.

Nivel	Estado
Nivel 0 - Organización inmadura	La organización no tiene una implementación efectiva de los procesos
Nivel 1 - Organización básica	La organización implementa y alcanza los objetivos de los procesos
Nivel 2 - Organización gestionada	La organización gestiona los procesos y los productos de trabajo se establecen, controlan y mantienen
Nivel 3 - Organización establecida	La organización utiliza procesos adaptados basados en estándares
Nivel 4 - Organización predecible	La organización gestiona cuantitativamente los procesos
Nivel 5 - Organización optimizando	La organización mejora continuamente los procesos para cumplir los objetivos de negocio

El modelo **SPICE** (ahora formalizado como ISO/IEC 15504) es un estándar internacional que define un **marco para evaluar y mejorar los procesos de desarrollo de software**.

Su propósito es **determinar el nivel de capacidad y madurez de los procesos de una organización**, con el fin de trazar un **camino estructurado de mejora**.

Características principales:

- Es un **modelo dual**, compuesto por:
 - Un **modelo de calidad**, que define las buenas prácticas y atributos que debe cumplir un proceso.
 - Un **modelo de evaluación**, que permite medir qué tan bien se cumplen dichos atributos.
- Define **6 niveles de madurez** para clasificar a las organizaciones según su capacidad de proceso.

Niveles de Madurez de SPICE:

1. **Incompleto** – El proceso no se realiza o no logra su propósito.
2. **Ejecutado** – El proceso logra su propósito, pero sin control formal.
3. **Gestionado** – El proceso se planifica y supervisa.
4. **Establecido** – Se utiliza un proceso definido y estandarizado.
5. **Predecible** – El proceso se mide y controla con métricas.
6. **Optimizado** – El proceso se mejora continuamente con base en datos.

Ejemplo práctico: Una empresa en nivel 1 desarrolla software exitosamente pero sin métricas ni documentación formal. Al aplicar SPICE, puede avanzar hacia nivel 3 implementando procesos definidos y medibles que garanticen consistencia entre proyectos.

Lineamientos para la Implementación de Modelos de Calidad

Para aplicar un modelo de calidad en una organización, deben considerarse los siguientes pasos:

1. **Compromiso gerencial**: Sin apoyo directivo (sponsor), los proyectos de mejora suelen fracasar.
2. **Diagnóstico inicial**: Evaluar el estado actual de los procesos para conocer el punto de partida.
3. **Definición de objetivos claros**: Determinar qué se busca mejorar: tiempos, defectos, satisfacción del cliente, etc.
4. **Plan de acción**: Definir tareas, responsables, recursos y plazos.
5. **Implementación piloto**: Probar los nuevos procesos en un entorno controlado.
6. **Medición y retroalimentación**: Analizar resultados, medir el impacto y ajustar en base a la experiencia.
7. **Cultura de calidad**: Fomentar el compromiso de todo el equipo y la mejora continua como valor organizacional.

Modelo de Calidad para evaluar procesos

Plantean una definición teórica (lineamientos) del proceso que se utilizará en una organización (con diferentes

niveles de detalle según lo que se necesite), para luego compararlo con la ejecución del proyecto donde se implementa (instancia) ese proceso de desarrollo. Esto permite medir qué tanto se está cumpliendo en el proyecto con lo que plantea la definición teórica del proceso de desarrollo (definición de roles, asignación de responsabilidades, actividades a realizar, etc.), a través de las auditorías de proyecto.

¡No es lo mismo que un estándar de proceso! El objetivo de estos modelos de calidad es acreditar que una organización tiene cierto nivel de madurez en su proceso de desarrollo adoptado, o bien que cierta área de esa organización tiene determinado nivel de madurez.

Existen un montón de modelos de calidad. Los más utilizados en Argentina son el CMMI, CMMI 2.0, etc.

Los **modelos de calidad** definen una **referencia teórica** sobre cómo deberían ser los procesos en una organización, para luego **compararlos con su ejecución real**.

Su objetivo principal es **evaluar el nivel de madurez y mejorar los procesos** que influyen en la calidad del producto.

□ Características principales

- **Descriptivos**, no prescriptivos: indican qué debe lograrse, no cómo hacerlo.
- Permiten **evaluar y acreditar** el nivel de madurez de una organización o área.
- Se utilizan junto con **auditorías y evaluaciones** para medir cumplimiento.
- A diferencia de los **estándares de proceso**, no establecen métodos fijos sino metas de mejora continua.

Capability Maturity Model Integration – CMMI

Es un **modelo** de calidad para la **mejora y evaluación de procesos** para el desarrollo, mantenimiento y operación de sistemas de software, que constituye un **conjunto de buenas prácticas** que son publicadas en modelos.

Es un modelo de **referencia**, son descriptivos. Te dice que tienes que alcanzar determinado objetivo, pero vos **definís** qué práctica vas a implementar para lograrlo.

Este modelo empírico **recopiló buenas prácticas** a lo largo de la historia de diferentes organizaciones, tomando diferentes prácticas de aquellas que han logrado desarrollos exitosos.

El objetivo es que pueda ser **usado para guiar la mejora de procesos en un proyecto, división o una organización completa**.

La **acreditación se logra evaluando de manera objetiva** (comparación con el modelo de evaluación SCAMPI) y **subjetiva** (Experiencia y pensamiento del evaluador) **el proceso, y el uso de este proceso instanciado en proyectos**.

Constelaciones

Existen 3 constelaciones o modelos de negocio que cubren los modelos de CMMI:

- **Desarrollo (CMMI-DEV)**: provee la guía para medir, monitorear y administrar los procesos de desarrollos de software. Tiene dos representaciones, por etapas y continua.
- **Adquisición (CMMI-ACQ)**: provee la guía para permitir seleccionar, administrar y adquirir productos y servicios a otra empresa que posee su propia forma de trabajo, delegando el control de ese proyecto para obtener el producto o servicio final. Es decir, cuando la empresa no hace cosas, sino que **contrata gente que haga las cosas/trabajos** por ellos (no es lo mismo que subcontratación de personal).
- **Servicios (CMMI-SVC)**: provee la guía para entregar servicios, externos o internos.

Estas constelaciones poseen los modelos, capacitaciones y toda la información que es necesaria para que las empresas puedan acreditar distintos niveles de madurez.



Es un modelo de referencia creado por el **SEI (Software Engineering Institute)** para **mejorar y evaluar los procesos** de desarrollo, mantenimiento y operación de software.

□ **Propósito** Proveer una **guía estructurada** para que las organizaciones mejoren sus procesos de manera gradual, obteniendo **productos más predecibles y de mayor calidad**.

Tipos de CMMI (Constelaciones)

1. **CMMI-DEV (Development)** → Desarrollo de software y sistemas.

2. **CMMI-ACQ (Acquisition)** → Gestión de adquisiciones y proveedores.
3. **CMMI-SVC (Services)** → Entrega y gestión de servicios.

Representaciones del CMMI-DEV

Por Etapas (Maturity Levels) Evalúa la **madurez organizacional completa**. Cada nivel implica haber cumplido con los anteriores.

Nivel	Nombre	Descripción
1	Inicial	Procesos caóticos, sin previsibilidad. Éxito depende de individuos.
2	Gestionado	Se planifican, ejecutan y controlan los procesos. Se gestionan requisitos y calidad básica.
3	Definido	Procesos documentados, estandarizados y comprendidos. Formación, revisiones y métricas detalladas.
4	Cuantitativamente Administrado	Se gestionan los procesos mediante métricas y análisis estadístico. Decisiones basadas en datos.
5	Optimizado	Mejora continua y uso de innovaciones tecnológicas. Enfoque en optimización del rendimiento.

Para alcanzar un nivel, la organización debe cumplir con las prácticas específicas y genéricas de ese nivel y de los inferiores.

Continua (Capability Levels) Evalúa **áreas de proceso individuales** (no toda la organización).

Permite mejorar de forma focalizada en procesos específicos como calidad, gestión de requisitos, o configuración.

Áreas de proceso

- Un área de proceso es un **conjunto de prácticas relacionadas** en una zona que, cuando se **implementan** en conjunto, **satisfacen un conjunto de objetivos** considerados importantes para hacer mejoras significativas en el mismo proceso.
- Existen en total **22 áreas** de procesos en todos los niveles de madurez, las cuales son iguales en ambas representaciones.
- **16 de esas 22 áreas de procesos son comunes a todas las constelaciones CMMI.**
- **Nivel 1:** no existen **áreas de proceso**, ya que se considera que la organización está en estado de inmadurez.
- **Nivel 2:** son **7 en total**, de las cuales la última es opcional (depende si la organización realiza o no actividades relacionadas)
 - **REQM** → Requirement Management
 - **PP** → Project Planning
 - **PMC** → Project Monitor and Control
 - **MA** → Measure and Analytics
 - **PPQA** → Product and Process Quality Assurance
 - **SCM** → Software Configuration Management
 - **SAM** → Supplier Agreement Management

Gestión de Reqs Gestión de proyecto Gestión de proyecto Métricas de proyecto Disciplina transversal Disciplina transversal

Nivel	Categoría			
	Administración de Proyectos	Soporte	Administración de Procesos	Ingeniería
5 Optimizado		* Análisis y Causal y Resolución (CAR)	* Administración de Performance Organizacional (OID)	
4 Cuantitativamente Administrado	* Administración Cuantitativa del Proyecto (QPM)		* Performance del Proceso Organizacional (OPP)	
3 Definido	* Administración de Riesgos (RSKM) * Administración Integrada de Proyectos (IPM)	* Análisis y Resolución de Decisión (DAR)	* Definición del Proceso Organizacional (OPD) * Foco en el Proceso Organizacional (OPF) * Capacitación Organizacional (OT)	* Desarrollo de Requerimientos (RD) * Solución Técnica (TD) * Integración de Producto (PI) * Verificación (VER) * Validación (VAL)
2 Administrado	* Administración de Requerimientos (REQM) * Planificación de Proyectos (PP) * Monitoreo y Control de Proyectos (PMC) * Administración de Acuerdo con el Proveedor (SAM)		* Aseguramiento de calidad de Proceso y de Producto (PPQA) * Administración de Configuración (CM) * Medición y Análisis (MA)	
1 Inicial		Procesos sin definir o improvisados		

Representaciones CMMI-DEV

- **Por etapas:** CMM (el antecesor a CMMI), se basaba mucho en por etapas. Identificaba a las organizaciones y las dividía en maduras (eran las del nivel del 2 al 5) e inmaduras (nivel 1), mientras más maduras más capacidad de lograr sus objetivos, bajando sus riesgos. La ventaja de esta es que evaluaba la organización (el área que se quería trabajar). Esto facilita la comparación entre organizaciones. (madurez organizacional).



Es necesario cumplir con los lineamientos definidos para cada área de proceso de los niveles inferiores, y de las áreas de proceso definidas en el nivel que se está acreditando. Es decir, la acreditación es acumulativa para los niveles inferiores.

Niveles que se acreditan en la representación por etapas:

- **Inicial:** Las organizaciones en este nivel no disponen de un ambiente estable para el desarrollo y mantenimiento de software. Aunque se utilicen técnicas correctas de ingeniería, los esfuerzos se ven minados por falta de planificación. El éxito de los proyectos se basa la mayoría de las veces en el esfuerzo personal, aunque a menudo se producen fracasos y casi siempre retrasos y sobrecostes. El resultado de los proyectos es impredecible. Nivel de las organizaciones inmaduras.
- **Administrado:** la organización ha logrado todos los objetivos genéricos y específicos del nivel de madurez 2, es decir, los proyectos de la organización han asegurado que los requisitos son gestionados y de que los procesos se planifican, se realizan, se miden y controlado. También, hay un razonable seguimiento de la calidad.
- **Definido:** la organización ha alcanzado todos los objetivos específicos y de las áreas de proceso asignadas a los niveles de madurez 2 y 3. En este nivel los procesos están bien caracterizados y entendidos, y se describen en las normas, procedimientos, herramientas y métodos. Aquí los procesos se describen con más detalle y más rigurosidad que en el nivel de madurez 2. También se realiza formación del personal, técnicas de ingeniería más detalladas y un nivel más avanzado de métricas en los procesos. Se implementan técnicas de revisión de a pares.
- **Cuantitativamente administrado:** Se caracteriza porque las organizaciones disponen de un conjunto de métricas significativas de calidad y productividad, que se usan de modo sistemático para la toma de decisiones y la gestión de riesgos. El software resultante es de alta calidad. Las medidas detalladas del rendimiento de los procesos son recogidas y analizadas estadísticamente.
- **Optimizado:** este nivel se centra en mejora continua del rendimiento de los procesos a través de los aumentos y mejoras tecnológicas innovadoras.

Los objetivos cuantitativos de mejora de procesos para la organización se establecen y se revisan de forma continua a fin de reflejar los cambios objetivos de negocio, y se utilizan como criterios para la administración de la mejora de procesos. El alcanzar estas áreas se detecta mediante la satisfacción o insatisfacción de varias metas claras y cuantificables.

Para ser de un nivel, se debe cumplir con los requisitos de ese nivel y con los de los anteriores.

Nosotros hacemos foco en el nivel 2, las áreas del proceso son:



- Administración de Requerimientos.
- Planeamiento de Proyectos.
- Monitoreo y Control de Proyectos.
- Administración de Acuerdo con el Proveedor.
- Medición y Análisis
- Aseguramiento de Calidad del Proceso y del Producto.
- Administración de Configuración.

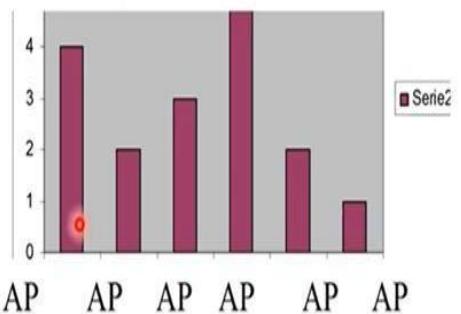
Administración de acuerdo con el proveedor no es obligatoria. Es necesaria si se contrata a una empresa externa como proveedora, ya que se debe comprobar que cumpla con el nivel de calidad que se tiene internamente. Hay 2 tipos de proveedores:

- Proveedor Tipo A: aquellos que cumplen con el mismo nivel de calidad.
- Proveedor Tipo B: aquellos que no cumplen con el mismo nivel de calidad.

Si alcanzo nivel 2 significa que la organización tiene la madurez para gestionar sus proyectos y que el producto que se producirá será algo esperable.

- Continua:** El objetivo del uso de esta representación, es acreditar la capacidad de la organización ante cada una de las áreas de proceso de forma individual. En lugar de medir la madurez de toda la organización, mido la capacidad de un proceso en particular. Procesos de nivel cero son los que no se ejecutan (Por ejemplo: si le pregunto a una empresa si hace gestión de configuración de software y me dice que no lo hace, entonces, es nivel 0 en esa área). Apunta a mejorar áreas de proceso que uno elige. Se centra en la mejora de un proceso o un conjunto de ellos relacionados a un área de proceso que una organización desea mejorar, una organización puede obtener la certificación para un área de proceso en cierto nivel de capacidad.

Continua



Roles – Grupos

Los roles se adaptan a las necesidades de la organización. Cuando hablamos de grupo no nos referimos a conjunto de personas, sino de alguien que asuma ese rol.

Lo importante es que exista alguien responsable de cubrir las actividades de cada uno de los roles o grupos.

Conclusión Integradora Cuándo aplicar cada modelo de calidad del software

La elección del modelo de calidad más adecuado depende del grado de madurez de la organización, el tipo de proyectos que desarrolla y los objetivos estratégicos que persigue en relación con la calidad.

CMMI – Para organizaciones que buscan madurez y estandarización

- Cuándo aplicarlo:**
Ideal para empresas medianas o grandes de desarrollo de software que desean institucionalizar procesos estables, predecibles y repetibles.
Es especialmente útil cuando se trabaja con clientes corporativos o licitaciones internacionales, donde se exige demostrar un nivel de madurez formal.
- Ventajas:**
 - Brinda reconocimiento internacional y comparabilidad.
 - Permite medir y mejorar la madurez organizacional.
 - Favorece la gestión de riesgos y la mejora continua.
- Limitaciones:**
 - Implementación costosa y extensa en tiempo.
 - Requiere fuerte compromiso gerencial y capacitación permanente.

Recomendado para: organizaciones con procesos establecidos que buscan optimizarlos o certificarse a nivel global.

SPICE (ISO/IEC 15504) – Para organizaciones que buscan flexibilidad técnica

- Cuándo aplicarlo:** Adecuado para empresas en crecimiento o áreas específicas que desean evaluar y mejorar procesos técnicos concretos, sin necesidad de una certificación completa de madurez organizacional.
Es muy útil cuando se quiere medir la capacidad de un proceso puntual (por ejemplo: pruebas, mantenimiento, SCM, etc.).
- Ventajas:**
 - Flexible y adaptable a distintos contextos.
 - Permite mejorar gradualmente los procesos elegidos.
 - Compatible con otros modelos (CMMI, ISO 9001).
- Limitaciones:**
 - Menor difusión comercial.
 - Requiere conocimiento técnico para realizar evaluaciones correctas.

Recomendado para: organizaciones que quieren mejorar procesos específicos antes de abordar una certificación integral.

ISO 9001 – Para organizaciones que buscan gestión global de la calidad

- Cuándo aplicarlo:** Ideal para organizaciones de cualquier rubro o tamaño que necesitan demostrar a clientes o entes reguladores que cuentan con un sistema de gestión de calidad documentado y controlado.
En software, puede aplicarse al nivel organizacional (no técnico) y complementarse con CMMI o SPICE.
- Ventajas:**
 - Amplio reconocimiento comercial.
 - Mejora la gestión administrativa y documental.
 - Puede coexistir con otros modelos.
- Limitaciones:**
 - No garantiza la calidad técnica del producto.
 - Se centra en la gestión organizacional más que en la ingeniería de software.

Recomendado para: organizaciones que buscan una certificación reconocida en gestión de calidad global.

Síntesis final

Situación de la organización	Modelo más conveniente
Busca estandarizar y certificar la madurez de todos sus procesos	CMMI
Desea evaluar o mejorar un proceso específico (p. ej., testing, SCM, QA)	SPICE / ISO 15504
Necesita certificar su sistema de gestión general de calidad	ISO 9001
Ya posee ISO 9001 y quiere enfocarse en procesos técnicos de software	ISO 9001 + CMMI o SPICE (combinado)

□ Conclusión general

La calidad del software no depende de aplicar un único modelo, sino de integrar prácticas que aseguren la mejora continua de los procesos y productos.

Mientras ISO 9001 asegura la gestión organizacional, SPICE permite medir la capacidad de procesos técnicos, y CMMI proporciona una ruta estructurada hacia la madurez organizacional completa.

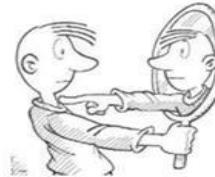
En conjunto, estos modelos permiten construir una cultura de calidad sólida, sostenible y orientada a la excelencia.

Relación de CMMI con Ágil

CMMI cara a cara con Ágil

- “Nivel 1”
 - Identificar el alcance del trabajo
 - Realizar el trabajo
- “Nivel 2”
 - Política Organizacional para planear y ejecutar
 - Requerimientos, objetivos o planes
 - Recursos adecuados
 - Asignar responsabilidad y autoridad
 - Capacitar a las personas
 - Administración de Configuración para productos de trabajo elegidos
 - Identificar y participar involucrados
 - Monitorear y controlar el plan y tomar acciones correctivas si es necesario
 - Objetivamente monitorear adherencia a los procesos y QA de productos y/o servicios
 - Revisar y resolver aspectos con el nivel de administración más alto

Referencias:
Verde : Da soporte,
Negro: Neutral,
Rojo: Desigual



No coincide con el monitoreo objetivo pues hace referencia a las auditorías realizadas por agentes externos al equipo. Para poder usar CMMI siendo ágil, ambas deben ceder un poco: Ágil plantea de gestión ágil de requerimientos y no pide que haya un control de qué requerimientos tiene el producto en cada momento de tiempo, pero CMMI sí. De alguna forma se debe tener trazabilidad de los requerimientos en US y demás para CMMI. Así como CMMI acepta no tener el registro de las Daily por ejemplo. Ágil debe ceder Auditorías.

- “Nivel 3”
 - Mantener un proceso definido
 - Medir la performance del proceso
- “Nivel 4”
 - Establecer y mantener objetivos cuantitativos para el proceso
 - Estabilizar la performance para uno o más subprocesos para determinar su habilidad para alcanzar logros
- “Nivel 5”
 - Asegurar mejora continua para dar soporte a los objetivos
 - Identificar y corregir causa raíz de los defectos

Negro:
Rojo:



¿Por qué no coincide a medida que subimos de nivel? CMMI dice que tienes que definir un proceso y que después lo tienes que cumplir (lo que se verifica con las auditorías). En cambio, ágil en ningún momento evalúa que hayas seguido el proceso que definiste. CMMI a partir de nivel 3 o 4 plantea métricas/estadísticas de los proyectos y productos, y en base a la comparación de esas medidas se arma una medida del proceso. Ágil plantea que la experiencia no es extrapolable a otros proyectos.

Valores esenciales de cada metodología

CMMI	MÉTODOS ÁGILES
→ Medir y mejorar el proceso [Mejores Procesos ↓ Mejor Producto]	→ Respuestas a clientes → Mínima sobrecarga → Refinamiento de Requerimientos <ul style="list-style-type: none">- Metáforas- Casos de negocio
→ Características de las personas <ul style="list-style-type: none">- Disciplinados- Siguen reglas- Aversión al riesgo	→ Características de las personas <ul style="list-style-type: none">- Comfortable- Creative- Risk Takers
→ Comunicación <ul style="list-style-type: none">- Organizacional- Macro	→ Comunicación <ul style="list-style-type: none">- Person to Person- Micro
→ Gestión de Conocimiento <ul style="list-style-type: none">- Activos de proceso	→ Gestión de Conocimiento <ul style="list-style-type: none">- Personas

Características CMMI vs. Metodologías ágiles

CMMI	MÉTODOS ÁGILES
→ Mejora Organizacionalmente <ul style="list-style-type: none"> -Uniformidad -Nivelación 	→ Mejora en el Proyecto <ul style="list-style-type: none"> - Tradición Oral - Innovación
→ Capacidad/Madurez <ul style="list-style-type: none"> - Éxito por Predictibilidad 	→ Capacidad/Madurez <ul style="list-style-type: none"> - Éxito por darse cuenta de oportunidades
→ Cuerpo de Conocimiento <ul style="list-style-type: none"> - Cruzando dimensiones - Estandarizado 	→ Cuerpo de Conocimiento <ul style="list-style-type: none"> - Personal - Evolucionando - Temporal
→ Reglas de Atajo <ul style="list-style-type: none"> - Desalentadas 	→ Reglas de Atajo <ul style="list-style-type: none"> - Alentadas

CMMI	MÉTODOS ÁGILES
→ Comités	→ Individuos
→ Confianza del Cliente <ul style="list-style-type: none"> - En la Infraestructura del Proceso 	→ Confianza del Cliente <ul style="list-style-type: none"> - Sw funcionando, Participantes
→ Cargado al frente <ul style="list-style-type: none"> - Mover a la derecha 	→ Conducido por Pruebas <ul style="list-style-type: none"> - Mover a la izquierda
→ Alcance de la vista <ul style="list-style-type: none"> [Involucrado, Producto] <ul style="list-style-type: none"> - Amplio - Inclusivo - Organziacional 	→ Alcance de la vista <ul style="list-style-type: none"> [Involucrado, Producto] <ul style="list-style-type: none"> - Pequeño - Focalizado
→ Nivel de Discusión <ul style="list-style-type: none"> - Palabras - Definiciones - Duradero - Exhaustivo 	→ Nivel de Discusión <ul style="list-style-type: none"> - Trabajo en mano

En cuanto al enfoque

CMMI	MÉTODOS ÁGILES
→ Descriptivo	→ Prescriptivo
→ Cuantitativo <ul style="list-style-type: none"> - Número científicos y duros 	→ Cualitativo <ul style="list-style-type: none"> - Conocimiento tácito
→ Universalidad	→ Situacional
→ Actividades	→ Producto
→ Estratégico	→ Táctico
→ “¿Cómo lo llamaremos?”	→ “Sólo hazlo!”
→ Gestión de Riesgos <ul style="list-style-type: none"> - Proactiva 	→ Gestión de Riesgos <ul style="list-style-type: none"> - Reactiva

CMMI	MÉTODOS ÁGILES
→ Foco de Negocio <ul style="list-style-type: none"> - Interna - Reglas 	→ Foco de Negocio <ul style="list-style-type: none"> - Externo - Innovación
→ Predictibilidad	→ Performance
→ Estabilidad	→ Velocidad

Similitudes entre ambas

- Meta: organizaciones de alto desempeño;
- Ambas planean;
- Ambas son Consultant Money Makers (CMMs);
- Ninguna es completa;
- Ninguno es aplicable a cualquier proyecto;
- No nuevas ideas;
- Ambas tienen reglas (reglas == requerimientos del proceso):
 - o Incumplir tiene serias repercusiones;
 - o 'SEPG' (Grupo de proceso de ingeniería de software) & 'Política de Proceso'.

El modelo **CMMI (Capability Maturity Model Integration)** y las **metodologías ágiles** buscan ambos mejorar la calidad del software y el rendimiento del equipo, pero lo hacen desde **enfoques distintos**:

- **CMMI** enfatiza la **definición, documentación y cumplimiento de procesos**, con foco en la **madurez organizacional y la mejora continua medible**.
- **Ágil** se centra en la **adaptabilidad, la colaboración y la entrega continua de valor**, priorizando a las **personas y la comunicación por sobre los procesos formales**.

Esto genera tensiones naturales: mientras **CMMI exige trazabilidad, control y auditorías**, **Ágil promueve autonomía, flexibilidad y mínima burocracia**.

Compatibilidad y concesiones Para que **CMMI y Ágil puedan coexistir**, ambas partes deben **ceder** en algunos aspectos:

Aspecto	Enfoque CMMI	Enfoque Ágil	Punto de conciliación
Monitoreo y auditorías	Auditorías formales por agentes externos para asegurar cumplimiento del proceso.	Seguimiento informal, basado en retrospectivas y reuniones diarias.	Reemplazar auditorías externas por evidencias automáticas (repositorios, tableros, pruebas, métricas continuas).
Gestión de requerimientos	Control estricto y trazabilidad completa de requerimientos.	Gestión dinámica y evolutiva de historias de usuario.	Mantener trazabilidad mínima y automatizada (US → tareas → entregables).
Cumplimiento de procesos	Se debe definir y cumplir un proceso estándar, verificado con métricas y auditorías.	Se adapta el proceso según el contexto y la experiencia del equipo.	Documentar solo los procesos críticos y permitir ajustes ágiles dentro del marco definido.
Medición y mejora	A partir del Nivel 3, se requiere recopilar estadísticas de desempeño y madurez.	No se generalizan métricas ni experiencias entre proyectos.	Usar métricas livianas y adaptativas (velocidad, defectos, satisfacción del cliente).

En síntesis, **CMMI aporta estructura y mejora continua**, mientras que **Ágil aporta adaptabilidad y rapidez de respuesta**.

Por qué no coinciden en niveles avanzados de CMMI

A medida que una organización busca niveles de madurez más altos en CMMI (Nivel 3, 4 o 5), se exige:

- **Definir un proceso estándar y cumplirlo.**
- **Realizar mediciones estadísticas de desempeño.**
- **Estandarizar las lecciones aprendidas** entre proyectos.

Esto choca con el principio ágil que dice que **"cada equipo y proyecto son únicos"** y que **la experiencia no es extrapolable**.

Por lo tanto, **a mayor nivel de formalización de CMMI, menor flexibilidad para aplicar Ágil de forma pura**.

Sin embargo, es posible **integrar ambos** si se usa CMMI como **marco de mejora**, y Ágil como **forma de ejecución**.

Valores esenciales

CMMI	Metodologías Ágiles
Enfoque en procesos y madurez organizacional.	Enfoque en personas, equipos y producto.
Control y trazabilidad.	Adaptación y entrega continua.
Documentación y evidencia.	Comunicación directa y colaboración.
Mejora sistemática basada en métricas.	Mejora continua basada en retroalimentación.
Visión a largo plazo, procesos estandarizados.	Visión iterativa e incremental.

Similitudes entre ambos Aunque parecen opuestos, comparten una base común:

- Buscan **organizaciones de alto desempeño y calidad sostenida**.
- Ambos requieren **planificación y gestión consciente**.
- Ninguno es aplicable universalmente: deben **adaptarse al contexto**.
- Ambos imponen **reglas y disciplina**, ya que el incumplimiento puede impactar la calidad del producto.
- Cuentan con estructuras internas de control o mejora:
 - o En CMMI, el **SEPG** (Software Engineering Process Group).
 - o En Ágil, los **equipos autoorganizados** y las **retrospectivas** cumplen una función similar de mejora continua.

Conclusión

CMMI y Ágil **no son incompatibles**, pero **responden a prioridades distintas**.

CMMI busca la **madurez de los procesos organizacionales**, mientras que Ágil prioriza la **eficiencia y adaptabilidad del equipo**.

Cuando se integran adecuadamente, permiten lograr **procesos maduros sin perder agilidad**, combinando **disciplina y flexibilidad** para alcanzar productos de software de alta calidad.

Diferentes tipos de Auditorías: Auditorías de Proyecto y Auditorías al Grupo de Calidad.

Auditorías de calidad de Software

Es una actividad incluida dentro de las disciplinas de soporte, la cual implica una evaluación independiente (realizada por un grupo de trabajo que no tienen nada que ver con el equipo del proyecto) de productos o procesos de software que permiten asegurar el cumplimiento con estándares, lineamientos, especificaciones y procedimientos basada en un criterio objetivo incluyendo documentación. Las auditorías implican esfuerzo y costo para los proyectos, sin embargo, sus beneficios son superiores. Son un instrumento para el Aseguramiento de Calidad en el Software.

Beneficios de las auditorías

- Se obtienen opiniones objetivas e independientes;
- Permiten identificar áreas de insatisfacción potenciales del cliente;
- Permite asegurar que se están cumpliendo con las expectativas;
- Permite dar visibilidad sobre los procesos de trabajo;
 - Permite visualizar los puntos fuertes de una organización;
 - Permite identificar oportunidades de mejora;
- Da visibilidad a la gerencia sobre los procesos de trabajo;
- Determinan que se implementen de manera efectiva: el proceso de desarrollo organizacional y del proyecto, y las actividades de soporte.

Auditorías de Calidad de Software Concepto general : las auditorías de calidad son evaluaciones independientes para verificar que los procesos y productos de software cumplen con los estándares, procedimientos y requisitos establecidos.

Por ejemplo: Una empresa de desarrollo realiza auditorías trimestrales para verificar si los equipos cumplen con el proceso de control de versiones, revisiones de código y documentación de pruebas definido en su manual de calidad ISO 9001.

Tipos de auditorías

Auditoría de proyecto

Responsable de ver que el proyecto se haya ejecutado con el proceso que se definió. Esto bajo la premisa de que en un proyecto se debe realizar lo que define el proceso, obviamente adaptado a lo que sirve en ese contexto del proyecto particular (teniendo en cuenta que, para obtener una acreditación de calidad, se tiene que ajustar hasta cierto punto al modelo teórico de referencia).

Lo que se hace es tomar evidencias de lo que se está haciendo y contrastarlo con lo documentado en las ERS, arquitectura, diseño, etc. Acá puede haber diferencias respecto a metodologías tradicionales o empíricas. En SCRUM, por ejemplo, el mismo equipo realiza estas auditorías en la ceremonia de Sprint Retrospective. En cambio, en metodologías tradicionales, el auditor debe ser externo al proyecto y a veces a la empresa.

En resumen: se evalúa el proceso adoptado (definición teórica), pero esa adopción del proceso se da en un proyecto, por ende, la auditoría se realiza sobre el proyecto. Estas auditorías se llevan a cabo de acuerdo a lo establecido en las PACS (Plan de Aseguramiento de Calidad de Software). El objetivo de esta auditoría es verificar objetivamente la consistencia del producto a medida que evoluciona a lo largo del proceso de desarrollo.

Auditorías de Configuración Funcional

Valida que el producto cumpla con los requerimientos. La auditoría funcional compara el software que se ha construido (incluyendo sus formas ejecutables y su documentación disponible) con los requerimientos de software especificados en la ERS. El propósito de la auditoría funcional es asegurar que el código implementa sólo y completamente los requerimientos y las capacidades funcionales descriptos en la ERS.

El responsable de QA deberá validar si la matriz de rastreabilidad está actualizada.

Auditoría de configuración física

Valida que el ítem de configuración cumpla con la documentación técnica que lo describe (esto permiten la trazabilidad y la satisfacción de los requerimientos). La auditoría física compara el código con la documentación de soporte. Su propósito es asegurar que la documentación que se entregará es consistente y describe correctamente al código desarrollado. El PACS debería indicar la persona responsable de realizar la auditoría física. El software podrá entregarse sólo cuando se hayan arreglado las desviaciones encontradas.

Auditorías externas de Aseguramiento de Calidad (No sé si son un tipo más)

Son auditorías realizadas por un personal externo a la organización, a quienes se encargan de realizar las auditorías de calidad. Esto permite evaluar a los miembros de QA, para determinar si sus evaluaciones dentro de la organización se están realizando de forma correcta.

Auditorías de Proyecto Evalúan si el **proyecto se ejecuta según el proceso definido** por la organización.

Verifican que se cumplan los procedimientos de planificación, desarrollo, pruebas y control de cambios.

Ejemplo: En un proyecto de desarrollo de una app bancaria, el auditor revisa que todas las historias de usuario aprobadas estén documentadas en Jira, con su respectiva evidencia de prueba unitaria y funcional, y que los entregables coincidan con lo establecido en la ERS (Especificación de Requerimientos de Software).

Si detecta que algunas historias no tienen trazabilidad con pruebas, lo reporta como una no conformidad.

Objetivo: comprobar que el proyecto cumple con los procesos y estándares definidos.

Auditorías de Configuración Funcional Valida que el **producto cumple con los requerimientos funcionales establecidos**.

Ejemplo: Durante la entrega del sistema, el auditor compara los requerimientos funcionales del módulo “Gestión de Turnos” con la funcionalidad real implementada en la aplicación.

Si un requerimiento dice “el sistema debe permitir cancelar un turno con 24 h de antelación” y la app solo permite cancelar con 48 h, se registra una desviación funcional.

Objetivo: asegurar que el software implementa **exactamente** lo que se definió en los requerimientos, ni más ni menos.

Auditorías de Configuración Física Valida la **consistencia entre la documentación y el producto desarrollado**.

Ejemplo: El auditor revisa que el código fuente y los diagramas de arquitectura entregados coincidan. Si el diseño indica que la base de datos tiene una tabla “usuarios” con cinco campos, pero en el código se implementaron seis, hay una discrepancia de configuración física.

También se verifica que el manual de instalación refleje los mismos pasos que la versión del sistema realmente utiliza.

Objetivo: garantizar que la **documentación describe correctamente el producto final**.

Auditorías al Grupo de Calidad (Auditorías Externas) Evalúan la eficacia del **sistema de calidad** y de los **miembros de QA** dentro de la organización.

Ejemplo: Una consultora externa visita la empresa para auditar al equipo de QA. Revisa si las auditorías internas se realizan según el plan anual, si los reportes se completan en tiempo y forma, y si las no conformidades tienen seguimiento.

Si detecta que las auditorías internas no se están registrando formalmente, recomienda implementar un sistema de control de acciones correctivas (CAPA).

Objetivo: comprobar que el área de QA realmente está cumpliendo su función de aseguramiento de calidad.

Roles en una auditoría

- Auditado:** **Participa** en la auditoría, y es quien **propone la fecha** de la auditoría, **entrega evidencia**, contesta las dudas del auditor, propone planes de acción para las desviaciones encontradas, contesta el reporte de auditoría, propone el plan de acción para las deficiencias citadas en el reporte. Generalmente es el Líder de Proyecto, pero puede ser otro.
- Auditor:** puede ser una persona o dos y deben ser **de afuera del proyecto** que se está auditando. Puede ser el **grupo de aseguramiento de calidad**, que es el grupo que le da soporte y auditorías de estos tipos.
 - **Acuerda la fecha** de la auditoría.
 - **Comunica** el alcance de esta.
 - **Recolecta** y analiza la evidencia objetiva que es relevante y suficiente para tomar conclusiones acerca del proyecto auditado.
 - **Realiza la auditoría**.
 - **Prepara el reporte**.
 - realiza el seguimiento de los planes de acción acordados con el auditado.
- Gerente de SQA (Software Quality Assurance):** Es quien **prepara el plan de auditoría**, **calcula su costo**, **asigna recursos**, **resuelve las no-conformidades** entre auditor y auditado. (orientado a la gestión de la auditoría)

Roles en una Auditoría — con ejemplos

Rol	Función	Ejemplo
Auditado	Participa en la auditoría, entrega evidencias y responde preguntas.	El Líder del Proyecto muestra los informes de avance, los casos de prueba y el registro de bugs. También explica cómo gestionan los cambios de requerimientos.
Auditor	Evalúa objetivamente el proceso y los productos.	Un miembro del Grupo de Aseguramiento de Calidad (GAC) revisa si el proyecto sigue el procedimiento de control de versiones. Si detecta que el equipo hace commits sin pasar por revisión de código, genera una no conformidad.
Gerente de SQA	Planifica las auditorías, asigna recursos y resuelve conflictos.	El Gerente de SQA define el calendario de auditorías anuales, estima el tiempo que dedicará cada auditor y aprueba los informes finales. Si hay desacuerdo entre auditor y auditado, media la situación.

Comparación — Auditorías de Proyecto vs. al Grupo de Calidad

Aspecto	Auditoría de Proyecto	Auditoría al Grupo de Calidad
Alcance	Evalúa el cumplimiento del proceso en un proyecto específico.	Evalúa la eficacia del área de QA dentro de la organización.
Responsable	Interno (auditor del GAC o SQA).	Externo o de otra área independiente.
Ejemplo	Revisar si el equipo de desarrollo documentó correctamente las pruebas unitarias y los resultados de testing.	Evaluando si el equipo de QA aplica correctamente el procedimiento de auditorías internas y mantiene registros actualizados.
Resultado	Informe de desviaciones y plan de acción del proyecto.	Informe sobre la madurez del sistema de calidad y su efectividad organizacional.

□ Resumen general con ejemplos integrados

Tipo de auditoría	Qué valida	Ejemplo
De proyecto	Cumplimiento del proceso definido.	Verificar si el equipo de desarrollo sigue las políticas de control de cambios y documentación.
De configuración funcional	Que el software cumple los requerimientos.	Revisar si las funcionalidades implementadas coinciden con las especificadas en la ERS.
De configuración física	Que el código y la documentación sean coherentes.	Confirmar que los diagramas y manuales coincidan con el código real.
Al grupo de calidad / externa	Que el sistema de calidad se gestione correctamente.	Consultora externa revisa la efectividad de las auditorías internas de QA.

Proceso de Auditorías: Responsabilidades. Preparación y ejecución.

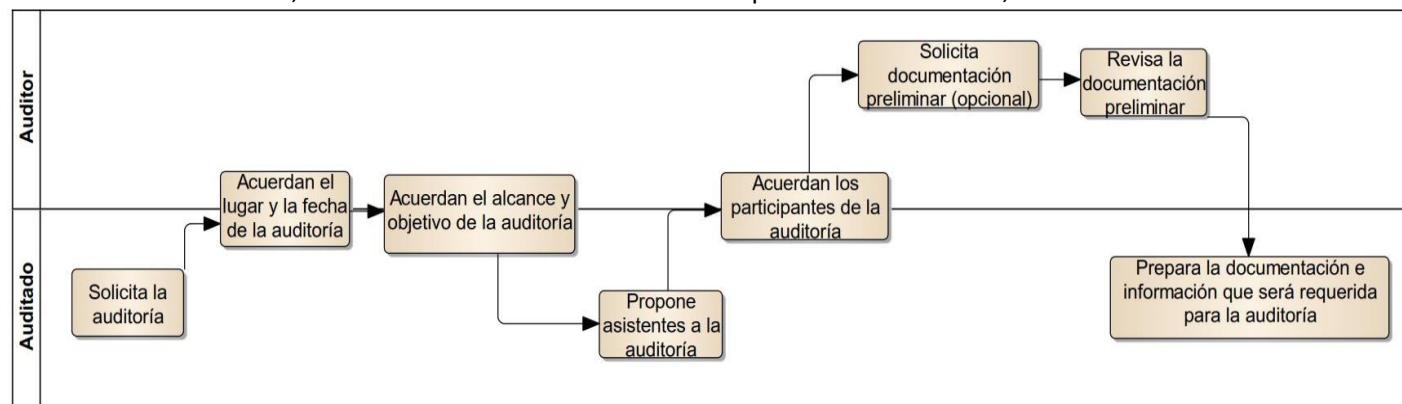
Reporte y seguimiento.

Etapas de una auditoría

1. Preparación y planificación

No son sorpresa. El **auditado solicita** la auditoría y junto con el **auditor definen la fecha**, el alcance y objetivo, acordando los participantes de esta.

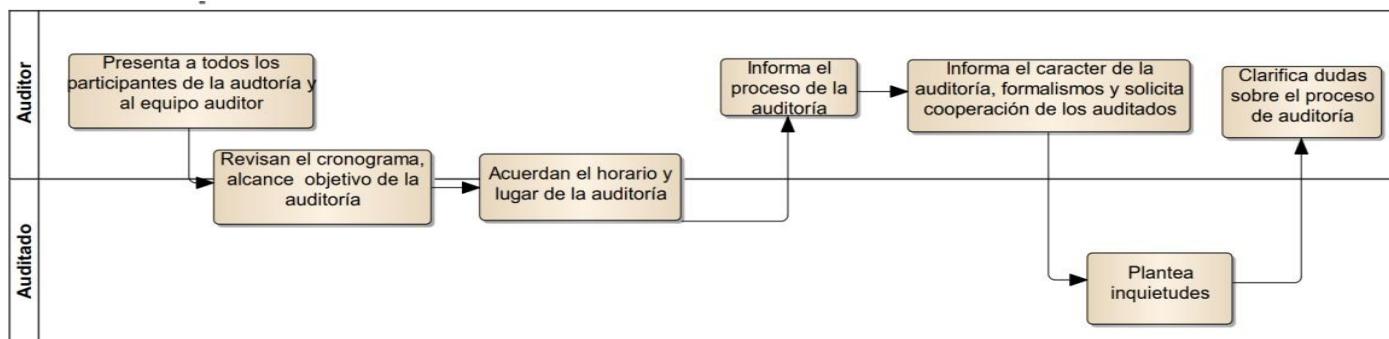
Se deben **definir métricas** de auditoría, como por ejemplo: esfuerzo por auditoría, cantidad de desviaciones, duración de la auditoría, cantidad de desviaciones clasificadas por auditor de CMMI, etc.



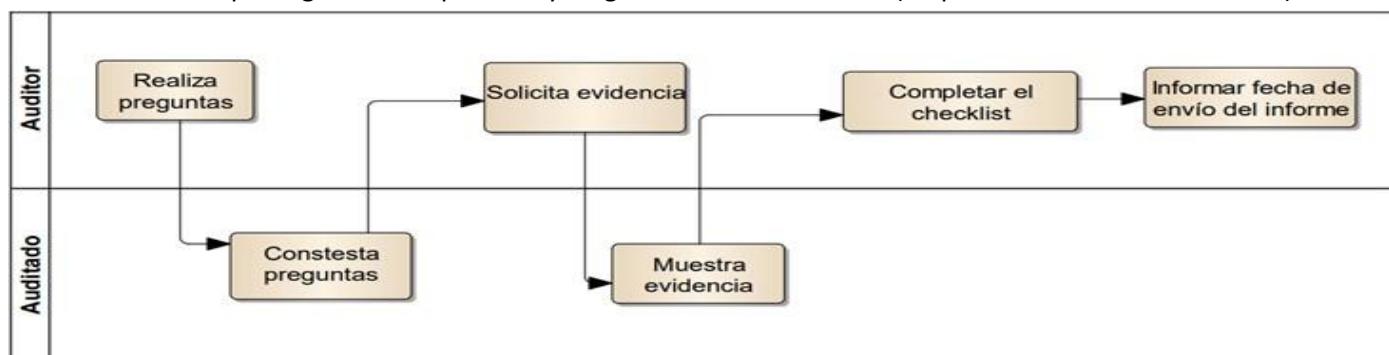
2. Ejecución

El auditor **escucha lo que la gente dice que hace** y luego ve la **documentación, pide evidencia** (lo que debería estar haciendo).

- **Reunión de apertura:** se informa cómo será el proceso, indicando el lugar y la hora.



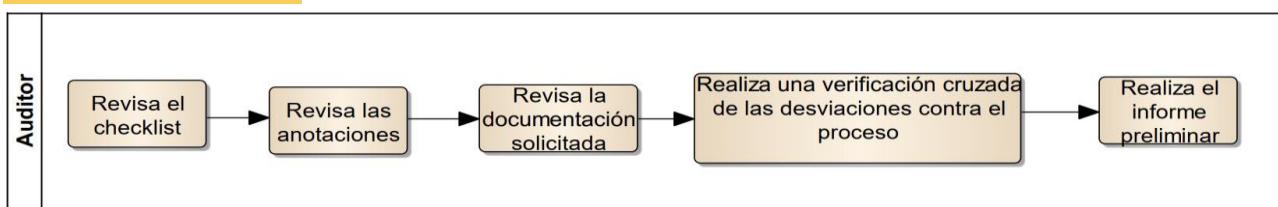
- **Ejecución:** se responden las preguntas, se muestra la evidencia y se completa el **checklist**. El auditor escucha lo que la gente dice que hace y luego ve la documentación (lo que debería estar haciendo).



3. Análisis y reporte de resultados

El auditor realiza el reporte de resultados y el auditado analiza la respuesta, puede o no estar de acuerdo con algunas prácticas y se deja asentado el documento final. Se evalúan los resultados, se hace una reunión de cierre y se hace entrega del reporte final.

- Evaluación de Resultados



- Reunión de cierre

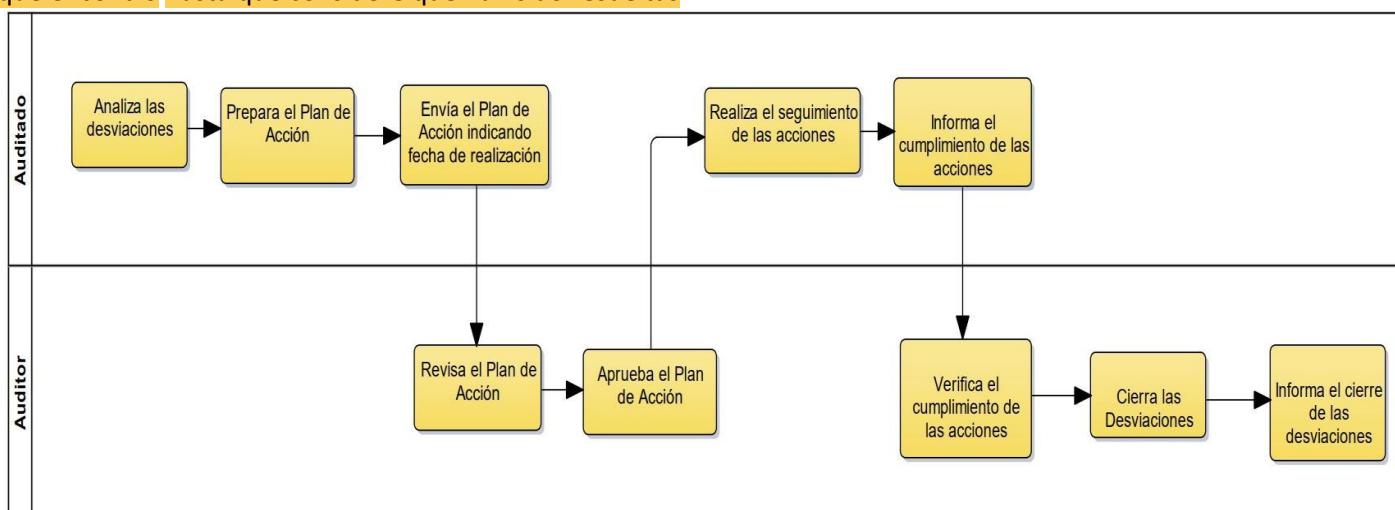


- Entrega de reporte final



4. Seguimiento

Se analizan las desviaciones y prepara un plan de acción que se envía al auditor para que lo revise y apruebe. En función del acuerdo entre auditado y auditor, puede que el auditor haga un seguimiento de las desviaciones que encontró hasta que considere que han sido resueltas.



□ Etapas de una Auditoría

1. Preparación y planificación Las auditorías **no son sorpresa**: el auditado solicita la auditoría y, junto con el auditor, **definen la fecha, el alcance y los objetivos**. Se acuerdan también los **participantes** y la **documentación necesaria**.

Se definen métricas de auditoría, por ejemplo:

- Esfuerzo por auditoría (en horas-persona)
- Cantidad de desviaciones encontradas
- Duración total de la auditoría
- Desviaciones clasificadas por tipo o gravedad (por ejemplo: leves, mayores, críticas)

Ejemplo: En un proyecto de software, el auditor y el líder de proyecto acuerdan auditar la gestión de requisitos. Se planifica la revisión de los registros de cambios, actas de reuniones y documentos de trazabilidad. Se estima una duración de 6 horas con dos participantes principales.

2. Ejecución Durante la ejecución, el auditor **verifica la práctica real** comparándola con lo documentado y los estándares.

Primero escucha lo que la gente dice que hace y luego **verifica la evidencia** de que efectivamente se hace así.

Pasos típicos:

- **Reunión de apertura:** se presenta el objetivo, el alcance, la metodología y el cronograma de la auditoría.
- **Ejecución:** se realizan entrevistas, se revisan documentos, repositorios, reportes de control de cambios, evidencias de pruebas, etc. Se completa un **checklist** de cumplimiento.

Ejemplo: El auditor entrevista al responsable de calidad, quien explica el proceso de revisión de código. Luego, el auditor revisa los commits en GitHub y los reportes de revisión para confirmar que se cumple el procedimiento.

3. Análisis y reporte de resultados

El auditor **elabora el informe de auditoría**, registrando los hallazgos, desviaciones y observaciones.

El auditado **revisa y puede manifestar su conformidad o desacuerdo**. Finalmente se deja asentado el **documento final** con los resultados.

Incluye:

- Evaluación de resultados
- Reunión de cierre (presentación de hallazgos y acuerdos)
- Entrega del reporte final

Ejemplo: En el informe final se detectan dos desviaciones menores (documentación desactualizada y falta de evidencia de revisión). En la reunión de cierre se acuerda que serán corregidas antes del próximo sprint.

4. Seguimiento

El auditado **prepara un plan de acción** para corregir las desviaciones detectadas y lo envía al auditor para su revisión.

El auditor puede **hacer un seguimiento posterior** para verificar que las acciones correctivas fueron implementadas eficazmente.

Ejemplo:

El equipo actualiza la documentación del procedimiento de revisión y adjunta evidencias en el repositorio. El auditor revisa los cambios y confirma que las desviaciones fueron cerradas.

Conclusión El proceso de auditoría permite **mejorar continuamente los procesos** y garantizar que se cumplan los estándares establecidos. En **ambientes ágiles**, las auditorías pueden adaptarse con un enfoque más **colaborativo y de mejora continua**, priorizando el aprendizaje sobre el control.

Herramientas y técnicas utilizadas en auditorías

- **Checklists:** tienen preguntas tipo para garantizar que independientemente de quién haga la auditoría el **foco** de las cosas que se controlan sea el mismo. Son de **mínima**, es decir, se debe garantizar de mínimo contestar estas preguntas, pero el auditor puede solicitar más cosas (hacer más preguntas).
- **Muestreo:** consiste en seleccionar una **muestra representativa de los productos y/o procesos** a auditar.
- **Revisión de registros**
- **Herramientas automatizadas**

Las auditorías se apoyan en diferentes herramientas y técnicas para garantizar la **objetividad, la consistencia y la eficiencia** del proceso de verificación.

Checklists (Listas de verificación)

Consisten en una serie de **preguntas tipo** que guían al auditor para revisar los aspectos clave del proceso o producto.

Aseguran que, **sin importar quién realice la auditoría**, los puntos de control sean los mismos.

Son **mínimos obligatorios**, pero el auditor puede realizar **preguntas adicionales** según la situación.

Ejemplo: En una auditoría de control de cambios, el checklist puede incluir preguntas como:

- ¿Se registra cada cambio en el sistema de control de versiones?
- ¿Cada cambio cuenta con aprobación previa?
- ¿Se documentan los motivos del cambio?

Muestreo

Técnica que consiste en **seleccionar una muestra representativa** de los productos, procesos o documentos a auditar, en lugar de revisar todo el universo de elementos.

Permite ahorrar tiempo y concentrarse en casos significativos.

Ejemplo:

De 50 historias de usuario cerradas durante el sprint, se seleccionan 10 al azar para verificar que todas cuenten con evidencias de pruebas y criterios de aceptación cumplidos.

Revisión de registros

El auditor **verifica documentación, reportes, actas o evidencias** generadas durante el proceso.

Se busca confirmar que los registros **reflejan fielmente** las actividades realizadas.

Ejemplo:

Revisar los informes de prueba para comprobar que se registraron los resultados de cada caso y que se corrigieron los defectos encontrados.

Herramientas automatizadas

En entornos digitales, se pueden usar **herramientas de software** que facilitan la auditoría, como sistemas de seguimiento, repositorios, herramientas de gestión de calidad o scripts de análisis.

Ejemplo:

Utilizar **SonarQube** para verificar la calidad del código o **Jira** para comprobar el flujo de trabajo y las aprobaciones en un proceso ágil.

Resultados/hallazgos de una auditoría

- **Buenas prácticas:** cuando es algo superador a lo definido que tenemos que hacer. Mucho mejor de los esperado. Ejemplo: se armó una muy buena herramienta para gestionar el plan de proyecto para que la gente pueda accederlo y mejorarlo.
- **Desviaciones:** cualquier cosa que no se hizo o que no se hizo cómo el proceso lo definió. Hay dos grados:
 - No Adecuado: lo que realmente está mal.
 - Necesita Mejora: si lo estás haciendo, pero necesita mejorarse.Requiere un plan de acción por parte del auditado.
- **Observaciones:** son cosas que se advierten por el auditor que no llegan a ser desviaciones pero que pueden llegar a ser riesgosas las prácticas y pueden llegar a generar un problema, por eso se destacan a consideración del equipo para que ellos decidan si hacerlo o no. Algo para revisar. Deberían mejorarse, pero no requieren un plan de acción. Ejemplo: técnica de estimación mala.

Los **hallazgos** son los resultados del análisis del auditor. Se clasifican en **buenas prácticas, desviaciones y observaciones**, según el grado de cumplimiento y mejora.

Buenas prácticas

Situaciones donde el proceso **superá la esperada** o se aplican soluciones innovadoras y efectivas.

Se destacan como **referencia positiva** para otros equipos.

Ejemplo: El equipo desarrolló una herramienta interna que automatiza la generación de informes del plan de proyecto, mejorando la transparencia y el acceso a la información.

Desviaciones

Cualquier situación en la que **no se cumple con lo definido en el proceso o los requerimientos**.

Se clasifican según su gravedad:

- **No Adecuado:** lo que está realmente mal o incumple de forma significativa.
- **Necesita Mejora:** se cumple parcialmente, pero debe optimizarse.

Requieren **un plan de acción** por parte del auditado.

Ejemplo:

- No Adecuado: no se realizó la revisión por pares del código antes del merge.
- Necesita Mejora: las revisiones se hacen, pero sin dejar registro en el sistema.

Observaciones

Aspectos **no críticos**, pero que **podrían representar riesgos** si no se corrigen.

Se informan para consideración del equipo, pero **no requieren plan de acción formal**.

Ejemplo: El auditor detecta que el equipo usa una técnica de estimación de esfuerzo poco precisa, lo que podría generar problemas de planificación en el futuro.

Reporte de auditoría

En este documento se deben informar las siguientes desviaciones:

- Cualquier **desviación** que resulta en la **disconformidad de un producto** respecto de sus requerimientos
- Cualquier **desviación al proceso definido** o a los requerimientos documentados.

El **reporte de auditoría** es el documento formal que resume los resultados.

Debe incluir toda la información relevante y las **desviaciones identificadas**.

Contenido mínimo:

- Desviaciones que causan **disconformidad del producto con sus requerimientos**.
- Desviaciones respecto de los **procesos definidos o los requerimientos documentados**.
- Resumen de **observaciones y buenas prácticas** detectadas.
- Conclusiones y recomendaciones.

Ejemplo: Se detectó una desviación crítica: falta de trazabilidad entre requerimientos y casos de prueba.

Se recomienda implementar un tablero de control que relacione ambos elementos antes del próximo release.

Métricas de auditoría

Cada organización establece según sea apropiada o no:

- **Esfuerzo por auditoría.**
- **Cantidad de desviaciones.**
- **Duración de auditoría.**

Cada organización puede definir sus propias métricas para **evaluar la efectividad** del proceso de auditoría y la **madurez del sistema de calidad**. **Ejemplos de métricas:**

- **Esfuerzo por auditoría:** cantidad de horas-persona invertidas.
- **Cantidad de desviaciones:** número total y tipo (mayores, menores).
- **Duración de auditoría:** tiempo total desde la preparación hasta el cierre.
- **Porcentaje de desviaciones cerradas:** mide la eficacia del seguimiento.
- **Frecuencia de auditorías por proyecto o área.**

Ejemplo: En una organización se registró:

- Duración promedio por auditoría: 8 horas
- 3 desviaciones por proceso auditado
- 100% de desviaciones corregidas dentro del plazo acordado

Calidad de Producto: Planificación de pruebas para el software- Niveles y tipos de pruebas para el software. Técnicas y herramientas para probar software. Técnicas y Herramientas para la realización de revisiones técnicas del software.

La **calidad de producto** en software se asegura mediante un conjunto de **actividades planificadas y sistemáticas** que buscan garantizar que el producto cumpla con los requerimientos del cliente y los estándares establecidos. Esto se logra a través de los procesos de **Verificación y Validación (V&V)**, **revisiones técnicas** y **pruebas de software**.

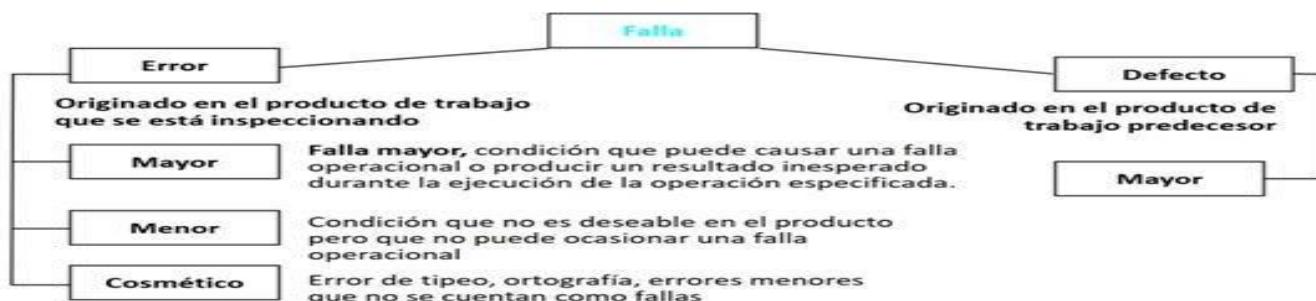
Verificación y validación

- Es un proceso de ciclo de vida completo.
- Inicia con las revisiones de los requerimientos y continúa con las revisiones del diseño, inspecciones del código hasta la prueba.
- Validación: ¿Estamos construyendo el producto correcto?
- Verificación: ¿Estamos construyendo el producto correctamente? Falla: error en un producto de trabajo.

Producto de trabajo: salida de cualquier actividad correspondiente al ciclo de vida de desarrollo.

¿Por qué existen las fallas?

- Problemas de comunicación.
- Limitaciones de memoria.



□ Verificación y Validación (V&V)

Concepto	Pregunta guía	Objetivo
Verificación	¿Estamos construyendo el producto correctamente?	Asegura que el producto cumpla con las especificaciones y estándares establecidos.
Validación	¿Estamos construyendo el producto correcto?	Asegura que el producto satisface las necesidades del usuario y los objetivos del negocio.

□ Características

- Es un **proceso que abarca todo el ciclo de vida** del software.
- Inicia desde la revisión de requerimientos hasta las pruebas finales.
- Se aplican tanto a **productos de trabajo** (documentos, código, diseños, etc.) como al producto final.

□ Causas comunes de fallas

- Problemas de comunicación.
- Requerimientos mal interpretados.
- Limitaciones de memoria o hardware.
- Errores humanos en el desarrollo o pruebas.

Principios

- Prevenir es mejor que curar.
- Evitar es más efectivo que eliminar.
- La retroalimentación enseña efectivamente.
- Priorizar lo rentable.
- Olvidarse de la perfección, no se puede conseguir.
- Enseñar a pescar, en lugar de dar el pescado.

Principios de V&V

1. **Prevenir es mejor que corregir:** detectar errores temprano ahorra tiempo y costos.
2. **Evitar es más efectivo que eliminar:** diseñar procesos que eviten errores.
3. **La retroalimentación enseña efectivamente:** los errores deben generar aprendizaje.
4. **Priorizar lo rentable:** enfocar esfuerzos en áreas críticas.
5. **Olvidarse de la perfección:** buscar calidad suficiente, no absoluta.
6. **Enseñar a pescar, no dar el pescado:** fomentar equipos autónomos en calidad.

Existen dos aproximaciones complementarias:

- Revisiones técnicas.
- Pruebas de Software.

Revisiones técnicas – Peer Review

Es una actividad realizada por un colega, cuyo propósito es mejorar la calidad de software, mediante la detección temprana de errores en cualquier artefacto que se genere, por ejemplo, en el código, requerimientos, diseño, arquitectura, riesgos, estimaciones, planes, etc.

Es un proceso estático de validación y verificación; y no corrige errores.

Objetivo: introducir el concepto de verificación y validación. Busca evitar el retrabajo. Motiva a realizar un mejor trabajo.

Nunca se pone en juicio el autor del artefacto, sólo el artefacto en sí, ya que es necesario que se revelen todos los errores entre los miembros del equipo. Se debe desarrollar una cultura de trabajo que brinde apoyo y no buscar culpables cuando se descubran errores.

Es una actividad que trascendió cualquier metodología, filosofía y en muchas ocasiones la utiliza ágil, para transformar las auditorías en peer reviews evitando así traer a alguien externo al equipo.

Por ejemplo, siendo un programador Jr, solicitar a un Sr una revisión técnica respecto a un patrón de diseño arquitectónico que se quiere implementar.

Ventajas:

- Pueden descubrirse muchos errores;
- Pueden inspeccionarse versiones incompletas;
- Pueden considerarse otros atributos de calidad.

Desventajas:

- Es difícil introducir las inspecciones formales;
- Sobrecargan al inicio los costos y conducen a un ahorro sólo después de que los equipos adquieran experiencia en su uso;
- Requieren tiempo para organizarse y “parecen” ralentizar el proceso de desarrollo.

Actividad realizada por colegas para detectar errores tempranos en cualquier artefacto del desarrollo (requerimientos, diseño, código, arquitectura, etc.).

Objetivo Prevenir defectos antes de que lleguen a etapas posteriores (evitar retrabajo).

Ejemplo: Un programador junior pide a un desarrollador senior revisar su implementación de un patrón arquitectónico antes de subir el código.

Ventajas

- Detecta errores en etapas tempranas.
- Puede aplicarse a versiones incompletas.
- Permite revisar aspectos de calidad no funcional (mantenibilidad, legibilidad, etc.).

Desventajas

- Dificultad inicial para implementarlas.
- Incrementan costos al principio.
- Requieren organización y tiempo (parece que ralentizan el desarrollo).

Tipos de revisiones

- **Formales:** tienen un proceso definido con roles.
 - **Inspecciones** (Inspección de código de Fagan e inspección de Gilb).
- **Informales:** cuando no existe un proceso de cómo realizarlo.
 - **Walkthrough o recorrido.**

Tipo	Formalidad	Ejemplo	Características
Informal (Walkthrough o recorrido)	No sigue proceso formal	Reunión entre colegas al finalizar una iteración	No se registran métricas, útil para metodologías ágiles
Formal (Inspección)	Sigue proceso definido con roles y checklist	Inspección de código de Fagan o Gilb	Se documenta, se miden resultados y se genera reporte

Walkthrough o recorrido (informal)

Técnica de análisis estático en la que un diseñador o programador dirige miembros del equipo de desarrollo y otras partes interesadas a través de un producto de software, donde los participantes formulan preguntas y realizan comentarios acerca de posibles errores, violación de estándares de desarrollo y otros problemas.

No existe un proceso formal. Consiste en reuniones informales de colegas donde se debaten las correcciones a aplicar al producto de trabajo. No hay control del proceso.

Tiene los siguientes objetivos:

1. Mínima Sobrecarga
2. Capacitación de Desarrolladores
3. Rápido retorno

Esta técnica no obtiene métricas para aprender y dejar registros. Sin embargo, es una de las técnicas más elegidas en los enfoques agiles. Hay una junta de revisión entre colegas después de completar cada iteración del software (una revisión rápida), en la que pueden exponerse los conflictos y problemas de calidad sobre el producto.

- Reunión informal entre colegas.
- Se formulan preguntas, se señalan errores o mejoras.
- Foco en capacitación y aprendizaje rápido.
- No deja registro formal ni métricas.

Ejemplo:Después de un sprint, el equipo realiza una revisión rápida del código desarrollado, debatiendo mejoras sin documentarlas.

Inspecciones (formal)

Tiene un proceso formal y cuenta con un conjunto de roles. Es necesario la utilización de un checklist, que ayuda a la memoria para saber que cosas controlar. Se toman métricas y finalmente se realiza un reporte de la revisión al final de la inspección para analizar los defectos encontrados.

Es una actividad que garantiza la calidad del software, cuyo éxito depende de la planificación. Tiene como objetivos:

1. Descubrir errores.
2. Verificar que el software alcanza sus requerimientos.
3. Garantizar que el software fue construido de acuerdo con ciertos estándares.
4. Conseguir un software desarrollado de manera uniforme.
5. Hacer que los proyectos sean más manejables.

Son procesos time boxing y exigen un alto esfuerzo intelectual.

- Proceso estructurado con roles definidos.
- Utiliza checklists para guiar la revisión.
- Se registran métricas y defectos encontrados.
- Requiere planificación, seguimiento y un reporte final.

Roles participantes

Al ser una técnica formal, si o si debe contar con los siguientes roles:

1 Autor: creador o encargado de mantener el producto a inspeccionar. Inicia el proceso seleccionando a un moderador y junto a este eligen al resto de los roles. Entrega el producto a ser inspeccionado al moderador.

2 Moderador: planifica y lidera la revisión. Trabaja junto al autor para elegir los demás roles. Entrega el producto a inspeccionar al inspector 2 días antes de la reunión. Coordina la reunión de forma tal que no ocurran conductas inapropiadas y realiza un seguimiento de los defectos encontrados.

3 Anotador: registra los hallazgos de la inspección. Usualmente termina confeccionado el reporte de la revisión.

4 Lector: lee el producto a ser inspeccionado. Este rol es necesario para que los participantes no se dispersen.

5 Inspector: Examina el producto antes de la reunión para encontrar defectos. Registra sus tiempos de preparación. todos pueden ser inspectores. Todos pueden inspeccionar.

Ciertos roles pueden ser asumidos por la misma persona.

Las revisiones técnicas...

SON	NO SON
<ul style="list-style-type: none"> ● La forma más barata y efectiva de encontrar fallas ● Una forma de proveer métricas al proyecto ● Una buena forma de proveer conocimiento cruzado ● Una buena forma de promover el trabajo en grupo ● Un método probado para mejorar la calidad del producto 	<ul style="list-style-type: none"> ● Utilizadas para encontrar soluciones a las fallas ● Usadas para obtener la aprobación de un producto de trabajo ● Usadas para evaluar el desempeño de las personas

Etapas del proceso de inspección

1. **Planificación:** el moderador, a pedido del autor, planifica la inspección definiendo el lugar, el tiempo de duración y los roles. La duración de las reuniones no debe superar las 2 horas, dado que la inspección es un proceso de alto esfuerzo intelectual.
2. **Visión general:** esta etapa es opcional. El autor realiza una descripción general del producto a inspeccionar.
3. **Preparación:** es la preparación de cada rol para la reunión. Cada rol adquiere una copia del producto de trabajo que deberá leer y analizar, con el fin de encontrar potenciales defectos. Esta preparación permite que la reunión de inspección sea más productiva.
4. **Reunión de inspección:** el equipo realiza un análisis para recolectar los potenciales defectos previos y descartar falsos positivos. El lector lee el producto de trabajo y los inspectores comparten los defectos encontrados, los cuales son registrados por el anotador. La reunión finaliza con una conclusión acerca de si se acepta o no el producto de trabajo inspeccionado. Finalmente se realiza un informe detallando que se revisó, por quien, que se descubrió y que se concluyó.
5. **Corrección:** finalizada la reunión, el autor realiza las correcciones de los defectos encontradas.
6. **Seguimiento:** Dependiendo de la gravedad puede existir un proceso de re-inspección. En caso de que los defectos sean graves, se realiza nuevamente una inspección. De lo contrario, si el defecto era muy simple, el autor simplemente lo corrige. Otros defectos que no implican una re-inspección, pueden implicar que el autor se reúna con el moderador únicamente para tratar las correcciones realizadas.
 1. **Planificación** → se asignan roles y se define duración.
 2. **Visión general (opcional)** → el autor explica el contexto.
 3. **Preparación** → cada participante analiza el producto.
 4. **Reunión de inspección** → se revisan los defectos, el anotador registra todo.
 5. **Corrección** → el autor realiza las mejoras necesarias.
 6. **Seguimiento** → se verifica que los defectos se hayan corregido.

Definición de estándares

Un aspecto importante del aseguramiento de calidad es la definición o selección de estándares que deben aplicarse al proceso de desarrollo de software o al producto de software.

Los estándares son importantes por tres razones:

- Se basan en conocimiento sobre la mejor o más adecuada práctica para la organización. Con frecuencia, este conocimiento se adquiere sólo después de una gran cantidad de pruebas y errores. Configurarla dentro de un estándar, ayuda a la empresa a reutilizar esta experiencia y a evitar errores del pasado.
- Al usar estándares se establece una base para decidir si se logró un nivel de calidad requerido. Desde luego, esto depende del establecimiento de estándares que reflejen las expectativas del usuario para la confiabilidad, la usabilidad y el rendimiento del software.
- Los estándares aseguran que todos los trabajadores dentro de una organización adopten las mismas prácticas. En consecuencia, se reduce el esfuerzo de aprendizaje requerido al iniciar un nuevo proyecto o trabajo.

Para la gestión de calidad de software se utilizan dos estándares:

- Estándares de producto: Se aplican al producto de software a desarrollar. Incluyen estándares de documentos y estándares de codificación.
- Estándares de proceso: establecen los procesos que deben seguirse durante el desarrollo, por ejemplo, incluyen las definiciones de requerimientos, procesos de diseño y validación, etc.

Estándares de Calidad de Software

Los estándares definen **cómo se debe desarrollar y documentar** el software para garantizar uniformidad y calidad.

Razones para definir estándares

1. Incorporan buenas prácticas aprendidas por la organización.
2. Permiten evaluar si se alcanzó el nivel de calidad esperado.
3. Fomentan uniformidad en los equipos y proyectos.

Tipos de estándares

Tipo	Aplicación	Ejemplo
De producto	Definen cómo debe ser el software final.	Estándares de codificación, documentación o interfaz.
De proceso	Establecen cómo se desarrolla el software.	Procedimientos para diseño, revisión, validación, etc.

Ejemplo real: Una empresa adopta el estándar de codificación **PEP8 (Python)** y el proceso de **revisión por pares obligatoria** antes de cada merge. Esto garantiza legibilidad y coherencia entre desarrolladores.

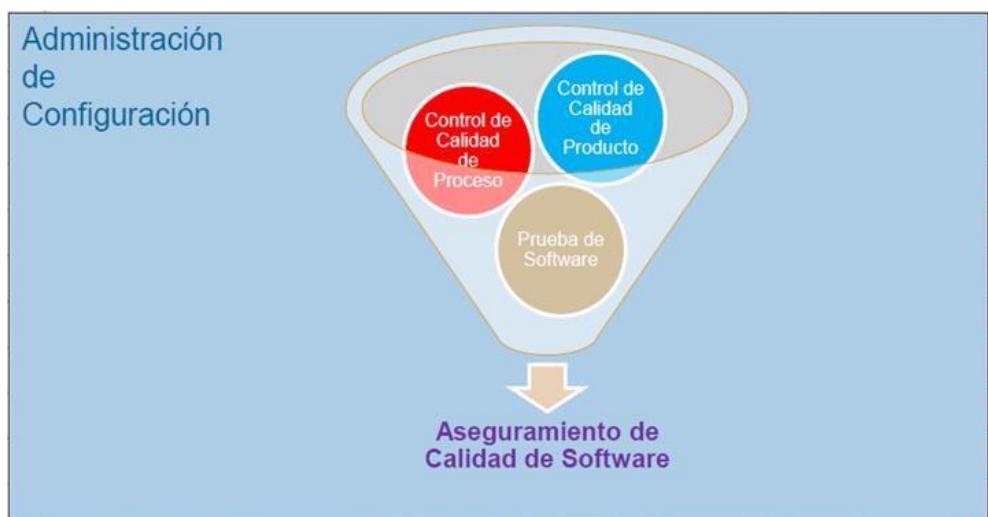
Testing en ambientes Ágiles.

Contexto

El testing de software, es una de las tareas a realizar en el ámbito del aseguramiento de calidad de software, en conjunto con el control de calidad del proceso y del producto. La calidad del software se obtiene y se logra a lo largo de todo el desarrollo de este (detectar errores en etapas tempranas es siempre menos costoso, que detectarlos después), por lo que una buena administración de configuración (SCM) es el puntapié inicial para permitir la realización de tareas de aseguramiento de calidad.

Hay que recordar que SCM permite determinar la configuración de ítems, trazabilidad, control de cambios, auditorías del producto y reportes de estado. QA agrega a esto, el control de calidad del proceso. Existen diversos estándares (ISO, CMMI, etc.) que permiten acreditar la calidad del proceso, debido a que la calidad del producto no es algo estandarizable por la variación de requerimientos de un caso a otro.

Dentro del aseguramiento de la calidad del software, existen actividades destinadas al control de calidad del proceso y del producto. Se realiza un control de calidad sobre el proceso de desarrollo, bajo el argumento de que el proceso de desarrollo posee un gran impacto en la calidad del producto. Es decir, en teoría la calidad del producto depende de la calidad del proceso que se está utilizando.



Luego de desarrollar el software, las actividades de testing revelan la calidad del Software en sí, es decir, si el producto satisface con los requerimientos definidos por el cliente en etapas anteriores. Cabe destacar que la actividad de testing no asegura la calidad por sí solo, sino que debe estar acompañada de las demás actividades. QA != Testing

El testing de software es una de las actividades fundamentales dentro del **aseguramiento de calidad del software (QA)**, junto con el control de calidad del proceso y del producto.

La calidad no se obtiene al final del proyecto, sino que se construye desde el inicio.

Por eso, detectar errores en etapas tempranas es siempre menos costoso que detectarlos después.

□ En este sentido, una buena Gestión de Configuración (SCM) es el punto de partida para garantizar la trazabilidad, control de cambios, auditorías y reportes del producto.

A esto, QA agrega el control de calidad del proceso, siguiendo estándares como ISO 9001, ISO/IEC 25010 o CMMI, que miden la calidad del proceso, ya que la calidad del producto depende directamente del proceso utilizado.

□ Relación entre QA y Testing

Concepto	Enfoque	Momento	Objetivo
QA (Quality Assurance)	Preventivo	Durante todo el ciclo de vida	Asegurar que el proceso sea de calidad
Testing	Detectivo	Una vez construido el producto o módulo	Detectar defectos en el producto

□ QA ≠ Testing

El testing forma parte del QA, pero no lo sustituye. QA busca evitar los defectos; el testing, encontrarlos.

Definición de Testing

Proceso mediante el cual se somete a un software o un componente de software a condiciones específicas con el fin de determinar y demostrar si el mismo es válido o no en función de los requerimientos especificados.

El testing es una actividad destructiva cuyo objetivo es encontrar defectos, cuya presencia es asumida de antemano en el código.

Testing de software es un proceso, o una serie de procesos, diseñados para asegurar que el código hace lo que fue diseñado para hacer, o visto de otra manera, que no haga nada que no esté intencionado. Se dice que es el proceso de ejecutar un programa con la intención de encontrar fallas. El software debería ser predecible y consistente, sin presentarle sorpresas a los usuarios.

El Testing es parte del QA (Quality Assurance) y forma parte del control de calidad. Se hace cuando el producto ya está construido (o lo que se desea testear), en cambio el QA se aplica en todo el ciclo de vida (las buenas prácticas en la implementación son parte de QA).

Las actividades de testing se consideran exitosas si se encuentran defectos en la ejecución de los casos de prueba. Por lo que, un software con alta calidad lograda a lo largo de todo el desarrollo (implementación de QA), dificulta encontrar defectos y puede dirigir a un testing no exitoso. Por otro lado, si el software tiene un bajo nivel de calidad, el costo de retrabajo es alto, ya que van a existir muchos idas y vueltas entre las actividades de testing y las de desarrollo, para la corrección de errores.

El testing es una actividad costosa, por lo que es necesario lograr un nivel de cobertura óptima teniendo en cuenta el costo-beneficio. Nunca se podrá cubrir el 100% de las pruebas, por cuestiones lógicas de tiempo y costos. Esta cobertura se comienza a definir desde el momento en los cuales se establecen los criterios de aceptación.

El testing es el proceso mediante el cual se somete un software a condiciones específicas para determinar si cumple los requerimientos definidos.

Es una actividad destructiva, porque su intención es romper el sistema para revelar defectos.

“Testing es ejecutar un programa con la intención de encontrar fallas.”

Un software de alta calidad (gracias a buenas prácticas de QA) puede resultar en un testing poco exitoso (porque se encuentran pocos errores).

Por el contrario, un software con baja calidad genera mucho retrabajo y costos elevados de corrección.

Costo y cobertura

El testing puede representar entre el 30% y el 50% del costo total del proyecto.

Por eso se debe buscar un nivel de cobertura óptimo, priorizando pruebas de alto impacto y riesgo.

□ La cobertura se empieza a definir junto con los criterios de aceptación de las historias de usuario (US).

Principios del Testing

- El Testing es una actividad destructiva que encuentra defectos cuya presencia se asume.
- Se testea con una actitud negativa tratando de demostrar que algo es incorrecto.
- El Testing exitoso es aquel que encuentra defectos.
- El costo del Testing está entre un 30% y un 50% del valor del producto.
- El Testing pone en evidencia defectos, pero no agrega calidad ni garantiza que el producto no tiene errores.
- Un desarrollo exitoso puede llevar a un Testing no exitoso.
- El Testing es necesario siempre.
- Se parte de la suposición de que siempre se tendrá defectos por encontrar, ya que es una característica inherente de los productos desarrollados por equipos de personas.
- Puede empezar antes de la codificación porque necesita de los requerimientos para armar los casos de prueba.
- El testing NO asegura que se tenga un producto de calidad (ni la agrega), ni que el proceso por el que se desarrolló sea de calidad.
- Agrupamiento de defectos: los defectos en el software suelen agruparse en un conjunto limitado de módulos o áreas (regla de Pareto, 80% de los defectos se agrupan en el 20% de la funcionalidad). Bajo este principio, las pruebas deben priorizarse y enfocar el esfuerzo en ese conjunto acotado de funcionalidades.

□ Principios del Testing

1. Actividad destructiva: busca deliberadamente fallas.
2. Actitud negativa: el tester intenta demostrar que el software no funciona correctamente.
3. Éxito = encontrar errores.
4. Costo elevado: puede representar hasta la mitad del costo del proyecto.
5. No agrega calidad: solo revela defectos; la calidad la asegura QA.
6. Siempre necesario: ningún software está libre de errores.
7. Anticipación: puede comenzar antes de la codificación (se crean casos de prueba desde los requerimientos).
8. Agrupamiento de defectos: el 80% de los defectos se concentran en el 20% de los módulos (regla de Pareto).
9. No garantiza ausencia de errores: un testing exitoso no implica un producto perfecto.

Principio	Explicación
1. Una parte necesaria de un caso de prueba es definir el resultado esperado .	Si el resultado esperado de un caso de prueba no ha sido predefinido, lo más probable es que un resultado erróneo se interpretará como un resultado correcto , debido a el fenómeno de "el ojo viendo lo que quiere ver", a pesar de la definición destructiva adecuada de prueba, hay todavía un deseo subconsciente de ver el resultado correcto.
2. Un programador debe evitar testear su propio programa.	Los propios desarrolladores "no quieren" encontrar sus propios defectos , por lo que el carácter de pruebas destructivas se deja de lado. Además, puede que el programa contenga errores por malentendidos del programador sobre el dominio, y si él mismo testea, no se van a detectar estos defectos.
3. Una empresa de desarrollo no debe testear sus propios programas.	Misma explicación que el principio 2 orientado a empresas, agregando que si las mismas empresas testean sus programas, es posible que destinen menos recursos y eviten encontrar defectos para cumplir con el calendario y los costos establecidos.
4. Cualquier proceso de prueba debe incluir una inspección minuciosa de los resultados de cada prueba .	Se deben realizar inspecciones minuciosas para detectar la totalidad de los defectos encontrados tras la ejecución de una prueba , ya que pueden surgir más de un defecto (y esto es lo que se busca) por cada caso de prueba.
5. Los casos de prueba deben escribirse para condiciones de entrada que no son válidas e inesperadas , así como para las que son válidas y esperadas .	Muchos errores que se descubren repentinamente en el desarrollo de software aparecen cuando se usa de alguna manera nueva o inesperada, y no responde cómo debería (catcheando el error)
6. Examinar un programa para ver si no hace lo que se supone que debe hacer es sólo la mitad de la batalla; la otra mitad es ver si el programa hace lo que no se supone que debe hacer	Los programas deben ser examinados para detectar defectos secundarios no deseados.
7. Evite los casos de prueba desecharables, a menos que el programa sea realmente un programa de descarte.	Los casos de pruebas utilizados en el testing deben ser reproducibles , es decir, no se deben realizar casos de prueba sobre la marcha (ad-hoc) ya que es imposible reportar un defecto sin tener las condiciones y los pasos en los cuáles el defecto surgió . Además, realizar testing es muy costoso, por lo cuales los casos de prueba deben ser reutilizados para volver a testear escenarios luego de la corrección de errores .
8. No planee un esfuerzo de prueba bajo la suposición tácita de que no se encontrarán errores.	Es un error pensar de esta manera al momento de realizar software. Se debe tener en claro la definición de testing, en la cual se define que el objetivo de esta actividad es encontrar errores, y se presume de antemano su existencia.
9. La probabilidad de que existan más errores en una parte de un programa es proporcional al número de errores ya encontrados en esa parte .	El concepto es útil porque nos da una idea de en qué sección del programa hacer foco o asignar más recursos, si una sección particular de un programa parece ser mucho más propenso a errores que otras secciones, es recomendable realizar pruebas adicionales y es probable que encontremos más errores.
10. Las pruebas son extremadamente creativas e intelectualmente desafiantes .	Aunque existen métodos y estrategias para abarcar un mejor nivel de cobertura testing en los casos de pruebas, siempre es necesario un poco de creatividad del diseñador de estos.

□ **Testing en entornos Ágiles** En metodologías ágiles (como Scrum o XP), el testing:

- Se **integra de manera continua** durante todo el sprint.
- Se enfoca en la **colaboración entre desarrolladores y testers**.
- Se apoya en la **automatización** para asegurar velocidad y repetibilidad.
- Se basa en **criterios de aceptación claros** definidos en las historias de usuario.

Ejemplo: En un sprint de desarrollo, antes de implementar una nueva funcionalidad ("el usuario puede cambiar su contraseña"), el tester diseña casos de prueba automáticos que verifican la validez de los nuevos requisitos. Una vez desarrollada, la integración continua ejecuta automáticamente esas pruebas para validar la historia.

□ **Tipos de Testing frecuentes en entornos Ágiles**

Tipo	Propósito	Ejemplo
Unitario	Validar el comportamiento de un módulo o función	Verificar que la función <code>calcular_descuento()</code> devuelve el

Tipo	Propósito	Ejemplo
		valor correcto
Integración	Verificar que los módulos interactúan correctamente	El módulo de pagos se comunica correctamente con el de stock
Funcional / de Aceptación (UAT)	Validar que la historia de usuario cumple los criterios de aceptación	"Como usuario, quiero recibir un email al registrarme"
Regresión	Asegurar que una nueva versión no rompe funcionalidades anteriores	Después de cambiar el login, las compras siguen funcionando
Exploratorio	Testing manual creativo sin casos predefinidos	El tester explora libremente la app buscando errores no previstos

□ Testing continuo

En Agile, el testing está integrado en la **Integración Continua (CI)** y la **Entrega Continua (CD)**.

Cada cambio de código se prueba automáticamente antes de ser integrado.

Ejemplo: Cada vez que un desarrollador hace un commit, el sistema ejecuta:

- Tests unitarios → validan el código.
- Tests de integración → prueban interacciones.
- Tests de regresión → aseguran que no se rompa nada previo.

□ Conclusión El testing en ambientes ágiles es:

- Una **actividad colaborativa y continua**, no una etapa final.
- Enfocada en la **prevención temprana de errores**.
- Soportada por **automatización y retroalimentación constante**.
- Fundamental para garantizar entregas **frecuentes, funcionales y confiables**.

Los **10 principios clásicos del testing de software** (según Glenford Myers, "The Art of Software Testing"), **versión mejorada** de la tabla con lenguaje unificado, ejemplos aplicados y una columna extra que muestra cómo se aplica **en entornos ágiles** (

□ Principios Fundamentales del Testing de Software

Nº	Principio	Explicación	Ejemplo / Enfoque Ágil
1	Definir el resultado esperado	Si no se establece el resultado esperado antes de ejecutar un caso de prueba, un resultado incorrecto podría interpretarse como correcto ("el ojo ve lo que quiere ver").	En Scrum, los criterios de aceptación de una historia de usuario cumplen esta función: definen claramente qué debe pasar para considerar la historia como "hecha".
2	Un programador no debe testear su propio código	Los desarrolladores tienden a ser menos críticos con su propio trabajo, y pueden no detectar defectos derivados de malentendidos del dominio.	En Agile se fomenta el pair testing (tester + dev) o la revisión cruzada de código mediante pull requests o code reviews .
3	La empresa que desarrolla no debe testear su propio producto	Si una empresa testea su propio software, puede existir un sesgo para cumplir plazos o minimizar costos, evitando encontrar errores.	En proyectos ágiles grandes, suele haber un equipo QA independiente , o bien QA se integra en el equipo Scrum pero con autonomía de decisión.
4	Inspección minuciosa de los resultados	Cada caso de prueba debe analizarse cuidadosamente, ya que un solo test puede revelar múltiples defectos.	Se utilizan herramientas de gestión de pruebas (como TestLink, Jira Xray o Zephyr) para registrar y revisar los resultados detalladamente.
5	Probar entradas válidas e inválidas	No solo deben probarse los casos esperados, sino también los escenarios erróneos o extremos.	En TDD (Test Driven Development), se crean pruebas tanto para el " camino feliz " como para los casos límite y errores esperados .
6	Verificar lo que no debe hacer el programa	No basta con comprobar que hace lo que debe, sino también que no hace lo que no debe hacer .	Ejemplo: validar que un usuario sin permisos no pueda acceder a funciones restringidas.
7	Evitar casos de prueba desecharables	Los casos de prueba deben ser reutilizables y reproducibles , para poder verificar correcciones o regresiones.	En Agile, los casos de prueba automatizados se integran en pipelines de CI/CD (Jenkins, GitHub Actions, etc.) y se ejecutan continuamente.
8	Asumir que se encontrarán errores	El testing parte de la premisa de que el software contiene defectos . Si no se buscan activamente, se ocultarán.	Las retrospectivas de sprint permiten ajustar estrategias de testing y asumir que siempre habrá errores por mejorar .
9	Las áreas con más errores tienden a tener más errores	Si una parte del sistema ya mostró defectos, es probable que tenga más. Se debe enfocar más esfuerzo ahí.	Se priorizan pruebas de regresión y automatización en módulos con historial de fallos o alta criticidad.
10	El testing es una tarea creativa e intelectual	Aunque existen métodos sistemáticos, diseñar pruebas efectivas requiere imaginación y pensamiento crítico .	Los testers ágiles combinan casos automatizados con testing exploratorio , usando creatividad para descubrir escenarios no previstos.

□ Conclusión general El testing:

- No es solo una tarea técnica, sino también **cognitiva y estratégica**.
- Requiere equilibrio entre **método y creatividad**.
- En contextos ágiles, estos principios se **mantienen vigentes**, pero se adaptan al trabajo colaborativo, la automatización y la entrega continua.

Mitos del Testing

- “El testing es el proceso para demostrar que los errores no están presentes”.
- “El propósito del testing es demostrar que un programa realiza sus funciones previstas de forma correcta”
- “El testing es el proceso que demuestra que un programa hace lo que se supone que debe hacer”

Estas definiciones o afirmaciones sobre el testing son incorrectas, ya que el **objetivo de testear un programa es agregar valor al producto revelando su calidad y brindando confianza** en el software, de forma más concreta, encontrar y remover defectos en el código de este. Entonces, no se prueba un sistema para mostrar que funciona, sino que se comienza con la suposición de que el software contiene defectos, y se realiza el testing para encontrar la mayor cantidad de ellos posible.

Por otro lado, **un programa puede hacer lo que se supone que debe hacer, y aun así, contener defectos**. Es decir, **un error está claramente presente si un programa no hace lo que se supone que debe hacer, pero los errores también están presentes si un programa hace lo que no se supone que debe hacer**.

<input type="checkbox"/> Mito	<input type="checkbox"/> Realidad
“El testing demuestra que el software no tiene errores.”	El testing no demuestra ausencia de errores , sino que revela su presencia . Su objetivo es aumentar la confianza en el producto encontrando defectos.
“El testing demuestra que el programa hace lo que debe hacer.”	Un programa puede cumplir con sus funciones y aún contener errores ocultos o efectos secundarios no deseados.
“El propósito del testing es confirmar que el sistema funciona.”	Se prueba para descubrir fallas , no para confirmar que todo está bien. Testing = actividad destructiva y preventiva , no validatoria.

Ejemplo práctico: En un e-commerce, el sistema puede permitir agregar productos al carrito (funciona bien), pero si no actualiza el stock correctamente, **hay un defecto funcional oculto**.

¿Cuánto Testing es suficiente?

El **testing exhaustivo es imposible** por la cantidad de tiempo que requiere. El **momento** en que se **deja de hacer testing** depende del nivel de riesgo o costo asociado al proyecto. Los riesgos permiten definir prioridades de que se debe testear primero y con qué esfuerzo.

El **criterio de aceptación** se utiliza normalmente para **decidir** si una determinada **fase de testing ha sido completada**. Este puede ser definido en términos de:

- Costos.
- % de tests corridos sin fallas.
- Inexistencia de defectos de una determinada severidad.
- Pasa exitosamente el conjunto de pruebas diseñado y la cobertura estructural.
- Good Enough: Cierta cantidad de fallas no críticas es aceptable.
- Defectos detectados es similar a la cantidad de defectos estimados.

El criterio de aceptación sirve para definir y negociar en ágil con el Product Owner y en tradicional con el Líder del Proyecto a cuántos defectos son aceptables para terminar.

¿Cuánto Testing es suficiente?

- **El testing exhaustivo es imposible:** el tiempo y costo lo hacen inviable.
- Se debe definir cuándo detener las pruebas según **riesgos, costos y criterios de aceptación**.

Criterios de aceptación comunes:

- Límite de costos o tiempo de prueba alcanzado.
- Porcentaje de pruebas ejecutadas sin fallos (por ejemplo, 95%).
- No existen defectos de severidad “crítica” o “bloqueante”.
- Cobertura estructural o funcional cumplida.
- “Good Enough”: se acepta cierta cantidad de defectos menores.
- Defectos detectados ≈ defectos estimados.

En Ágil: estos criterios se negocian entre el **Product Owner y QA** durante la definición de “Definition of Done”.

Conceptos importantes

Defecto versus Error

La **diferencia** entre ambos conceptos es el **momento** en el cual se detectan y solucionan los errores o defectos. Un **error** es detectado y corregido en una misma etapa, y un **defecto** es un error que se traslada de una etapa a otra etapa posterior en la cual se introdujo. El testing encuentra defectos, ya que son errores que surgieron en etapas anteriores, pero se detectan en la etapa de prueba.

Hay que aclarar que **ambos conceptos pueden o no generar fallas en el sistema**, es decir, un **mal funcionamiento de este**. Por ejemplo, un error en los colores de una interfaz no es una falla, ya que no conllevan a un mal funcionamiento del sistema.

Concepto	Momento de detección	Descripción
Error	En la misma etapa en que se introduce	Una acción incorrecta del desarrollador o analista (por ejemplo, una mala interpretación de un requerimiento).
Defecto	En una etapa posterior a la introducción del error	Es el resultado visible de un error previo, detectado durante pruebas o uso.

Un error en el color de un botón no es una **falla**, pero sí un **defecto cosmético**.

Defecto

Un defecto posee **dos características principales**, que permiten catalogarlos en la etapa de testing:

- **Severidad**: define la **gravedad del defecto**, y es determinada por la persona que realiza el testing, por lo que esta característica es de **carácter técnico**. El valor de la severidad se asigna dependiendo de la siguiente escala:
 - Bloqueante → el defecto no permite continuar con la ejecución del sistema.
 - Crítico → el sistema funciona, pero la funcionalidad que se está testeando tiene un defecto crítico.
 - Mayor → la funcionalidad que se está testeando **funciona, pero no de forma correcta**.
 - Menor → la funcionalidad se **ejecuta** correctamente, pero con **advertencias erróneas o errores de baja importancia**.
 - Cosmética → formato de fechas, formato de números, distribución de componentes en una GUI, temas de presentación de interfaces, etc.
- **Prioridad**: la prioridad define el **impacto del defecto en la funcionalidad** para el negocio, y permite ordenar la atención de los defectos según las necesidades del cliente. Una escala posible para la prioridad puede ser:
 - Urgencia
 - Alta
 - Media
 - Baja

Estas características son **independientes entre sí**, ya que varían dependiendo del tipo de negocio y de sistema que se está desarrollando. Por ejemplo, un defecto cosmético puede tener baja prioridad para un sistema de inventarios, pero puede tener una alta prioridad si se trata de un sistema de una empresa de marketing, en donde el aspecto y la presentación es algo muy importante.

Defectos – Clasificación

Por Severidad (impacto técnico)

Nivel	Descripción
Bloqueante	Impide continuar con la ejecución del sistema.
Crítico	Afecta una funcionalidad esencial.
Mayor	Funciona parcialmente o con comportamiento incorrecto.
Menor	Funciona, pero muestra advertencias o mensajes erróneos.
Cosmético	Errores visuales o de formato, sin impacto funcional.

Por Prioridad (impacto en el negocio)

Nivel	Descripción
Urgente	Debe resolverse inmediatamente.
Alta	Impacta el uso principal del sistema.
Media	Afecta parcialmente la funcionalidad.
Baja	No afecta operaciones críticas.

Ejemplo: Un error de color en un logo → Cosmético, baja prioridad. Una transacción bancaria que no descuenta saldo → Crítico, alta prioridad.

Casos de prueba

Son una secuencia de **pasos** que hay que seguir para **obtener un resultado esperado**, y se tienen que dar ciertas condiciones previas que fijan **una situación** para que se pueda ejecutar la prueba.

Es el **artefacto más importante** del Testing. Este se **hace una vez**, pero sirve para realizar **infinitas pruebas**, debiendo obtener en todas el mismo resultado esperado.

Se trata de minimizar la cantidad de casos de prueba maximizando la cantidad de defectos encontrados. El objetivo de los casos de prueba es descubrir defectos.

Los casos de prueba salen de los **requerimientos**, permitiendo hacer tanto la verificación como la validación.

Está compuesto por: un **objetivo** (lo que se desea controlar), **condiciones de prueba** (Datos de entrada y de entorno que deben estar presente para llevarse a cabo la prueba) y un **resultado esperado**.

- Son secuencias de pasos con datos de entrada, condiciones previas y un resultado esperado.
- Su objetivo es descubrir defectos, no probar que todo está bien.
- Derivan de los requerimientos, permitiendo verificar y validar.
- Deben ser reutilizables y trazables.

□ **Estructura básica de un caso de prueba:**

Campo	Descripción
ID del caso	Identificador único.
Objetivo	Qué se desea comprobar.
Condiciones previas	Estado o datos necesarios.
Pasos	Secuencia de acciones a realizar.
Resultado esperado	Qué debería ocurrir.
Resultado obtenido	Qué ocurrió realmente.

□ **Ejemplo:**

- **Objetivo:** Validar login con credenciales inválidas.
- **Condición previa:** Usuario registrado.
- **Resultado esperado:** Mostrar mensaje de error.

Ciclos de prueba

Es la ejecución de un conjunto de casos de prueba en una versión determinada del producto. Generalmente se tienen 2 ciclos, y el primero es conocido como ciclo 0. El ciclo 0 siempre es manual, es donde se configura todo y a partir del ciclo 1 ya se pueden automatizar las pruebas.

En caso de detectarse defectos, el producto vuelve a desarrollo para su corrección.

Si existe una cantidad alta de ciclos de prueba, es probable que QA no sea bueno, por lo que deberían revisarse ciertos aspectos acerca de la calidad del proceso y producto, de manera de evitar tener grandes cantidades de ciclos, que se traducen en mucho retrabajo, lo cual a su vez conduce a incrementar los costos de desarrollo.

- Cada ciclo ejecuta un conjunto de casos sobre una **versión del producto**.
- **Ciclo 0:** manual y base; se configuran los entornos y se validan los primeros casos.
- **Ciclo 1+ :** pueden incluir **automatización** y validación de correcciones.

□ **Indicador de calidad:** Muchos ciclos → posible problema de calidad en el proceso QA.

Ciclo de pruebas con regresión

Este concepto plantea la necesidad de ejecutar exhaustivamente todos los casos de prueba en cada ciclo de prueba, como si fuesen el ciclo 0. Este planteo es el ideal, pero el menos utilizado, por cuestiones de practicidad y tiempo.

El otro enfoque, consiste en aplicar una prueba exhaustiva de todos los casos de prueba en el ciclo 0, pero a partir del ciclo 1 (si el incremento o producto vuelve nuevamente a testing) ejecutar sólo los casos de prueba asociados a defectos encontrados previamente, y no los casos de prueba que hayan pasado sin encontrar defectos. Esto permite agilizar el proceso de pruebas, pero estadísticamente es muy probable que al arreglar un defecto reportado por testing, en el desarrollo de su corrección se introduzcan nuevos errores en otras partes del producto. Por lo que, si no se prueban todos los casos, a pesar de que previamente no se hayan detectado defectos, pueden encontrarse nuevos, debido al proceso descrito anteriormente.

Una solución a esta encrucijada es no aplicar regresión (ejecutar sólo los casos de prueba con defectos asociados), y cuando finalmente estén todos los defectos “corregidos”, hacer una ejecución de todos los casos de prueba (como si se aplicara regresión), para aumentar la confianza de que no se han introducido nuevos errores en las correcciones.

La automatización del testing permite hacer tantos ciclos de prueba con regresión como se deseen. La desventaja es el proceso de automatizar la ejecución de los casos de prueba, pero se puede ver ese esfuerzo de automatización como el esfuerzo propio del ciclo 0 manual, y luego esa automatización permite ejecutar los casos de prueba N veces. En empresas donde el core de negocio no es hacer software, conviene asumir ese esfuerzo y automatizar las pruebas.

Claramente, la mejor estrategia es aplicar regresión, porque no existe un punto medio al ser un método de caja negra, ya que no se puede saber con certeza lo que va a afectar al corregir un defecto y en dónde impactará esa corrección. Sin embargo, en la práctica se utiliza sin regresión, debido a los costos y tiempos que implica ese proceso.

Testing con Regresión

Enfoques:

1. **Regresión total (ideal):** repetir todos los casos de prueba cada ciclo.
 - Mayor confianza – Alto costo.
2. **Regresión parcial:** ejecutar solo los casos que antes fallaron.
 - Más ágil – Riesgo de nuevos errores ocultos.

Solución práctica: Aplicar regresión completa al final de las correcciones o usar **automatización** (Selenium, Cypress, Jenkins CI/CD).

Testing Automatizado

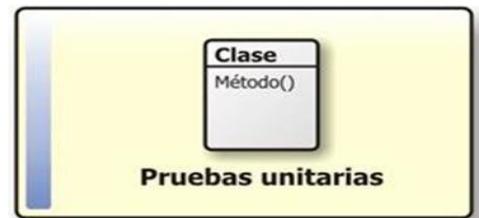
- Implica programar los casos de prueba para ejecutarlos repetidamente.
- Requiere **alto esfuerzo inicial**, pero reduce tiempos en los siguientes ciclos.
- Ideal para regresión, smoke tests y validaciones repetitivas.
- Muy usado en entornos **DevOps y Ágiles** (integración continua).

Conclusión general

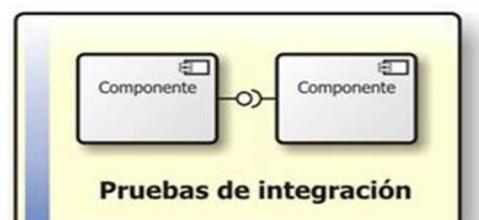
Aspecto	Enfoque tradicional	Enfoque ágil
Propósito del testing	Verificar el producto terminado	Integrar testing durante todo el desarrollo
Responsable	Equipo QA dedicado	Todo el equipo (QA, Devs, PO)
Documentación	Alta formalidad	Casos de prueba ligeros, automatizados
Frecuencia	Al final del ciclo	En cada sprint / integración continua

Los niveles de prueba determinan el foco o la **granularidad de la prueba** que se está por ejecutar. En general, primero se **comienzan** probando **componentes pequeños**, y luego se **integran** en pruebas de mayor **granularidad**. Esto es al revés de cómo se desarrollan los componentes.

- **Pruebas unitarias:** las hace el **desarrollador** y están **incluidas en el DoD**. Son pruebas que se realizan sobre un componente, de manera independiente a los otros. Se hacen teniendo **acceso al código fuente**, y se pueden usar herramientas para realizarlas (automatización, depuración). Se suelen **reparar los errores** apenas **se encuentran**, sin registrarlos formalmente.



- **Pruebas de integración:** El propósito de la ejecución de estas pruebas es **verificar el funcionamiento de dos o más componentes juntos que se relacionen entre ellos**, es decir, clases en las cuales existe una interacción a modo de peticiones, a través de sus interfaces (boundaries). El **resultado de estas pruebas es el build** (el CI o continuous integration), el cual luego **se envía al equipo de testing** y es lo que se prueba en siguientes etapas.



Se suele llevar a cabo una prueba de integración incremental, desde lo más general (que abarca más componentes) a lo más específico (top-down) o desde lo más específico a lo más general (bottom-up).

- **Pruebas de sistema o de versión:** En estas pruebas se realizan **testeos sobre la versión de un incremento de producto** o de un producto (dependiendo si se usa Agile o métodos tradicionales), y existen razones psicológicas que demuestran que deben realizarlas **personas ajena**s al desarrollo de los programas (obligatoriamente).



Estas pruebas se llevan adelante siguiendo un **proceso sistemático** y metodológico, que permite encontrar defectos que puedan ser reproducibles y reportar de qué manera ocurre el defecto detectado, es decir, cual es el camino de pasos que hay que seguir para encontrar el error.

Estas pruebas están **basadas en casos de prueba**. Se trata de emular de la mejor manera posible un entorno de trabajo idéntico al entorno real en el que se usara el SW. Se llevan a cabo pruebas del funcionamiento real y cotidiano del producto. Abarca requerimientos funcionales y no funcionales.

- **Pruebas de aceptación de usuario:** se **realizan en el despliegue** y **debería realizarlas el usuario** para **verificar que se cumple con lo que el mismo requirió**. Busca que el cliente/usuario se familiarice con el producto, generando comodidad y confianza sobre el mismo (**su objetivo principal no es encontrar fallas**).



También **busca reproducir un entorno de producción**. **Comprende tanto la prueba realizada por el usuario en**

ambiente de laboratorio (pruebas alfa), como la prueba en ambientes de trabajo reales (pruebas beta). En la teoría, los usuarios arman sus pruebas de usuario. En la práctica, las arma Testing.

En la teoría, además del usuario, deben estar presentes el gerente de testing, el líder del proyecto y empleados con roles funcionales.

En Agile, además del usuario, deben estar todos los miembros del equipo (Sprint Review).

□ 1. Pruebas unitarias

- **Responsable:** Desarrollador.
- **Objetivo:** Validar el funcionamiento de componentes individuales (funciones, clases o módulos) de manera aislada.
- **Características:**
 - Acceso al código fuente.
 - Se automatizan frecuentemente.
 - Los errores se corrigen en el momento (sin registro formal).
- **Ejemplo:** Probar que la función `calcular_total(precio, cantidad)` devuelva el valor correcto para diferentes combinaciones de entrada.

□ 2. Pruebas de integración

- **Responsable:** Desarrolladores o equipo técnico.
- **Objetivo:** Verificar que los componentes interactúen correctamente entre sí.
- **Técnicas:**
 - Top-Down: de lo general a lo específico.
 - Bottom-Up: de lo específico a lo general.
- **Resultado:** Build o integración continua (CI).
- **Ejemplo:** Probar que el módulo de "carrito de compras" comunique correctamente el total al módulo de "pagos".

□ 3. Pruebas de sistema (o versión)

- **Responsable:** Equipo de Testing (independiente del desarrollo).
- **Objetivo:** Validar el funcionamiento global del producto en un entorno similar al real.
- **Incluye:** Pruebas funcionales y no funcionales (rendimiento, seguridad, usabilidad).
- **Ejemplo:** Verf que el sist permita registrar un nuevo usuario, iniciar sesión y realizar una compra completa sin errores.

□ 4. Pruebas de aceptación de usuario

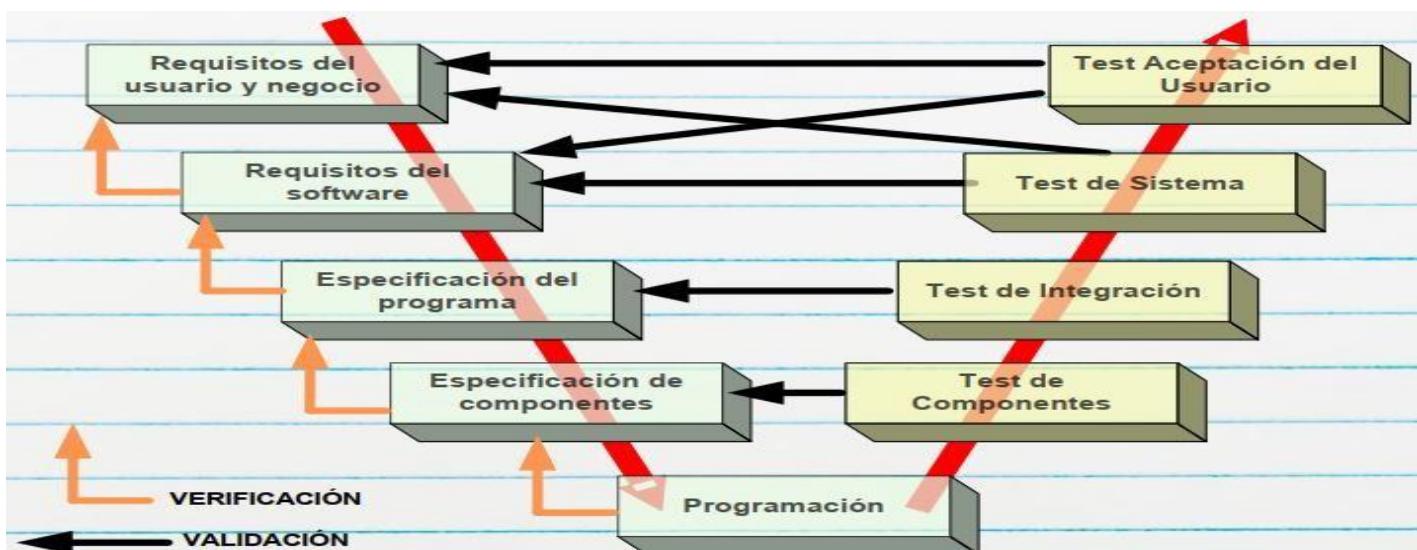
- **Responsable:** Cliente o usuario final.
- **Objetivo:** Validar que el sistema cumple con lo que se solicitó y genera confianza.
- **Tipos:**
 - Alfa: en laboratorio.
 - Beta: en entorno real.
- **Ejemplo:** Un cliente prueba el software de facturación en su negocio real antes del lanzamiento final.

Modelo en V

Este modelo se utiliza para poder determinar cuándo se está en condiciones de realizar determinadas actividades, en función de la etapa de desarrollo en la que se encuentre el software.

Acá se puede notar cómo las pruebas se realizan en orden de granularidad inverso al proceso de desarrollo del software. Es decir, se desarrolla desde componentes de granularidad gruesa, como son los requerimientos abstraídos de detalles de implementación, hacia componentes de menor granularidad, como son clases o módulos, dependiendo del paradigma. En cambio, para las pruebas, la granularidad se da en sentido inverso.

- Testing de Componentes → Testing unitario.
- Especificación del programa → diseño/arquitectura.



El modelo en V muestra la relación entre las fases de desarrollo y las de prueba:

Etapa de Desarrollo	Etapa de Prueba Asociada
Requerimientos	Pruebas de aceptación
Diseño del sistema	Pruebas de sistema
Diseño detallado	Pruebas de integración
Codificación	Pruebas unitarias

□ Mientras el desarrollo va de lo general a lo específico, las pruebas van en sentido inverso.

Ambientes de Testing

Son todos los recursos, tanto hardware como software, que se requieren para poder trabajar con el producto.

Existen distintos ambientes en el desarrollo de software, los cuales poseen distintas necesidades, según la etapa de desarrollo en la que se encuentre el software.

Desarrollo

Implica hardware y software necesarios (librerías, extensiones, IDEs, compiladores, etc.) para poder desarrollar y desplegar el producto y utilizarlo. En este ambiente se realizan las pruebas unitarias (y también las de integración generalmente).

Pruebas

Es el ambiente que utilizan los testers para llevar a cabo las pruebas, y los desarrolladores no deben tener acceso. En este se realizan las pruebas del sistema. Normalmente acá se realizan las pruebas de sistema.

Preproducción

Debería tener las mismas características que el ambiente productivo para poder comprobar que el producto funcionará una vez desplegado en producción de manera correcta. En este ambiente se realizan las pruebas de aceptación.

El problema de este entorno es que muchas veces resulta difícil (o imposible en algunos casos), debido a que es muy costoso replicar principalmente las configuraciones de hardware. Por ejemplo, sería prácticamente imposible replicar la configuración de servidores del motor de búsqueda de Google, para realizar pruebas de aceptación.

Producción

Este entorno, es la configuración de software y hardware que tienen los usuarios finales del software, y que utilizan para el desempeño de sus actividades laborales. Obviamente en este entorno no se realizan pruebas (probar en este ambiente tiene grandes consecuencias), ya que en teoría la versión del producto cumple con los criterios del Definition of Done.

Ambiente	Descripción	Tipo de Pruebas
Desarrollo	Utilizado por los programadores para construir y probar componentes.	Unitarias e integración.
Testing	Usado por testers, aislado del desarrollo.	Pruebas de sistema.
Preproducción	Replica del entorno real, para validar antes del lanzamiento.	Aceptación.
Producción	Entorno real del usuario final.	No se prueban nuevas versiones aquí.

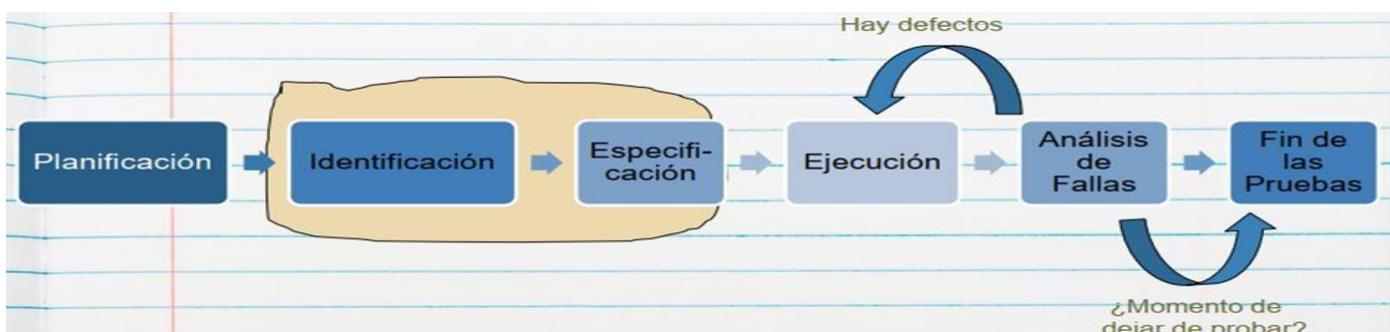
□ Ejemplo: antes de liberar una app móvil, se prueba en preproducción con dispositivos y red similares a los reales.

Proceso de pruebas

El Testing es un proceso definido (que se lleva a cabo durante todo el ciclo de vida del producto), por lo que está compuesto de etapas detalladas.

Testing Ad Hoc: se realiza testing sin ningún proceso definido, perdiendo la trazabilidad ya que no se registra el paso a paso de lo que se probó, sino que se prueba libremente sin llevar ningún registro.

Generalmente el proceso de pruebas es estándar (puede tener algunas variaciones, pero sigue una estructura general).



- **Planificación:** Determina como se incluirá el Testing en el plan del proyecto, y como será el Test Plan (que recursos se usará, que riesgos se tendrá, cuál será el criterio de aceptación, que entornos se emularán, quien realizará cada Testing, cuando, etc.). El resultado de la planificación es el **Plan de Pruebas**, que debe contener:
 - Riesgos y objetivos del Testing.
 - Estrategia de Testing.
 - Recursos.
 - Criterio de Aceptación.
- **Diseño (Identificación y especificación de casos de prueba):** Revisando las bases de la planificación del Testing, se identifican los datos necesarios, diseñan y priorizan los CP que se llevaran a cabo (se define que entorno se usara, como se llevaran a cabo, por quien, cuando, etc.). Analiza si los requerimientos son testeables o no. Se define si se usa regresión o no.
- **Ejecución:** Cuando se ejecutan los CP, se registran y se comparan los resultados generando un reporte de defectos (con defectos encontrados, condiciones, entornos). Se trata de automatizar lo más que se pueda respecto de estas ejecuciones (ahormando tiempo/costo). Incluye: Creación de los datos necesarios para la prueba, automatización de todo lo que sea necesario, implementar y verificar el ambiente, ejecutar los casos de prueba, registrar los resultados de la ejecución y comparar los resultados reales con los esperados.
- **Análisis de fallas (Evaluación y Reporte):** Se hace un seguimiento de la corrección de los defectos encontrados hasta que se cierren todos los CP, es decir que se hayan solucionado todos. Se evalúa el criterio de aceptación, se reporta el resultado de las pruebas a los interesados, se verifica los entregables y que los defectos se hayan corregido y se evalúa cómo resultaron las actividades de testing y se analizan las lecciones aprendidas.
Acá se confecciona el informe de reportes.
- **Fin de las pruebas:** En las empresas más maduras se deja de probar recién cuando no hay defectos bloqueantes, críticos, mayores y los defectos menores y cosméticos son muy pocos, los que se ponen en la nota de Release. Se confecciona el informe final (opcional).

Proceso de pruebas (Testing process)

1. **Planificación**
 - Define el Test Plan: qué, quién, cómo, cuándo, y con qué recursos.
 - Incluye riesgos, estrategia, y criterios de aceptación.
 - **Ejemplo:** definir que se testeará el login, pagos y generación de reportes en las próximas 2 semanas.
2. **Diseño**
 - Se crean los **casos de prueba** y se priorizan.
 - Se determinan los datos de entrada, entorno y condiciones.
 - **Ejemplo:** caso de prueba “Login inválido”: ingresar usuario correcto con contraseña incorrecta y verificar el mensaje de error.
3. **Ejecución**
 - Se ejecutan los casos, se registran resultados y defectos.
 - Se pueden automatizar para ahorrar tiempo.
 - **Ejemplo:** correr 50 casos de prueba de compra online y registrar cuáles fallan.
4. **Análisis de fallas**
 - Seguimiento y validación de los defectos encontrados.
 - Se evalúa el cumplimiento de los criterios de aceptación.
 - **Ejemplo:** de los 15 defectos críticos reportados, 13 fueron corregidos y 2 reabiertos.
5. **Fin de pruebas**
 - Se cierra la fase cuando no quedan defectos críticos o bloqueantes.
 - Se genera el informe final de calidad.
 - **Ejemplo:** el producto se libera con solo 2 defectos cosméticos documentados en el Release Note.

Artefactos de Testing

Plan de pruebas

Este plan, es análogo al plan de proyecto que se elabora en el enfoque tradicional (no forma parte de ese plan). En este plan, se definen:

- Datos necesarios para las pruebas.
- Recursos destinados a las pruebas.
- Herramientas a utilizar.
- Si se aplicará regresión (o no).
- Etc.

Para la confección de este plan no es necesario el código, sino que sólo se necesitan los requerimientos, en el formato que sea que se encuentren (ERS o Casos de Uso en métodos tradicionales, User Stories en Agile, etc.)

Casos de prueba

- Es el artefacto más importante del testing, y consiste en una secuencia de pasos a seguir, las cuales describen un conjunto de acciones a realizar para lograr un resultado concreto y esperado. Para esto se deben explicitar las condiciones de inicio (las cuales fijan una situación particular), y los datos utilizados en ese escenario.
- Se confeccionan con el objetivo de descubrir la mayor cantidad de defectos y minimizar el esfuerzo y tiempo para elaborarlos y ejecutarlos.
- Mientras no cambien los requerimientos, estos casos pueden ser utilizados las veces que sea necesario.
- La característica más importante de los casos de prueba es ser REPRODUCIBLES. Si se encuentra un defecto y no es reproducible a través de un caso de prueba, no es un defecto.

Los casos de prueba se derivan de diferentes fuentes:

- Documentos del cliente;
- Información relevada;
- Requerimientos;
- Especificaciones de programación;
- Código.

Reporte de incidentes o defectos

Luego de la ejecución de los casos de prueba, se elabora un reporte, indicando cuáles fueron los casos de prueba que han pasado, y aquellos en los que se ha encontrado defectos, indicando cuáles fueron esos defectos encontrados y cómo reproducirlos para su detección y corrección.

Este artefacto permite a los desarrolladores corregir esos defectos, para enviarlos nuevamente a testing.

Informe final

suele contener métricas e información final del incremento del producto, se reporta cuántos ciclos y la cantidad de errores por ciclo, métodos, tipos de prueba utilizados, etc. Este es el único artefacto que es negociable en algunas organizaciones informales, se acuerda en la contratación del trabajo.

Tiene utilidades estadísticas.

Plan de pruebas

Es el documento que define cómo se realizarán las pruebas de un proyecto.

No requiere el código, sino los requerimientos funcionales y no funcionales (ERS, Casos de Uso o User Stories).

Debe incluir:

- Datos necesarios para las pruebas.
- Recursos humanos y técnicos.
- Herramientas de testing y automatización.
- Estrategia de regresión (si se aplicará o no).
- Riesgos, criterios de aceptación, y calendario.

Ejemplo:

En un sistema bancario, el plan de pruebas podría indicar que se usarán 3 testers, Selenium para pruebas automáticas, y que se validará regresión en cada sprint.

Casos de prueba

El artefacto más importante del testing.

Define una secuencia de pasos reproducibles para verificar una funcionalidad del sistema.

Elementos de un caso de prueba:

- **Nombre:** descripción clara del escenario.
Ejemplo: "Ingreso al sistema con contraseña incorrecta"
- **Precondiciones:** estado necesario antes de la prueba.
Ejemplo: "El usuario Juan está registrado en la base de datos."
- **Pasos:** acciones a seguir.
Ejemplo: 1. Juan ingresa al sitio. 2. Ingresa su usuario y una contraseña incorrecta. 3. Hace clic en Ingresar.
- **Datos de entrada:** valores específicos que se usarán.
- **Resultado esperado:** lo que el sistema debe mostrar o ejecutar.
Ejemplo: El sistema muestra el mensaje "Contraseña incorrecta".

Características clave:

- Reproducibles (deben poder repetirse).
- Derivados de fuentes formales: requerimientos, código, especificaciones, documentación del cliente.
- Pueden reutilizarse mientras no cambien los requerimientos.

Reporte de incidentes o defectos

Documento que se genera después de ejecutar los casos de prueba, registrando los defectos encontrados.

Debe incluir:

- Casos de prueba fallidos.
- Descripción del defecto.
- Pasos para reproducirlo.
- Prioridad/severidad.
- Estado del defecto (nuevo, en revisión, corregido, cerrado).

Ejemplo:

Defecto #1024 – “El sistema se bloquea al eliminar un cliente sin dirección.”

- Reproducible 100%.
- Severidad: alta.
- Estado: corregido y reabierto.

Informe final de pruebas

Documento que resume el resultado general del proceso de testing.

Incluye métricas, estadísticas y evaluación de calidad.

Contenido habitual:

- Número de ciclos ejecutados.
- Cantidad de errores detectados/corregidos por ciclo.
- Tipos de prueba aplicadas.
- Cobertura alcanzada.
- Criterios de aceptación cumplidos.
- Lecciones aprendidas.

Ejemplo:

En el Sprint 5 se ejecutaron 45 casos de prueba, se detectaron 12 defectos (3 críticos), y se corrigieron todos antes del cierre.

El Testing y el Ciclo de Vida

Si desarrollamos con ciclo de vida en cascada (Ciclo de Vida Secuencial) las pruebas se hacen al final ya que es el momento en el que nos entregan el producto. En cambio, si desarrollamos con ágil (Ciclo de Vida Iterativo/Incremental), hacemos testing por iteración. El problema es que se pueden incluir errores por la integración de los incrementos.

Enfoque	Cuándo se realiza el Testing	Riesgos principales
Cascada (secuencial)	Al final del desarrollo, cuando se entrega el producto completo.	Se detectan errores tarde, lo que encarece su corrección.
Ágil (iterativo/incremental)	En cada iteración o sprint.	Possible aparición de errores al integrar incrementos sucesivos.

En Agile, el testing es continuo y acompaña cada incremento, mientras que en Cascada es una fase posterior.

Estrategias de prueba

Se utilizan para pasar por la mayor cantidad de funcionalidades con la menor cantidad de casos pruebas confeccionados, debido a que el tiempo y el presupuesto para diseñarlos y ejecutarlos es limitado.

Hay dos grandes enfoques: Caja Negra y Caja Blanca. Cada uno tiene fortalezas y debilidades particulares: un método puede ser bueno para algunas cosas, y no para otras cosas. El mejor método es no usar un único método, sino usar una variedad de técnicas ayudará a un testing efectivo.

Las estrategias permiten cubrir la mayor cantidad de funcionalidades con el menor número de casos de prueba posibles, optimizando tiempo y recursos. Existen dos grandes enfoques:

Caja Negra

En esta estrategia no se dispone de la estructura interna de la implementación, sino que se analizan las funcionalidades como una caja negra, en términos de entradas y salidas de esa “caja”.

El proceso consiste en ingresar determinados datos a esa funcionalidad vista como caja negra, y luego comparar los resultados obtenidos con los resultados esperados. Para ello, se realiza un testing exhaustivo de entrada, probando cada posible entrada conducida como caso de prueba, no necesariamente las válidas. En transacciones no sólo se debe considerar todos los datos posibles, sino todas las secuencias posibles de transacciones previas.

Se clasifican en basados en especificaciones y basados en experiencia:

- Métodos basados en especificaciones: Son aquellos que se ejecutan utilizando la documentación de especificaciones realizadas del producto.
 - Partición de equivalencias: proceso sistemático que consiste en identificar clases de equivalencia, que definen subconjuntos de datos que producen un resultado equivalente.
 - Análisis de los valores límites: variante del anterior. Utilizar los límites o valores de borde de las clases de equivalencia para la definición de los casos de prueba.
- Métodos basados en experiencia: la experiencia y los conocimientos del tester son

fundamentales para determinar las entradas del sistema y analizar los resultados.

- Adivinanza de defectos: enfoque basado en la intuición y experiencia para identificar pruebas que probablemente expongan defectos del software, elaborando una lista de defectos posibles o situaciones propensas a error y realizando pruebas a partir de esa lista.
- Testing exploratorio: el tester mientras va probando el software, va aprendiendo a manejar el sistema y junto con su experiencia y creatividad, genera nuevas pruebas a ejecutar.

Partición de equivalencias

Analiza las condiciones externas (entradas y salidas) involucradas en una funcionalidad determinada. A esas condiciones externas las divide en clases de equivalencia, las cuales son subconjuntos de valores posibles, que arrojan resultados equivalentes en la funcionalidad que se quiere probar.

Ejemplos de entradas: campos de texto, fechas, coordenadas de un mapa, archivos, señales de sensores, etc. Ejemplos de salidas: mensaje en pantalla, advertencias, listados, tablas, emisión de señal, etc.

Procedimiento

1. Identificar condiciones externas de entrada y salida.
 - a. Si hay que ingresar dos datos que son iguales, pero con distintos valores, separarlo en condiciones externas diferentes. Por ejemplo: 2 calles, una condición externa es calle1 y la otra calle2.
2. Identificar subconjuntos de valores posibles (válidos y no válidos) de las condiciones externas identificadas, que produzcan resultados equivalentes en la ejecución de la funcionalidad.
 - a. Los subconjuntos no válidos tienen asociados mensajes de error en la condición externa de salida "Mensajes de Error" o "Mensajes de Advertencia" (no es necesario poner todos los mensajes, con algunos genéricos es suficiente).
 - b. Para no olvidarse de algún subconjunto → la unión de todos los subconjuntos debe ser igual al universo de la condición externa (teoría de conjuntos).
3. Armar los casos de prueba tomando de cada condición externa, un sólo valor particular (especificar) de cada subconjunto de valores posibles.

Consideraciones

- Prioridad
 - Alta → caminos felices o errores críticos en el dominio de negocio.
 - Baja → casos de prueba relacionados a validaciones (valores no ingresados, con mal formato, etc.).
- Nombre caso de prueba: describir el escenario de la funcionalidad que se está probando, ¡de forma clara! Por ejemplo: "Ingresar al sitio web con edad menor a 18 años"
- Precondiciones
 - Conjunto de características que tienen que cumplirse en el contexto de la funcionalidad, para que pueda ser ejecutada.
 - Deben ser valores concretos, específicos. Por ej: "El usuario Juan está logueado con permiso de administrador." - "La fecha actual es 25/10/2022" - "Las coordenadas del GPS son: 41°24'12.2" N" - "La tarjeta tiene el número: XXXX XXXX XXXX XXXX"
 - Los datos que se seleccionan de entre un grupo determinado (por ejemplo forma de pago), son precondiciones que deben estar previamente cargadas en el sistema.
- Pasos
 - Siempre arrancan seleccionando la opción que desencadena la funcionalidad: "El *nombreUsuario* ingresa a la opción ..."
 - Después: "El *nombreUsuario* ingresa/selecciona ..."
 - No tiene que haber ambigüedad, pensar que otra persona los tiene que leer y ejecutar.
 - Normalmente finalizan con un botón de confirmación o algo así.
- Resultado esperado
 - Puede mostrar varias cosas el sistema.
 - Deben ser resultados concretos, específicos.
 - Suelen ser mensajes de advertencia/error, listados, posición en el mapa, etc. pero siempre indicando con un dato particular. Por ejemplo: El sistema muestra el mensaje de advertencia "Debe seleccionar una fecha".

- Se debe relacionar la salida con el paso en la cual se produce.

Estrategia de Pruebas de Caja Negra □ Concepto General La **Caja Negra** es una técnica de testing donde **no se conoce ni se analiza el código interno** del software. El sistema se trata como una “caja” que recibe **entradas** y genera **salidas**, sin importar cómo internamente las procesa.

El objetivo es **verificar que el software haga lo que debe hacer** según las especificaciones, comparando los **resultados obtenidos** con los **resultados esperados**.

Cómo se realiza

1. Se ingresa un conjunto de **datos de prueba** (entradas).
2. Se observa la **salida generada**.
3. Se compara con el **resultado esperado**.
4. Si coinciden → la prueba pasa □.
Si difieren → se detecta un defecto □.

Importante

- Se prueban **entradas válidas e inválidas**.
- En sistemas transaccionales, se deben probar **todas las secuencias posibles** de operaciones previas, no solo los datos individuales.
- Se enfoca en el **comportamiento funcional**, no en la estructura del código.

Clasificación de técnicas de Caja Negra

1. Métodos basados en especificaciones Se apoyan en la documentación del sistema (requerimientos, casos de uso, historias de usuario).

Incluyen:

- **Partición de Equivalencias**
- **Análisis de Valores Límite**

2. Métodos basados en experiencia Usan la intuición y conocimiento **del tester para encontrar defectos**.

Incluyen:

- **Adivinanza de defectos**: el tester imagina posibles errores típicos (por experiencia).
Ejemplo: “Seguro el sistema no valida cuando el campo DNI está vacío.”
- **Testing exploratorio**: mientras prueba, el tester aprende sobre el sistema y crea **nuevos casos sobre la marcha**.
Ej: al probar el login, el tester nota que el botón “Olvidé mi contraseña” no funciona, entonces prueba más variaciones.

Ejemplo general de Caja Negra

Supongamos una **pantalla de inicio de sesión** que pide usuario y contraseña:

Entrada	Resultado Esperado
Usuario y contraseña válidos	Acceso al sistema
Usuario válido, contraseña incorrecta	Mensaje: “Contraseña inválida”
Usuario vacío	Mensaje: “Debe ingresar un usuario”
Contraseña vacía	Mensaje: “Debe ingresar una contraseña”

□ El tester **no necesita ver el código**, solo verificar si el sistema responde correctamente ante cada entrada.

Partición de Equivalencias □ Concepto La **partición de equivalencias** busca **reducir la cantidad de pruebas** necesarias, agrupando las posibles entradas en **clases de equivalencia**. Cada clase representa un conjunto de valores que **producen el mismo resultado**. Así, **basta probar un valor de cada clase** para cubrir todo el grupo.

□ Procedimiento Paso a Paso

1. **Identificar las condiciones externas** de entrada y salida.
Ejemplo: en un formulario de registro, las entradas pueden ser “nombre”, “edad”, “email”.
2. **Dividir en subconjuntos o clases de equivalencia** (válidas y no válidas).
Ejemplo para “edad”:
 - Clase válida: $18 \leq \text{edad} \leq 99$
 - Clase no válida 1: $\text{edad} < 18$
 - Clase no válida 2: $\text{edad} > 99$
 - Clase no válida 3: edad no numérica o campo vacío
3. **Seleccionar un valor representativo** de cada clase para armar los casos de prueba.
Ejemplo:
 - Edad = 25 (válida)
 - Edad = 10 (no válida – menor de edad)
 - Edad = 120 (no válida – supera el límite)
 - Edad = “abc” (no válida – formato incorrecto)

Ejemplo completo **Funcionalidad**: registro de usuario – el campo “Edad” debe aceptar valores entre 18 y 99.

Salida esperada: mensaje de éxito o mensaje de error.

Clase de Equivalencia	Valor elegido	Resultado Esperado
Válida (18–99)	25	Registro exitoso
No válida (<18)	15	Mensaje: “Debe ser mayor de edad”
No válida (>99)	120	Mensaje: “Edad no válida”
No válida (no numérica)	“abc”	Mensaje: “Formato inválido”

□ De este modo, en lugar de probar 82 valores válidos (de 18 a 99), **solo se prueban 4 casos representativos**.

Consideraciones prácticas

- **Prioridad Alta**: casos críticos o principales (por ejemplo, flujos de negocio normales).
- **Prioridad Baja**: validaciones simples (campos vacíos, formatos erróneos).

- **Nombre del caso de prueba:** claro y descriptivo
 - Ejemplo: "Registrar usuario con edad menor a 18 años"
- **Precondiciones:** estado necesario antes de ejecutar el caso.
 - Ejemplo: "El sistema está encendido y el formulario de registro abierto."
- **Pasos:** instrucciones precisas, sin ambigüedad.
 - Ejemplo: "El usuario ingresa edad = 15 y presiona 'Aceptar'."
- **Resultado esperado:** siempre específico.
 - Ejemplo: "El sistema muestra el mensaje: 'Debe ser mayor de edad'."

Ventajas de la Partición de Equivalencias

- Reduce el número de pruebas sin perder cobertura.
- Aumenta la eficiencia del testing.
- Facilita la detección de errores comunes.
- Permite reproducir defectos de manera sistemática.

Análisis de Valores Límite (AVL) Concepto

El Análisis de Valores Límite es una extensión de la Partición de Equivalencias que se centra en los extremos de los rangos válidos e inválidos de las entradas.

La idea principal es que los errores suelen aparecer en los bordes de los intervalos, porque allí es donde los desarrolladores suelen cometer errores de comparación (por ejemplo, usar < en lugar de <=).

En vez de probar todos los valores posibles, se prueban los límites y sus alrededores.

Cómo se aplica

Para cada condición de entrada que tiene un rango (por ejemplo, edad, cantidad, monto, fecha, etc.), se prueban:

- El valor mínimo del rango.
- El valor máximo del rango.
- Los valores inmediatamente adyacentes (uno por debajo y uno por encima).

Esto permite detectar errores típicos como:

- Inclusión o exclusión incorrecta de límites.
- Cálculos erróneos cuando el valor está justo al borde.
- Problemas de redondeo o truncamiento.

Ejemplo 1 — Edad de usuario Supongamos que un sistema solo permite registrarse a usuarios de 18 a 60 años inclusive.

Tipo de prueba	Valor de prueba	Resultado esperado
Justo debajo del mínimo	17	<input type="checkbox"/> Error: "Debe tener al menos 18 años"
En el límite inferior	18	<input type="checkbox"/> Registro exitoso
Valor medio	40	<input type="checkbox"/> Registro exitoso
En el límite superior	60	<input type="checkbox"/> Registro exitoso
Justo por encima del máximo	61	<input type="checkbox"/> Error: "Edad máxima permitida: 60 años"

Con solo 5 casos de prueba, se verifica todo el rango de edad posible, incluyendo los puntos más propensos a error.

Ejemplo 2 — Fecha válida Supongamos que una app permite programar turnos del 1 al 31 de marzo. Casos de prueba:

- Fecha = 28/02 → Error: "Fuera de rango permitido."
- Fecha = 01/03 → Turno válido.
- Fecha = 31/03 → Turno válido.
- Fecha = 01/04 → Error: "Fuera de rango permitido."

Si el desarrollador usa una comparación incorrecta (por ejemplo <31 en lugar de <=31), el AVL lo detecta.

Ejemplo 3 — Monto de transacción Si el sistema acepta montos entre \$100 y \$10.000, los casos de prueba serían:

Caso	Monto	Resultado esperado
A	99	<input type="checkbox"/> Error: "Monto mínimo \$100"
B	100	<input type="checkbox"/> Transacción aceptada
C	10.000	<input type="checkbox"/> Transacción aceptada
D	10.001	<input type="checkbox"/> Error: "Monto máximo \$10.000"

Aplicación combinada con partición de equivalencias El AVL complementa la Partición de Equivalencias:

- Primero se agrupan los valores en clases válidas/no válidas.
- Luego se seleccionan los valores de borde de cada clase.

Ejemplo: Campo "nota" con rango de 0 a 10.

Clase de equivalencia	Valores límite	Casos a probar
No válido bajo	-1	<input type="checkbox"/> "Valor fuera de rango"
Válido	0, 10	<input type="checkbox"/> "Aprobado/Reprobado según nota"
No válido alto	11	<input type="checkbox"/> "Valor fuera de rango"

Ventajas

- Detecta defectos comunes de comparación (<, >, <=, >=).
- Reduce el número de pruebas necesarias con alta efectividad.
- Es sistemático y fácil de automatizar.
- Muy útil en validaciones numéricas, fechas y rangos de datos.

Desventajas

- No cubre combinaciones complejas de condiciones múltiples (ej. varias entradas dependientes).

- No prueba lógica interna del programa (ya que es de tipo "caja negra").
- Puede pasar por alto errores lógicos que no dependen de los límites.

Ejemplo completo de caso de prueba documentado

Funcionalidad: Registro de nuevo cliente. **Condición externa:** Edad. **Regla de negocio:** Solo se permiten edades entre 18 y 60 años. **Casos de prueba AVL:**

Nº	Descripción	Entrada	Resultado esperado
1	Edad menor al límite inferior	17	Mensaje: "Debe tener al menos 18 años"
2	Edad igual al límite inferior	18	Registro exitoso
3	Edad igual al límite superior	60	Registro exitoso
4	Edad mayor al límite superior	61	Mensaje: "Edad máxima permitida: 60 años"

Caja blanca

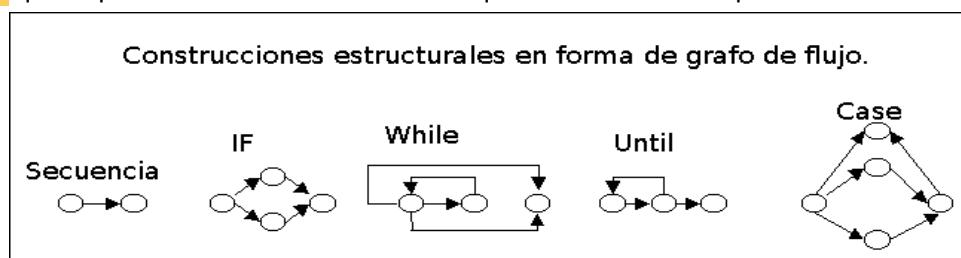
En estos se dispone del código (puede ser también pseudocódigo o un diagrama de flujo). Nos permiten diseñar casos de prueba, maximizando la cantidad de defectos con la menor cantidad de casos de prueba posibles.

Tiene dos fallas: La primera es que el número de caminos lógicos únicos puede ser sumamente grande, tomando muchísimo tiempo de probar absolutamente todas de ellas. La segunda falla es que las pruebas de camino exhaustivas no aseguran que el programa sea el correcto para las especificaciones, tampoco detecta caminos faltantes ni determina errores de datos sensibles.

Hay distintas coberturas con nivel diferente (la forma de recorrer los distintos caminos que provee el código para desarrollar una funcionalidad):

Cobertura de enunciados o caminos básicos

- Objetivo → encontrar todos los caminos independientes de una funcionalidad que deben recorrerse (testear) al menos una vez, para probar cada uno de ellos con casos de prueba.
- Permite obtener una métrica denominada complejidad ciclomática (M), que representa la cantidad de caminos independientes que posee una funcionalidad, y nos da un límite inferior para el número de casos de prueba que hay que construir, para ejecutar todas las instrucciones de la funcionalidad al menos una vez.
- El procedimiento es:
 1. En primer lugar, se requiere representar la funcionalidad a través de un grafo de flujo (los algoritmos que implementen métodos recursivos quedan fuera de esta prueba de cobertura)



2. Luego se calcula la complejidad ciclomática (M). Para esto hay dos formas:

- a. $M = E - N + 2*P$, siendo:
 - M = complejidad ciclomática.
 - E = número de aristas del grafo.
 - N = número de nodos del grafo.
 - P = número de componentes conexos o nodos de salida.
- b. También se puede obtener la complejidad ciclomática como $M = \text{número de regiones cerradas} + 1$.

Caja Blanca

Cobertura de enunciados o caminos básicos

$$M = E - N + 2*P$$

$$M = 12 - 10 + 2*1$$

$$M = 4$$

$$M = \text{Número de regiones} + 1$$

$$M = 3 + 1$$

Caja Blanca

Cobertura de enunciados o caminos básicos

$$M = E - N + 2*P$$

$$M = 12 - 10 + 2*1$$

$$M = 4$$

$$M = \text{Número de regiones} + 1$$

$$M = 3 + 1$$

r1

r2

r3

En el ejemplo, la cantidad mínima de casos de prueba para garantizar la cobertura de enunciados es de 4. Esto no quita que se puedan construir más de 4, pero esa es la menor cantidad de casos de prueba, que me permite maximizar la detección de defectos.

3. Se define el conjunto mínimo de caminos independientes (asignando valores que provocan ir tomando cada uno de los caminos de la funcionalidad). No son casos de prueba. Siguiendo el ejemplo:

TC 1	TC 2	TC 3	TC 4
N1 = 8	N1 = 8	N1 = 4	N1 = 4
N2 = 4	N2 = 4	N2 = 8	N2 = 8
N3 = 4	N3 = 8	N3 = 4	N3 = 8

4. Luego se diseñan y ejecutan los casos de prueba que permitan la ejecución de cada uno de los caminos identificados, donde los datos de la tabla de arriba se mapean a las precondiciones o a valores que ingresa el usuario en los casos de prueba (dependiendo de cómo es la funcionalidad).
 5. Se ejecutan los casos de prueba y se comprueba que los resultados sean los esperados.
- Dependiendo del valor de la complejidad ciclomática (M), el programa puede entrar en una de las siguientes clasificaciones (heurística):

Complejidad Ciclomática	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, Muy alto riesgo

Cobertura de sentencias

- Sentencia: cualquier instrucción como asignación de variable, invocación de métodos, etc. Las estructuras de control NO son sentencias, son decisiones.
- Objetivo → buscar la cantidad mínima de casos de pruebas que me permitan pasar, ejecutar o evaluar todas las sentencias.
 - Ejemplo: Partiendo de la siguiente porción de código, podemos determinar que existen 2 sentencias (de asignación de variables). Entonces, para ejecutar o evaluar las sentencias, es necesario solo un caso de prueba, asignando valores a "A", "C", "B" y "D" es posible ejecutar ambas sentencias ya que las condiciones de IF son independientes entre sí.

```

IF (A>0 && C==1)
    X = X + 1
IF (B==3 || D<0)
    Y=0
END
  
```

TC1
A=5
C=1
B=3
D=-3

Cobertura de decisión

- Decisión: estructura de control concreta, y dentro de cada decisión existen combinaciones que están unidas por operaciones booleanas de condiciones.
- Objetivo → buscar la cantidad mínima de casos de prueba que me permitan evaluar todas las ramas de las decisiones. Esto apunta a que el código tome cada rama de forma correcta, es decir, evaluar que la decisión derive en la rama del true y en la rama del false cuando corresponda en cada caso.
- En una decisión IF, si o si necesitamos dos casos de prueba: rama true y rama false.
- Las decisiones son las que se encuentran dentro de los paréntesis de los IF, y cada uno de los términos son condiciones, es decir A>0 y C==1, y el conjunto de las condiciones unidas por el booleano, representan una combinación (A>0 && C==1).
- En esta cobertura, no interesa qué condición (o combinación de condiciones) hay dentro de la decisión, siempre y cuando se garantice que la decisión es evaluada en su rama verdadera y en su rama falsa.

- En el ejemplo, con tan solo dos casos de prueba es suficiente, ya que las decisiones son independientes entre sí (no están anidados), por lo que independientemente de la rama que se evalúe en la primera decisión, voy a poder valorar cualquiera en la segunda. Entonces en el TC1 se evalúa la rama de “true” en ambas decisiones, y en TC2 se evalúa la rama de “false” en la dos.

Cobertura de condición

- Condiciones: son cada una de las evaluaciones lógicas dentro de una decisión, que están unidas por operadores lógicos.
- Objetivo → buscar la cantidad mínima de casos de pruebas que me permitan evaluar cada una de las condiciones, en su valor verdadero y en su valor falso, independientemente de que rama se tome en la decisión en la que se encuentre la condición (recordar que la decisión está formada por 1 o más condiciones combinadas de forma booleana).
- Siguiendo el ejemplo anterior, son necesarios 2 casos de prueba, ya que las condiciones son independientes y se pueden evaluar todas las condiciones en true en una prueba, y todas las condiciones en false en otra prueba, debido a que no importa la rama que tome cada decisión.

Cobertura de decisión condición

- Objetivo → evaluar las ramas verdaderas y falsas de las decisiones, y a su vez evaluar valores verdaderos y falsos de las condiciones.
- Siguiendo el ejemplo anterior, se necesitan 2 test cases, los cuales permiten evaluar las condiciones y decisiones al mismo tiempo en un mismo caso de prueba, con los valores de la imagen.

Cobertura múltiple

- Objetivo → evaluar el combinatorio de todas las condiciones, en todos sus valores de verdad posibles.
- Por ejemplo: Si tuviésemos un caso como el representado en el grafo de flujo, los casos de prueba van a ser 7, ya que solo el valor de A y B verdaderos al mismo tiempo, permiten ejecutar escenarios en donde se incluye la decisión de C y D. Este es un escenario con decisiones dependientes entre sí, si fuesen independientes, con 4 pruebas alcanza.

Pruebas de Caja Blanca □ Concepto general La **caja blanca** (o white box testing) se basa en **el conocimiento interno del código**. El tester tiene acceso al código fuente, pseudocódigo o diagramas de flujo, y diseña los casos de prueba en función de la lógica interna del programa, no solo de las entradas y salidas (como en la caja negra).

El objetivo es maximizar la detección de defectos con la menor cantidad posible de casos de prueba.

□ Limitaciones

- **Demasiados caminos posibles**: el número de combinaciones lógicas crece exponencialmente, y probar todas es imposible.
- **No asegura la corrección funcional**: aunque se prueben todos los caminos del código, eso no garantiza que el software cumpla los requisitos.
- **No detecta caminos faltantes** (es decir, lógica que debería existir pero no se implementó).

□ Tipos de cobertura (niveles de profundidad)

Existen distintas formas de recorrer el código, según el nivel de detalle que se quiera cubrir.

□ 1. Cobertura de Enunciados o Caminos Básicos

- Objetivo: Ejecutar cada camino lógico independiente al menos una vez.

Cada camino representa una secuencia única de instrucciones que puede seguir el programa.

□ Herramienta clave: Complejidad Ciclomática (M)

```

IF (A>0 && C==1)
    X = X + 1
IF (B==3 || D<0)
    Y=0
END
  
```

TC1	TC2
A=5	A=5
C=1	C=5
B=3	B=5
D=-3	D=5

TC1	TC2
A=0	A=5
C=1	C=5
B=3	B=5
D=-3	D=5

```

IF (A>0 && C==1)
    X = X + 1
IF (B==3 || D<0)
    Y=0
END
  
```

TC1	TC2
A=5	A=0
C=1	C=5
B=3	B=5
D=-3	D=5

1 DECISIÓN	A	F	F	V	V	V	V	V
	B	F	V	F	V	V	V	V
2 DECISIÓN	C	-	-	-	V	V	F	F
	D	-	-	-	V	F	V	F

Es una métrica que mide la cantidad mínima de caminos independientes de un programa.

Fórmulas:

1. $M = E - N + 2P$
 - E = número de aristas (flechas del grafo de flujo)
 - N = número de nodos (bloques o decisiones)
 - P = número de componentes conexos o salidas del grafo
2. $M = \text{número de regiones cerradas} + 1$

Ejemplo: Supongamos el siguiente pseudocódigo:

```
if A > 0:  
    print("Positivo")  
else:  
    print("No positivo")
```

```
if B == 1:  
    print("B es uno")
```

- **Caminos:**
 1. $A > 0$ y $B == 1$
 2. $A > 0$ y $B != 1$
 3. $A \leq 0$ y $B == 1$
 4. $A \leq 0$ y $B != 1$

→ $M = 4$ (por tanto, se necesitan al menos 4 casos de prueba).

Esto permite recorrer todas las ramas del código una vez, asegurando que cada instrucción fue ejecutada al menos una vez.

2. Cobertura de Sentencias

Objetivo: Ejecutar todas las instrucciones o líneas de código al menos una vez. No interesa la lógica de decisiones, solo que cada sentencia se ejecute.

Ejemplo:

```
if A > 0:  
    X = 1  
if C == 1:  
    Y = 2
```

- Hay dos sentencias ($X=1$ y $Y=2$).
- Con un solo caso de prueba (por ejemplo, $A=1$ y $C=1$), se ejecutan ambas.

Cobertura completa de sentencias con un test.

3. Cobertura de Decisión

Objetivo: Evaluar todas las ramas (true/false) de cada decisión.

Ejemplo:

```
if A > 0:  
    X = 1  
else:  
    X = -1
```

- Necesitamos dos casos de prueba:
 - $A = 5 \rightarrow$ rama true
 - $A = -3 \rightarrow$ rama false

Así se garantiza que cada camino de decisión se ejecuta.

Si hay dos if independientes, bastan dos casos de prueba que cubran true/false en ambos.

4. Cobertura de Condición

Objetivo: Verificar que cada condición lógica dentro de una decisión puede ser verdadera y falsa, sin importar qué rama se tome.

Ejemplo:

```
if (A > 0 and B == 1):  
    print("Condición cumplida")
```

- **Condiciones:**
 1. $A > 0$
 2. $B == 1$

Casos de prueba:

Nº	A	B	Resultado esperado
1	1	1	True/True
2	-1	0	False/False

Cada condición individual se evaluó como true y false, cumpliendo cobertura.

5. Cobertura de Decisión-Condición

Objetivo: Asegurar que:

- Cada decisión toma true/false.
- Cada condición dentro de esa decisión también toma true/false.

Ejemplo:

```
if (A > 0 and B == 1):  
    print("OK")
```

Casos necesarios:

Caso	A	B	Evaluación	Rama
1	1	1	True/True	True
2	-1	0	False/False	False

Con estos 2 casos se cubren ambas condiciones y ambas ramas.

6. Cobertura Múltiple

Objetivo: **Evaluar** todas las combinaciones posibles de valores de verdad de las condiciones.

Ejemplo:

if (A or B):

...

- A y B son booleanos.
- Combinaciones posibles:
 1. A=True, B=True
 2. A=True, B=False
 3. A=False, B=True
 4. A=False, B=False

→ Se necesitan **4 casos de prueba** para lograr cobertura múltiple.

Si las decisiones son **dependientes entre sí**, puede ser necesario probar más combinaciones (por ejemplo, hasta 7 u 8 caminos distintos).

Ejemplo completo de aplicación

Supongamos esta función:

```
def validar_edad(edad):
    if edad < 18:
        return "Menor"
    elif edad <= 60:
        return "Adulto"
    else:
        return "Mayor"
```

Tipo de cobertura	Casos mínimos	Ejemplos
Sentencias	1	edad = 30
Decisiones	3	17, 30, 70
Condiciones	3	Igual que anterior (todas se evalúan en true/false)
Caminos básicos (M=3)	3	M coincide con las 3 rutas posibles

Conclusión

Cobertura	Qué verifica	Casos típicos
Sentencias	Que todas las líneas de código se ejecutan.	1 o pocos
Decisión	Que todas las ramas (true/false) se recorren.	2 por decisión
Condición	Que cada condición se evalúa como true y false.	2 o más
Decisión-condición	Ambas cosas simultáneamente.	2+
Múltiple	Todas las combinaciones de verdad posibles.	4, 8 o más según casos

Ejemplo:

Si una función tiene $M = 4 \rightarrow$ se necesitan **al menos 4 casos de prueba** para cubrir todos los caminos.

Resumen comparativo de estrategias

Estrategia	Conocimiento del código	Nivel	Ejemplo típico	Ventaja principal
Caja negra	No	Funcional	"Validar login con contraseña inválida."	Evaluá el comportamiento del sistema.
Caja blanca	Sí	Estructural	"Ejecutar todas las ramas de un if-else."	Evaluá la lógica interna y calidad del código.

Tipos de prueba

Smoke Test

Es un tipo de prueba que se hace para validar que no haya fallas groseras, de gran magnitud, en el producto de software. Se realiza antes de comenzar el ciclo 0. Nos ahorra empezar a testear formalmente si encuentra un error, porque todavía hay algo que corregir.

Consiste en una corrida rápida para ver si el producto está en condiciones de pasar al ciclo de prueba.

Testing Funcional

Controla que el software se comporte de la misma manera que lo especificado en la documentación, cumpliendo con las funcionalidades y características definidas. Se basa en los requerimientos funcionales y el proceso de negocio. Hay testing funcional basado en dos aspectos:

- **Basado en requerimientos:** cuando se prueban **requerimientos específicos**, apunta a probar una funcionalidad sola (utilizan a los requisitos definidos en una ERS o los acuerdos que contienen las pruebas de usuario y los criterios de aceptación de una US para realizar las pruebas.)
- **Basado en proceso de negocio:** cuando se prueba **un proceso de negocio completo**, es decir, **se prueba todo el proceso**. Por ejemplo, en una venta se prueba la búsqueda del artículo, la selección y facturación del mismo.

Testing No Funcional

Se basa en cómo trabaja el sistema haciendo foco en los requerimientos no funcionales (foco en el “como”, no

en el “que”). Son las pruebas más complejas, por su gran dependencia al entorno del sistema, por lo que debe ser lo más parecido posible al del cliente. Sin embargo, se tienen características que no pueden probarse, como el ancho de banda del internet, la seguridad o performance en una determinada situación. Y se pueden solucionar con las pruebas de aceptación. Incluye varios tipos de prueba como:

- **Performance:** Se ve el tiempo de respuesta (escenario esperado respecto a los tiempos de respuesta) y la concurrencia. Deben pasar esta prueba sí o sí.
- **Carga:** no solo mira performance, mira el comportamiento de los dispositivos de hardware (procesadores, discos, etc.) y de las comunicaciones.
- **Stress:** queremos forzar al sistema para que falle, se lo somete a condiciones más allá de las normales. Se ve el tiempo que hay que esperar para probar nuevamente el sistema y la robustez del sistema (tiempo de recuperación).
- **Mantenimiento:** para ver si el producto está en condiciones de evolucionar, se controla que haya documentación, manual de configuración, etc. Se observa la facilidad que existe para corregir un defecto.
- **Usabilidad:** que sea cómodo para el usuario.
- **Portabilidad:** se prueba en los distintos entornos acordados con el cliente.
- **Fiabilidad:** probamos que podemos depender del sistema. Resultados que se obtienen, seguridad física del software.
- **De interfaz de usuario:** suelen ser más complejas las GUI's que las interfaces de comandos.
- **De configuración.**

Test Driven Development – TDD

Este es un tipo de proceso en el que nos enfocamos en construir primero la prueba unitaria cuando tengo los requerimientos y luego codear el componente. Si pienso en las pruebas después tengo menos errores, el fundamento filosófico de esto es que si no tengo claro que tengo que hacer no puedo crear pruebas para eso. Es una técnica de desarrollo del software que involucra dos prácticas:

1. **Test first development:** en esta técnica primero se escriben las pruebas unitarias referentes a la característica de producto a implementar. Definidas las pruebas unitarias, se piensa en la codificación necesaria para que las pruebas se ejecuten con éxito. Dado que las pruebas unitarias prueban unidades concretas de código, esta técnica obliga al desarrollador a modularizar su codificación haciendo que los métodos o clases a probar tengan una única responsabilidad. Además de pruebas unitarias, pueden incluirse en primera instancia pruebas de integración.
2. **Refactoring:** esta técnica consiste en reestructurar el código existente sin modificar el comportamiento que el mismo provee. Implica la mejora de aspectos no funcionales del software, cuyo objetivo es mejorar la claridad del código y reducir su complejidad, con el fin de hacerlo más mantenable.

Tipos de Prueba

Smoke Test (Prueba de Humo)

- **Objetivo:** detectar rápidamente si el sistema tiene errores graves antes de invertir tiempo en pruebas más profundas.
- **Momento de aplicación:** antes de comenzar el **ciclo 0** (es decir, antes del testing formal).
- **Método:** se ejecuta una revisión rápida y general del sistema para verificar que las funciones básicas no fallen.

Ejemplo:

Si se desarrolla un sistema de login, antes de iniciar las pruebas completas, se ejecuta un **Smoke Test** para comprobar si:

- Se puede abrir la aplicación.
- El botón “Iniciar Sesión” responde.
- No hay errores al cargar la pantalla principal.

Si alguna de estas funciones falla, no tiene sentido seguir con las pruebas hasta corregirla.

Testing Funcional

- **En qué se basa:** en qué hace el sistema según los **requerimientos funcionales** definidos (documentación o historias de usuario).
- **Objetivo:** verificar que el software cumpla con las **funcionalidades esperadas**.

Tipos:

1. **Basado en Requerimientos:**

- Se prueba cada requisito individualmente.
- Busca confirmar que cada funcionalidad específica se comporta según lo acordado.

Ejemplo:

Si el requisito dice “El usuario puede registrarse ingresando email y contraseña”, el caso de prueba sería:

- Ingresar email válido y contraseña → resultado esperado: registro exitoso.
- Ingresar email inválido → resultado esperado: mensaje de error.

2. Basado en Procesos de Negocio:

- Evalúa **procesos completos** que involucran varias funciones.
- Garantiza que todo el flujo funcione correctamente.

Ejemplo:

En una tienda online, se prueba **todo el proceso de compra**:

Buscar producto → añadir al carrito → pagar → recibir comprobante.

Si alguna parte falla, el proceso de negocio completo no se cumple.

Testing No Funcional

- **En qué se basa:** en **cómo** trabaja el sistema, no en qué hace.
- **Objetivo:** evaluar aspectos como rendimiento, seguridad, facilidad de uso o estabilidad.
- **Desafío:** dependen mucho del entorno real, por lo que se intenta replicarlo lo más fielmente posible.

□ Tipos principales:

1. **Performance (Rendimiento):** Mide los tiempos de respuesta y la capacidad de atender usuarios concurrentes.
Ejemplo: comprobar que una búsqueda devuelve resultados en menos de 2 segundos con 50 usuarios conectados.
2. **Carga:** Evalúa cómo responde el sistema ante **volúmenes elevados de trabajo**.
Ejemplo: subir 10.000 registros a la base de datos y ver si el sistema mantiene un rendimiento aceptable.
3. **Stress:** Empuja el sistema **más allá de sus límites** para ver hasta dónde resiste.
Ejemplo: probar con 1.000.000 de solicitudes simultáneas hasta que el sistema colapse y analizar su recuperación.
4. **Mantenimiento:** Comprueba la **facilidad para modificar o mejorar** el sistema sin romperlo.
Ejemplo: cambiar una función y verificar que el resto del sistema sigue funcionando correctamente.
5. **Usabilidad:** Evalúa la **experiencia del usuario**: claridad, facilidad de navegación, comprensión de los mensajes.
Ejemplo: probar si un usuario sin experiencia puede completar una compra sin ayuda.
6. **Portabilidad:** Verifica que el sistema funcione en distintos entornos o dispositivos.
Ejemplo: probar una app web en Chrome, Firefox y Safari.
7. **Fiabilidad:** Mide la **estabilidad del sistema a lo largo del tiempo** y su capacidad para producir resultados confiables.
Ejemplo: ejecutar 1000 operaciones seguidas sin que el sistema se congele o devuelva resultados erróneos.
8. **Interfaz de Usuario:** Asegura que la interfaz gráfica funcione correctamente (botones, menús, transiciones).
Ejemplo: verificar que los botones respondan al clic correcto y que los formularios muestren validaciones adecuadas.
9. **Configuración:** Evalúa que el software pueda adaptarse a diferentes configuraciones del sistema.
Ejemplo: probar la instalación en distintas versiones de Windows o Linux.

Test Driven Development (TDD)

- **Definición:** técnica de desarrollo donde **primero se escriben las pruebas unitarias** y **luego el código** que las hace pasar.
- **Filosofía:** “Si no puedo probarlo, no sé exactamente qué estoy construyendo”.

□ Etapas del proceso:**1. Test First Development:**

- Se definen las **pruebas unitarias** antes del código.
- Estas pruebas establecen el comportamiento esperado del componente.
- Luego se codifica solo lo necesario para que la prueba pase.

Ejemplo: Supongamos que queremos una función `sumar(a, b)` que devuelva la suma:

- Primero se crea la prueba:
- `def test_sumar():`
- `assert sumar(2, 3) == 5`
- Luego se crea la función:
- `def sumar(a, b):`
- `return a + b`
- Se ejecuta la prueba y pasa □.

Esto fuerza a escribir código **mínimo, limpio y funcional**.

2. Refactoring (Refactorización):

- Una vez que todas las pruebas pasan, se **mejora el código** sin alterar su comportamiento.
- Se busca simplificar, eliminar duplicaciones y mejorar la legibilidad.

Ejemplo: Si el código tiene varias funciones parecidas, se unifica en una más general, ejecutando las mismas pruebas para asegurar que el comportamiento no cambió.

□ Resumen Visual

Tipo de prueba	Enfocado en	Ejemplo
Smoke Test	Validar que el sistema sea testeable	Comprobar si la app inicia correctamente
Funcional	Que el sistema haga lo que debe	Probar el login o una compra completa
No Funcional	Cómo lo hace (rendimiento, usabilidad, etc.)	Medir tiempos de respuesta o facilidad de uso
TDD	Desarrollar escribiendo primero las pruebas	Crear test antes de programar la función

Mejora continua de procesos con Kanban

Kanban

No es ni:

- Un proceso de desarrollo de software.
- Una metodología de administración de proyectos.

Definición

Kanban es un **enfoque para la gestión de formas de trabajo (procesos)** para obtener mejora continua.

Esto lo logra a través del principio de “empieza por donde estés”, es decir, no plantea introducir cambios revolucionarios, sino que **introduce mejoras graduales a un proceso de desarrollo de software o a una metodología de administración de proyectos ya existente** en una organización. Esto permite **reducir la resistencia al cambio** por parte de las personas, fomentando la evolución gradual de los procesos existentes.

Este enfoque surge con estudios de **Toyota**, para mejorar las técnicas de almacenamiento y tiempo de stockeo en supermercados.

Para su aplicación en el desarrollo de software, al igual que Lean fue necesaria una adaptación.

Kanban aprovecha los principios de Lean:

- Definiendo el **valor** desde la perspectiva del cliente.
- **Limitando el trabajo** en proceso WiP.
- Identificando y **eliminando desperdicios**.
- Identificando y **eliminando las barreras en el flujo**, es decir, todo lo que **atrasaría el proceso**: Relacionado con el principio de lograr entregar lo antes posible.
- **Cultura de mejora continua**.

¿Qué es Kanban? Kanban no es una metodología ni un proceso de desarrollo de software.

Es un **enfoque de gestión visual** que busca **mejorar continuamente la forma de trabajar** en un equipo u organización.

Idea central: “Empieza por donde estás” No intenta cambiar todo de golpe, sino mejorar **gradualmente** los procesos existentes.

Este enfoque fue **inspirado por Toyota**, que lo aplicó originalmente para optimizar la **gestión de inventarios y tiempos de stock** en sus fábricas.

Luego, se **adaptó al desarrollo de software y gestión de proyectos** para reducir desperdicios, aumentar la eficiencia y fomentar la mejora continua.

Principios que Kanban toma del enfoque Lean

1. **Definir el valor desde la perspectiva del cliente:**
Todo esfuerzo debe aportar algo útil para el usuario final.
 Ejemplo: No tiene sentido agregar una función decorativa en una app si no mejora la experiencia del usuario.
2. **Limitar el trabajo en progreso (WiP):**
Se evita tener demasiadas tareas abiertas al mismo tiempo.
Menos tareas → más enfoque → entregas más rápidas.
3. **Eliminar desperdicios:**
Detectar y eliminar tareas innecesarias, esperas o retrabajos.
 Ejemplo: reducir tiempo perdido esperando aprobación de un cambio menor.
4. **Eliminar barreras en el flujo:**
Detectar todo lo que **retrasaría la entrega**: bloqueos, dependencias, demoras, etc.
5. **Fomentar una cultura de mejora continua:**
Siempre buscar cómo hacer las cosas mejor, sin necesidad de una “gran revolución”.

Prácticas de Kanban

Visualización del trabajo

El método Kanban utiliza un mecanismo de **señalización para hacer visible el trabajo** que es requerido por el cliente. Para ello, divide el **trabajo en piezas de trabajo** (que pueden ser U.S., Features, bugs, temas, épicas, cambios, etc.) y **las escribe en tarjetas señalizadoras (kan-bans)** que serán ubicadas en **tableros kanban**.

Estas tarjetas permiten indicar cuando el trabajo **se puede “pullar”** para realizar un trabajo determinado, además de **señalar en qué parte del proceso** se encuentra.

El tablero kanban **representa un sistema de flujo** en el que las **piezas de trabajo fluyen** a través de las diversas etapas de un proceso, de izquierda a derecha. **Cada etapa es representada por una columna del tablero**, sobre las cuales se aplica la **teoría de colas**.

Se logra transparencia al hacer visible para todo el equipo el trabajo mediante un tablero kanban siempre disponible y a la vista.

Existen dos tipos de columnas:

- **Producción:** piezas sobre las que se está trabajando.
- **Acumulación:** piezas que están listas para pasar a la siguiente etapa (sistema de arrastre).

Limitar el WiP (Work in Progress)

Se deben asignar límites concretos a cuántas piezas pueden estar en progreso en cada etapa del flujo de trabajo (en cada columna), para evitar atascamientos o cuellos de botella.

Las políticas para limitar el WiP crean un sistema de arrastre (pull): el trabajo es “arrastrado” al sistema cuando otro de los trabajos es completado y queda capacidad disponible, en lugar de “empujar” estos trabajos al paso siguiente cuando hay nuevo trabajo demandado.

Tener demasiado trabajo no finalizado es un desperdicio de tiempo, de dinero y alarga los tiempos de entrega. Observar, limitar y optimizar la cantidad de trabajo en progreso es esencial para tener éxito con Kanban, consiguiendo mejorar la calidad y el tiempo de entrega de servicio y aumentar la tasa de entrega.

Gestionar el flujo de trabajo

El trabajo fluye sobre un tablero permanente, no existe el concepto de iteración, proceso o proyecto. El tablero puede ser compartido con otros proyectos y otros equipos, ya que se trabaja cada pieza de trabajo de forma individual. Lograr que el flujo sea continuo e ininterrumpido.

Al flujo que se observa en ese tablero, se lo debe analizar, identificando diferentes características del proceso de trabajo: cuellos de botella, recursos ociosos, tiempos de entrega, tiempos de espera, etc.

Hacer explícitas las políticas

Las políticas de proceso deben ser escasas, simples, estar bien definidas, visibles, deben aplicarse siempre, y tienen que ser fácilmente modificables. Es una buena práctica poder cambiar fácilmente las políticas, ya que si producen efectos contraproducentes para nuestro proceso o también se considera que no pueden aplicarse las mismas, deben poder cambiarse.

También es importante visualizar las políticas; por ejemplo, colocando resúmenes entre las columnas donde se describe lo que debe estar hecho antes de que una tarjeta se mueva de una columna a la siguiente.

También se suele colocar el WIP de cada columna.

Las políticas de calidad o DoD deben estar definidas, publicadas y promovidas para lograr no sólo que se cumplan, si no buscar mejorar continuamente.

Mejorar y evolucionar

Kanban propone una cultura de mejora continua, donde no introduce cambios significativos en los procesos de desarrollo, sino que identifica ese proceso, lo visualiza (hacer explícito para todos los miembros) y propone mejoras sobre él, las cuales se van aplicando de forma gradual a lo largo del tiempo.

Círculo de retroalimentación

Son una parte esencial para cualquier proceso controlado que nos ayuda a realizar cambios evolutivos. Se debe definir con qué frecuencia se realizan las reuniones, donde depende en qué contexto se presentan ya que es importante para el resultado.

Si se realizan revisiones demasiadas frecuentes pueden obligar a cambiar cosas que no se vieron con los cambios anteriores, pero si no son demasiadas frecuentes, existe un bajo rendimiento durante mucho tiempo.

Prácticas Fundamentales de Kanban

1. Visualización del trabajo

El tablero Kanban es el corazón del sistema.

Representa el flujo de trabajo de izquierda a derecha, mostrando cada tarea en qué etapa se encuentra.

Cada tarea se representa con una tarjeta (kanban) que contiene información básica (ej. nombre de la tarea, responsable, fecha).

Ejemplo de tablero básico:

Pendiente	En progreso	En revisión	Terminado
Tarea 1	Tarea 2	Tarea 3	Tarea 4

Columnas de un tablero:

- **Producción:** tareas en las que se está trabajando.
- **Acumulación:** tareas listas para pasar a la siguiente etapa (esperando revisión, por ejemplo).

Ventaja: Todo el equipo puede ver el estado del trabajo en tiempo real, lo que da transparencia y coordinación.

2. Limitar el WiP (Work in Progress)

Cada columna del tablero tiene un límite máximo de tareas que pueden estar “en progreso”.

Esto evita los cuellos de botella y obliga a terminar lo que ya se empezó antes de tomar nuevas tareas.

Ejemplo:

- En la columna “En progreso” se define un límite de 3 tareas.
- Si ya hay 3, nadie puede agregar otra hasta que se complete alguna.
- Esto genera un sistema pull (de arrastre): el trabajo se “tira” cuando hay capacidad libre, no se “empuja” automáticamente.

Beneficios:

- Evita multitarea excesiva.
- Mejora la calidad.

- Aumenta la velocidad de entrega real.

Ejemplo práctico: En un equipo de desarrollo, si cada programador toma demasiadas historias a la vez, se retrasan todas. Con un límite de WiP, cada uno se enfoca y libera tareas antes de tomar nuevas.

3. Gestionar el flujo de trabajo

- Kanban **no trabaja por iteraciones ni proyectos cerrados.**

El tablero es **continuo y permanente**, y puede ser compartido entre varios equipos o proyectos.

El objetivo es mantener un **flujo continuo** de tareas sin interrupciones.

Se analizan indicadores como:

- Cuellos de botella.
- Tiempos de espera.
- Tiempos de entrega.
- Recursos ociosos.

Ejemplo: Si muchas tareas se acumulan en "En revisión", eso indica que hay un cuello de botella en el equipo de QA.

4. Hacer explícitas las políticas

Las **políticas del proceso** deben estar **claras, visibles y simples**.

Ayudan a todos a entender **cuándo y cómo** una tarea puede avanzar en el flujo.

Ejemplos de políticas:

- Para pasar de "Desarrollo" a "Revisión", la tarea debe tener pruebas unitarias completadas.
- Máximo 2 tareas en progreso por persona.

Estas políticas deben estar **publicadas en el tablero**, junto con los límites de WiP y los **criterios de calidad (DoD – Definition of Done)**.

También deben poder **ajustarse fácilmente** si se detecta que generan bloqueos o no aportan valor.

5. Mejorar y evolucionar

Kanban promueve una **cultura de mejora continua**:

1. Se **observa el proceso actual**.
2. Se **identifican oportunidades de mejora**.
3. Se aplican **cambios graduales** y se evalúan los resultados.

Esto reduce la resistencia al cambio, porque las mejoras son **pequeñas, medibles y consensuadas**.

Ejemplo: Si el equipo nota que las tareas se trapan por falta de revisores, se puede:

- Aumentar temporalmente el límite de revisiones simultáneas.
- O crear una columna extra "Revisión en Pares".

6. Circuito de retroalimentación

Es esencial para **evaluar y ajustar el proceso**.

Consiste en reuniones o revisiones periódicas donde el equipo analiza qué está funcionando y qué se puede mejorar.

Frecuencia: depende del contexto.

- Si son muy frecuentes → se pierde tiempo en cambios prematuros.
- Si son muy espaciadas → los problemas se mantienen demasiado.

Ejemplo: Un equipo puede reunirse cada 2 semanas para analizar:

- ¿Dónde se generaron bloqueos?
- ¿Qué políticas funcionaron?
- ¿Qué cambios aplicar en el próximo ciclo?

Resumen visual

Práctica	Qué busca	Ejemplo
Visualización	Hacer visible el trabajo	Tablero Kanban con columnas "Pendiente / En curso / Hecho"
Limitar WiP	Evitar saturación y multitarea	Máx. 3 tareas "En progreso"
Gestionar flujo	Mantener entrega continua	Detectar y eliminar cuellos de botella
Políticas explícitas	Claridad y coherencia en el proceso	Mostrar reglas y DoD en el tablero
Mejora continua	Evolución gradual del proceso	Ajustar límites y roles según resultados
Retroalimentación	Evaluar y aprender	Reuniones periódicas para analizar el flujo

Ejemplo general de aplicación: Un equipo de desarrollo usa Kanban con las siguientes reglas:

- Tablero con columnas: **Pendiente** → **En desarrollo** → **En revisión** → **Terminado**.
- WiP máximo: 4 tareas en desarrollo, 2 en revisión.
- Política: no se puede pasar a "Terminado" sin revisión de otro miembro.
- Reunión semanal de retroalimentación: se analizan demoras, se ajustan límites y se agregan mejoras.

Resultado:

- Se reducen los tiempos de entrega.
- El equipo detecta bloqueos antes.
- Se incrementa la calidad del producto.

¿Cómo aplicar Kanban?

1. **Empezar con lo que se tiene:** entender el proceso de desarrollo actual que se está utilizando.
2. **Identificar las unidades de trabajo a utilizar (US, CU, defectos).**
3. **Identificar clases de servicio:** diferentes trabajos tienen distintas políticas para tratarlos (**DoD**). Por ejemplo: requerimientos, defectos, desarrollo y solicitudes.
4. **Visualizar el flujo de trabajo** diseñando un tablero representando el flujo de trabajo a través de columnas.

5. Definir políticas para cada clase de servicio identificada. Esto implica acordar los WiP para cada columna, asignar color a las tarjetas kanban, capacidad de trabajo destinada, fechas de entrega, etc.
 6. Identificar cuellos de botella y resolverlos (asignando más recursos o ajustando el WiP donde corresponda).



Aplicación de Kanban

1. Empezar con lo que se tiene

Analizar el proceso actual de desarrollo, sin realizar cambios bruscos al inicio.

2. Identificar unidades de trabajo

Definir los tipos de elementos que se gestionarán en el tablero:

- *Historias de usuario (US)*
 - *Casos de uso (CU)*
 - *Defectos o incidencias*

3. Identificar *clases de servicio*

Determinar los tipos de trabajo que existen y las políticas específicas para cada uno.

Ejemplos: requerimientos, defectos, desarrollos, solicitudes.

4. Visualizar el flujo de trabajo

Diseñar un **tablero Kanban** con columnas que representen cada etapa del proceso (por ejemplo: "Por hacer", "En curso", "Hecho").

5. Definir políticas

Establecer reglas para cada clase de servicio, como:

- Límite de trabajo en progreso (**WIP**)
 - Colores para las tarjetas
 - Capacidad de trabajo
 - Fechas de entrega

6. Identificar cuellos de botella

Detectar dónde se acumula el trabajo y aplicar mejoras, como reasignar recursos o ajustar los límites WIP.

Métricas de Kanban

Las métricas más representativas de Kanban son:

- Cycle Time.
 - Lead Time.
 - Touch Time.
 - Eficiencia del Ciclo de Proceso

Estas métricas están desarrolladas en el tema de métricas de la unidad 2 de este anuncio.

Estas métricas están desarrolladas en el tema de métricas de la unidad 2 de este apunte.	
Métrica	Descripción
Cycle Time	Tiempo total que tarda una tarea desde que comienza hasta que se completa.
Lead Time	Tiempo desde que se solicita una tarea hasta que se entrega.
Touch Time	Tiempo efectivo de trabajo sobre una tarea.
Eficiencia del ciclo de proceso	Relación entre el Touch Time y el Lead Time; mide la productividad real del flujo.

Valores de Kanban

Los valores de Kanban se podrían resumir en una sola palabra, “respeto”. Sin embargo, es importante desgranar esto en una serie de nueve valores.

- **Transparencia:** La creencia de que compartir información abiertamente mejora el flujo de valor de negocio. Utilizar un lenguaje claro y directo es parte del valor.
 - **Equilibrio:** El entendimiento de que los diferentes aspectos, puntos de vista y capacidades deben ser equilibrados para conseguir efectividad. Algunos aspectos (como demanda y capacidad) causarán colapso si no se encuentran equilibrados por un periodo prolongado.
 - **Colaboración:** Trabajar juntos. El Método Kanban fue formulado para mejorar la manera en que las personas trabajan juntas, por ello, la colaboración está en su corazón.

- **Foco en el cliente:** Conociendo el objetivo para el sistema. Cada sistema kanban fluye a un punto de valor realizable — cuando los clientes reciben un elemento solicitado o servicio. Los clientes en este contexto son externos al servicio, pero pueden ser internos o externos a la organización como un todo. Los clientes y el valor que estos reciben es el foco natural en Kanban.
- **Flujo:** La realización de ese trabajo es el flujo de valor, tanto si es continuo como puntual. Ver el flujo es un punto de partida esencial en el uso de Kanban.
- **Liderazgo:** La habilidad de inspirar a otros a la acción a través del ejemplo, de las palabras y la reflexión. Muchas organizaciones tienen diferentes grados de jerarquía estructural, pero en Kanban, el liderazgo es necesario a todos los niveles para alcanzar la entrega de valor y la mejora.
- **Entendimiento:** Principalmente conocimiento de sí mismo (tanto individual como de la organización) para ir hacia adelante. Kanban es un método de mejora, por lo que conocer el punto de inicio es la base de todo.
- **Acuerdo:** El compromiso de avanzar juntos hacia los objetivos, respetando — y donde sea posible, acomodando — las diferencias de opinión o aproximaciones. Esto no es gestión por consenso sino un compromiso dinámico para mejorar.
- **Respeto:** Valorando, entendiendo y mostrando consideración por las personas. De manera apropiada al pie de esta lista se encuentra la base sobre la cual reposan el resto de los valores.

Valores de Kanban

1. **Transparencia:** Comunicación clara y abierta del estado del trabajo.
2. **Equilibrio:** Mantener balance entre demanda y capacidad del equipo.
3. **Colaboración:** Trabajar juntos para mejorar continuamente.
4. **Foco en el cliente:** Entregar valor real a quien recibe el producto o servicio.
5. **Flujo:** Priorizar la continuidad y la eliminación de bloqueos en el proceso.
6. **Liderazgo:** Inspirar y guiar a todos los niveles del equipo.
7. **Entendimiento:** Conocer el punto de partida y el contexto organizacional.
8. **Acuerdo:** Avanzar juntos, respetando diferencias.
9. **Respeto:** Base de todos los demás valores; reconocer y valorar a las personas.

Principios directores

1. **Sostenibilidad:** relativo a encontrar un foco sostenible y un foco en la mejora.
2. **Orientación al servicio:** enfocado a conseguir rendimiento y satisfacción del cliente.
3. **Supervivencia:** relativa al mantenimiento de la competitividad y adaptabilidad.

Principios fundacionales

Hay seis principios fundamentales de Kanban, los cuales pueden ser divididos en dos grupos: los principios de gestión de cambio y los principios de entrega o despliegue de servicios (Fig 3) Principios de gestión de cambio.

Cada organización es una red de individuos, conectados psicológicamente y socio- lógicamente para resistir al cambio. Kanban reconoce estos aspectos humanos con tres principios de gestión de cambios.

1. Empezar con lo que estés haciendo ahora: Entender los procesos actuales tal y como están siendo realizados en la actualidad, y Respetar los roles actuales, las responsabilidades de cada persona y los puestos de trabajo
2. Acordar en buscar la mejora a través del cambio evolutivo.
3. Fomentar el liderazgo en cada nivel de la organización desde las contribuciones individuales de cada persona hasta las posiciones más senior de la organización.

Hay dos razones clave para que “empezar desde donde estés” sea una buena idea. La primera es que se minimiza la resistencia al cambio ya que respetamos las prácticas actuales y a las personas que las llevan a cabo. Esto es crucial para involucrar a todos en los retos y desafíos del futuro. La segunda es que los procesos actuales, junto con sus deficiencias obvias, contienen la sabiduría y la potencialidad de mejora que incluso las personas que trabajan con ellos no aprecian en su totalidad. Dado que el cambio es esencial, no hay que imponer soluciones desde diferentes contextos, sino buscar la mejora evolutiva en todos los niveles de la organización.

A partir de las prácticas actuales hay que establecer los objetivos de mejora desde la situación actual a la situación

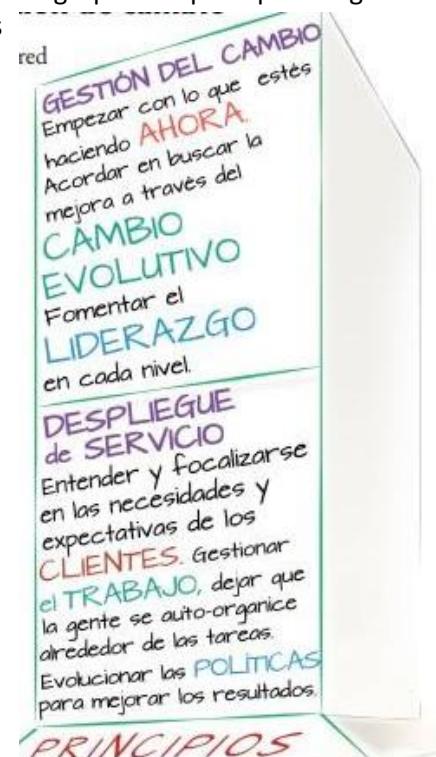


Figure 3 Principios de Kanban

deseada y evaluar las mejoras y como éstas nos van acercando hacia el objetivo.

Principios de despliegue de servicios Las organizaciones, independientemente de su tamaño, son ecosistemas de servicios interdependientes. Kanban reconoce esto con tres principios de despliegue de servicios, aplicables no solo a un servicio, sino a todos ellos

1. Entender las necesidades y en ellas. expectativas de tus clientes y focalizarse en ellas
2. Gestionar el trabajo: dejar que la gente se auto-organice alrededor de las tareas.
3. Evolucionar las políticas para mejorar los resultados hacia el cliente y del negocio

Estos principios están muy alineados con el principio director de orientación a servicios y entrega de valor al cliente. Cuando el trabajo en mismo y el flujo de valor al cliente no es claramente visible, las organizaciones se enfocan en vez de eso en lo que es visible: la gente trabajando en el servicio. ¿Están siempre ocupados? ¿Tienen las suficientes habilidades? ¿Podrían trabajar más duro? El cliente y el valor al cliente reciben menos atención. Estos principios hacen hincapié en que el foco debe estar en los consumidores del servicio y en el valor que reciben del mismo

Principios Fundacionales de Kanban Estos principios se dividen en **dos grupos**:

1. Principios de Gestión del Cambio

Kanban reconoce que las organizaciones son redes de personas que naturalmente **resisten el cambio**.

Por eso promueve una **transformación evolutiva**, basada en tres principios:

1. **Empezar con lo que se está haciendo ahora:**
 - Comprender los procesos actuales tal como son.
 - Respetar los roles, responsabilidades y estructuras existentes.
 - Esto reduce la resistencia al cambio y aprovecha la experiencia acumulada del equipo.
2. **Acordar buscar la mejora mediante el cambio evolutivo:**
 - Las mejoras deben surgir gradualmente, a partir del sistema actual.
 - Se busca pasar de la situación actual a la deseada evaluando cada avance.
3. **Fomentar el liderazgo en todos los niveles:**
 - Cualquier miembro del equipo puede ser líder a través del ejemplo, la colaboración y la mejora continua.
 - El liderazgo no depende del cargo, sino de la iniciativa para generar valor.

2. Principios de Despliegue de Servicios

Las organizaciones funcionan como **ecosistemas de servicios interdependientes**.

Estos principios promueven una gestión centrada en el **valor entregado al cliente**:

1. **Entender las necesidades y expectativas del cliente:**
 - El foco está en **comprender lo que realmente aporta valor** para los usuarios y consumidores del servicio.
2. **Gestionar el trabajo, no a las personas:**
 - Se permite que los equipos **se autoorganicen** alrededor de las tareas.
 - El énfasis está en cómo fluye el trabajo, no en controlar la actividad individual.
3. **Evolucionar las políticas para mejorar los resultados:**
 - Las políticas y reglas de trabajo deben revisarse y ajustarse continuamente.
 - El objetivo es mejorar tanto los **resultados hacia el cliente** como los **resultados del negocio**.

□ Estos principios mantienen el enfoque en el flujo de valor, evitando centrarse únicamente en la ocupación o esfuerzo de las personas.

Anexo

Diferencias entre Scrum y Kanban

Diferencia	SCRUM	KANBAN
Tiempo	Las iteraciones son de tiempo fijo.	El tiempo fijo es opcional. La cadencia puede variar. Pueden estar marcadas por la previsión de los eventos en lugar de tener un tiempo prefijado.
Compromiso	El equipo asume un compromiso de trabajo por iteración.	El compromiso es opcional.
Métrica por defecto	Para la planificación y mejora es la velocidad .	El por defecto es Lead Time (tiempo de entrega promedio)
Equipos	Multifuncionales	Multifuncionales o especializados.
Diagramas de seguimiento	Deben emplearse gráficos Burndown	No se prescriben diagramas de seguimiento.
Limitación WIP	Es indirecta y está marcada por el sprint.	Es directa para cada etapa de trabajo o el estado del trabajo.

Agregar trabajo.	No se puede agregar alcance una vez comenzado el sprint.	Siempre que haya capacidad disponible, se puede agregar trabajo.
Estimaciones.	Se deben realizar estimaciones.	La estimación es opcional.
Compartido entre equipos	Cada equipo tiene su sprint backlog.	Todos los equipos comparten la misma pizarra o tabla kanban.
Permanencia tablero	En cada sprint se limpia.	El tablero es persistente.
Roles	Se prescriben tres roles. PO, SM y Equipo.	No se prescriben roles.
Priorización	El product backlog debe estar priorizado.	La priorización es opcional.
Funcionalidades	Espera que las funcionalidades se dividan tal que se completen en un sprint.	No se prescribe el tamaño de la funcionalidad.

Kanban no tiene el concepto de iteración ni proyecto, porque el trabajo fluye a lo largo de las distintas etapas del proceso de manera continua. No se corta o para en ningún momento este proceso.

El tablero en Kanban es permanente, mientras que, en Scrum, el mismo se limpia al finalizar cada sprint, ya que contiene las columnas de “to-do”, “doing” y “done” propio del sprint que se está ejecutando.

Ser ágil y hacer ágil

Hacer ágil

Cuando implementamos métodos y prácticas ágiles estamos “haciendo” algo para lograr la agilidad. Esto nos ayuda a tener los eventos artefactos y procesos adecuados para entregar valor de forma continua con un buen nivel de calidad, gracias a esto podemos tener algunos beneficios como: adaptación a los cambios, mejorar la visibilidad, incrementar la productividad, mejorar la calidad y reducir los riesgos. Un buen programa de entrenamiento nos puede ayudar a alcanzar este estado.

Doing Agile (hacer ágil) es practicar rituales y ceremonias ágiles. Las reuniones de pie, las demostraciones y revisiones quincenales, la planificación del sprint, los tableros Kanban y los puntos de la historia. Todos estos elementos son indicaciones claras de “Doing Agile”.

Pero simplemente realizar algunas prácticas que se reconocen como Agile no se traducen en un entorno de trabajo ágil, en equipos de alto rendimiento, en mejor calidad o en un tiempo de comercialización más rápido.

Ser ágil

Cuando además de implementar prácticas adoptamos una forma diferente de pensar estamos “siendo” alguien distinto, alguien que vive la agilidad y cuyo comportamiento está alineado a los valores y principios ágiles, esto nos da beneficios adicionales como: deleite del cliente, placer por el trabajo, compromiso, innovación, creatividad y aprendizaje continuo. Cuando hablamos de ser ágil a nivel organizacional nos referimos a empresas que adoptan una cultura que permite obtener estos beneficios mediante un cambio de mentalidad en todos los niveles de liderazgo, no solo de los equipos de trabajo.

Ser ágil (Being Agile) realmente funciona de manera diferente. Incluye una estructura de toma de decisiones drásticamente nueva, participación empresarial totalmente integrada, equipos persistentes autoorganizados y multidisciplinares.

Además, el enfoque es en el producto (no proyecto), usando [DevOps](#) y una cultura totalmente pensada en el aprendizaje continuo.

En definitiva, significa convertir esos rituales en hábitos diarios que impulsan la acción de mejora continua. Esto es más que hacer agilidad, ser ágil es lo que es.

A medida que la transformación se expande y madura, deja de hacer agilidad, ser ágil es en lo que se transforma. Hay varias prácticas que pueden emplearse para ayudar a las organizaciones a mejorar:

- Confía en tus equipos. Direccionalmente, el liderazgo senior establece los objetivos estratégicos, pero los equipos autoorganizados y auto empoderados (liderados por Products Owners autónomos y autorizados) determinarán los mejores enfoques, arquitecturas técnicas y todo el “cómo” a desarrollar, probar y lanzar. Esto significa que no hay más comités directivos. Los problemas se resuelven en el nivel más bajo posible (nivel de equipo).

- Practica la transparencia. Esto se aplica a todo en lo que trabaja el equipo. La transparencia debe ser un mantra para todos los equipos ágiles, así como para su liderazgo. Todos los entregables son trabajos en progreso, por lo tanto, deben estar abiertos para inspección en todo momento. La clave es que los Product Owners establezcan expectativas con el liderazgo superior sobre la naturaleza de esos entregables y cómo se entregan. La mejor medida de progreso es el incremento de trabajo disponible para los usuarios.
- Gestionar los indicadores de progreso de manera diferente. Dejamos de administrar por informes de estado y eliminamos los mandatos basados en fechas. Podemos seleccionar objetivos de lanzamiento de productos e hitos, pero no serán de alcance fijo; serán de naturaleza más temática sin características específicas completamente definidas. Los radiadores de información son increíblemente importantes. La transparencia es un principio básico ágil y, como tal, debe integrarse en cada transformación.

Para aquellos que son habitualmente ágiles, ver errores rápidamente es una prueba de que estás aprendiendo, mejorando constantemente y acercándote a tus clientes.

Aquellos que piensan en una mentalidad ágil ponen la confianza y la transparencia por encima de todo en su cultura. Entonces, mírate en el espejo y pregúntate si eres realmente ágil, o si simplemente estás haciendo posturero. Porqué puede ser que no te valga solo hacer agilidad, ser ágil es lo que tienes que ser.

Conclusión

En mi experiencia las mejores transiciones hacia la agilidad se logran cuando además de la adopción de prácticas ágiles también se adopta el mindset Ágil, esto definitivamente es mucho más complejo, ya que un entrenamiento no será suficiente, se necesita de un acompañamiento adecuado, de preferencia de un coach experimentado que pueda ayudar a lograr una transición cultural.

Hacer ágil es definitivamente distinto a ser ágil, y aunque ninguno de los dos es fácil no debemos perder de vista los objetivos que queremos alcanzar, si queremos el mayor de los beneficios debemos desafiar nuestras actuales formas de pensar y promover ese cambio alrededor de nosotros. Mi recomendación es solicitar apoyo de un experto, ya sea interno o externo, no solo en metodologías Ágiles sino también en el proceso de transformación hacia nuevos paradigmas.

Cuadro comparativo Gestión Tradicional vs. Gestión ágil

Aspecto a comparar	Gestión Tradicional	Gestión Ágil
Respuesta a cambios	Poco flexible	Altamente flexible
Incertidumbre en el desarrollo del proyecto	Baja → se elimina en etapa de requerimientos	Alta → se elimina cuando sea necesario
Tiempos de entrega de software útil	Tardío	Temprano
Retroalimentación del cliente para incluir nuevas necesidades o correcciones	Baja	Alta
Participaciones en estimaciones	Las hace el líder de proyecto	Las hace el equipo de desarrolladores
Conformación de equipo	Jerárquico, con roles definidos.	Autogestionado según fortalezas de cada miembro
Proceso de desarrollo	Definido	Empíricos
Ciclos de vida admitidos	Cualquiera	Iterativos e incrementales
Se estima	Recursos y tiempos necesarios para el desarrollo	Alcances a implementar en iteraciones (Tamaño)
Son constantes	Los requisitos identificados en etapa de requerimientos	El equipo, recursos y tiempo de trabajo.
Planificación de tareas	Estimadas con detalle al planificar el proyecto	Estimadas con detalle al planificar la iteración

Preguntas frecuentes de la ingeniería de software (Sommerville)

Pregunta	Respuesta
¿Qué es software?	Programas de cómputo y documentación asociada. Los productos de software se desarrollan para un cliente en particular o para un mercado en general.
¿Cuáles son los atributos del buen software?	El buen software debe entregar al usuario la funcionalidad y el desempeño requeridos, y debe ser sustentable, confiable y utilizable.
¿Qué es ingeniería de software?	La ingeniería de software es una disciplina de la ingeniería que se interesa por todos los aspectos de la producción de software.
¿Cuáles son las actividades fundamentales de la ingeniería de software?	Especificación, desarrollo, validación y evolución del software.
¿Cuáles son los principales retos que enfrenta la ingeniería de software?	Se enfrentan con una diversidad creciente, demandas por tiempos de distribución limitados y desarrollo de software confiable.
¿Cuáles son los mejores métodos y técnicas de la ingeniería de software?	Aun cuando todos los proyectos de software deben gestionarse y desarrollarse de manera profesional, existen diferentes técnicas que son adecuadas para distintos tipos de sistema. Por ejemplo, los juegos siempre deben diseñarse usando una serie de prototipos, mientras que los sistemas críticos de controles de seguridad requieren de una especificación completa y analizable para su desarrollo. Por lo tanto, no puede decirse que un método sea mejor que otro.

Cuadro comparativo Auditorías vs. Revisiones técnicas

Aspecto	Auditoría	Revisión
Para qué sirven	Para revelar qué se está haciendo en la realidad, contrastado con lo que se comprometió a hacer.	Encontrar errores en un artefacto lo antes posible, para evitar su propagación.
Roles	Auditor, auditado y gerente de calidad	Autor, moderador, anotador, lector, inspector
Beneficios	Opiniones independientes, identificar áreas de insatisfacción potencial para el cliente, asegurar que se cumplen expectativas, dar visibilidad a procesos de trabajo, etc.	Detección de errores, evitar propagación de defectos, reducción de retrabajo (costos), aumentar calidad de un artefacto, etc.
Etapas	<ol style="list-style-type: none"> 1. Planificación 2. Ejecución 3. Análisis y Reporte de resultados 4. Seguimiento 	<p>En revisiones informales no hay pasos a seguir.</p> <p>En revisiones formales:</p> <ol style="list-style-type: none"> 1. Planificación 2. Visión general 3. Preparación 4. Reunión de Inspección 5. Corrección 6. Seguimiento
Tipos	De proyecto, de calidad, de producto (física y funcional)	Formales e informales
Hallazgos	Buenas prácticas, observaciones y desviaciones	Errores

Cuadro comparativo CMMI por Etapas y Continuo

Aspecto	Por etapas	Continuo
Características	Acreditan determinado nivel de madurez de una organización como un todo, dependiendo de qué tantas prácticas realicen, de acuerdo a lo especificado en el modelo CMMI.	Acreditan determinado nivel de madurez de un área de proceso de una organización, dependiendo de qué tantas prácticas realicen concernientes a esa área, de acuerdo a lo que la organización quiera especializar.
Ventajas	Permiten integrar las formas de trabajo de toda la organización. Garantiza cierto nivel de calidad en toda la organización, según el nivel que se acredite. Permite sentar bases en la organ y aumentar los niveles de calidad en toda la empr.	Permite a una organización especializarse en un área de proceso, independientemente del resto de áreas. La organización elige qué prácticas realizar, de acuerdo al área en la que se especialice.
Desventajas	Se deben realizar ciertas actividades en áreas de proceso que quizás no sean relevantes para la empresa.	Pueden quedar áreas de proceso sin demasiada calidad, debido a que se centran los esfuerzos en unas pocas. La organización como un todo se sigue considerando inmadura, si es que no implementa las actividades que plantea CMMI nivel 1 en las áreas de proceso.
Diferencias	Niveles: 5 → 1 a 5 Niveles: indican madurez organizacional Definidos por un conjunto de áreas de proceso Provee una secuencia para mejorar determinadas áreas de proceso progresivamente	Niveles: 6 → 0 a 5 Niveles: indican capacidad de un área de proceso Definidos por cada área de proceso Permite mejorar en un área de proceso que se desee

Respuesta a pregunta de parcial “Realice un análisis de los principios Lean y explique las prácticas que propone Kanban para aplicar en forma concreta los principios.”

- Eliminar desperdicios
 - Visualizar el trabajo (permite visualizar el flujo de trabajo y exponer aquellas actividades que no aportan valor y aquellas que generan tiempos de espera que “trablan” el flujo de trabajo)
 - Limitar el WIP (el trabajo a medias genera retrabajo, lo cual es un desperdicio. Al limitar el WIP, se evita ese trabajo a medias para concentrar esfuerzos)
 - Amplificar el aprendizaje
 - Hacer explícitas las políticas (que todos los miembros tengan acceso a las políticas usadas para cada unidad de trabajo, actividades a realizar, información necesaria del dominio, etc.)
 - Evolución y mejora continua (no introducir cambios grandes, sino que visualizar el proceso de trabajo actual y aplicar mejoras en él)
 - Embeber la integridad conceptual
 - Postergar decisiones hasta el momento responsable
 - Limitar el WIP (particularmente el WIP de la columna backlog)
 - Ver el todo
 - Visualizar el trabajo (a través del tablero permite que todos los miembros observen todo el trabajo que se hace, el sentido de cada una de las tareas y como estas conforman el flujo de trabajo como un todo que desencadena en la satisfacción de los clientes)
 - Dar poder al equipo
 - Haciendo explícitas las políticas y conformando un sistema de trabajo pull, donde cada miembro toma el trabajo cuando esté en condiciones de realizarlo, y no se lo impone nadie.
 - Mejorar colaborativamente
 - Entregar lo antes posible
- Gestionar el flujo de trabajo (permite ver de principio a fin los pasos que se deben seguir para lograr cumplir con las necesidades del cliente y hacer aquellos ajustes que sean necesarios para entregar el