UNIDAD 4 – METODOLOGÍAS ÁGILES

Software

Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación

Ingeniería de Software

Aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software.

Sistemático -> normas, procedimientos
Disciplinado -> Orden y subordinación entre los miembros
Cuantificable -> Medible

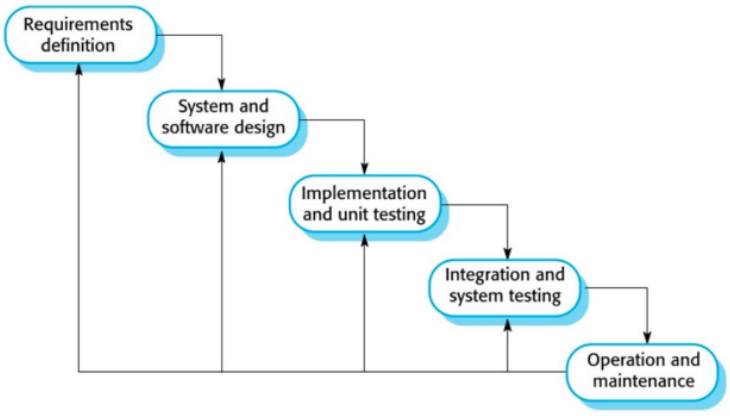
Proceso de Software

Conjunto de actividades que conducen a la elaboración de un producto de software. Requiere la definición de roles y la especificación de las actividades.

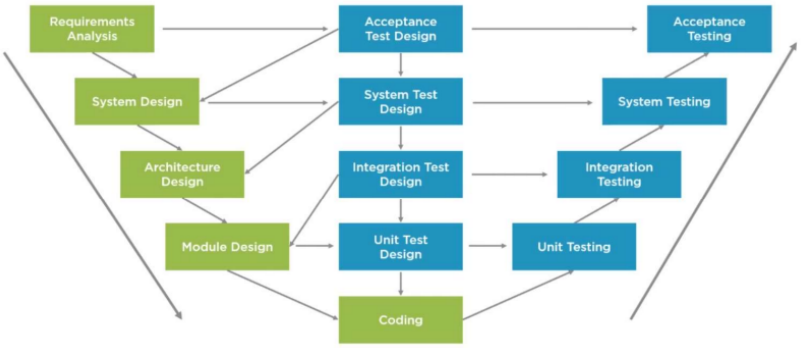
Modelo de Proceso de Software

Representación simplificada del proceso de software que puede utilizarse como base para crear procesos de software específicos.

Modelo cascada



Modelo en V



El costo de arreglar errores se incrementa con el paso del tiempo. Desde los requerimientos hasta el desarrollo y más aún en producción.

Modelos Secuenciales

Se usan generalmente cuando los requerimientos se entienden por completo, o el software no cambiará (sistema embebido). Puede presentar *problemas*: ser inflexible, se hace difícil responder al cambio de los requerimientos y es poco probable que se construya lo que el cliente realmente necesita.

Plan guiado y el Desarrollo ágil

El desarrollo guiado por un plan y el desarrollo ágil son dos enfoques diferentes para la gestión y ejecución de proyectos de desarrollo de software, el *desarrollo guiado por un plan* sigue un enfoque más rígido y secuencial,

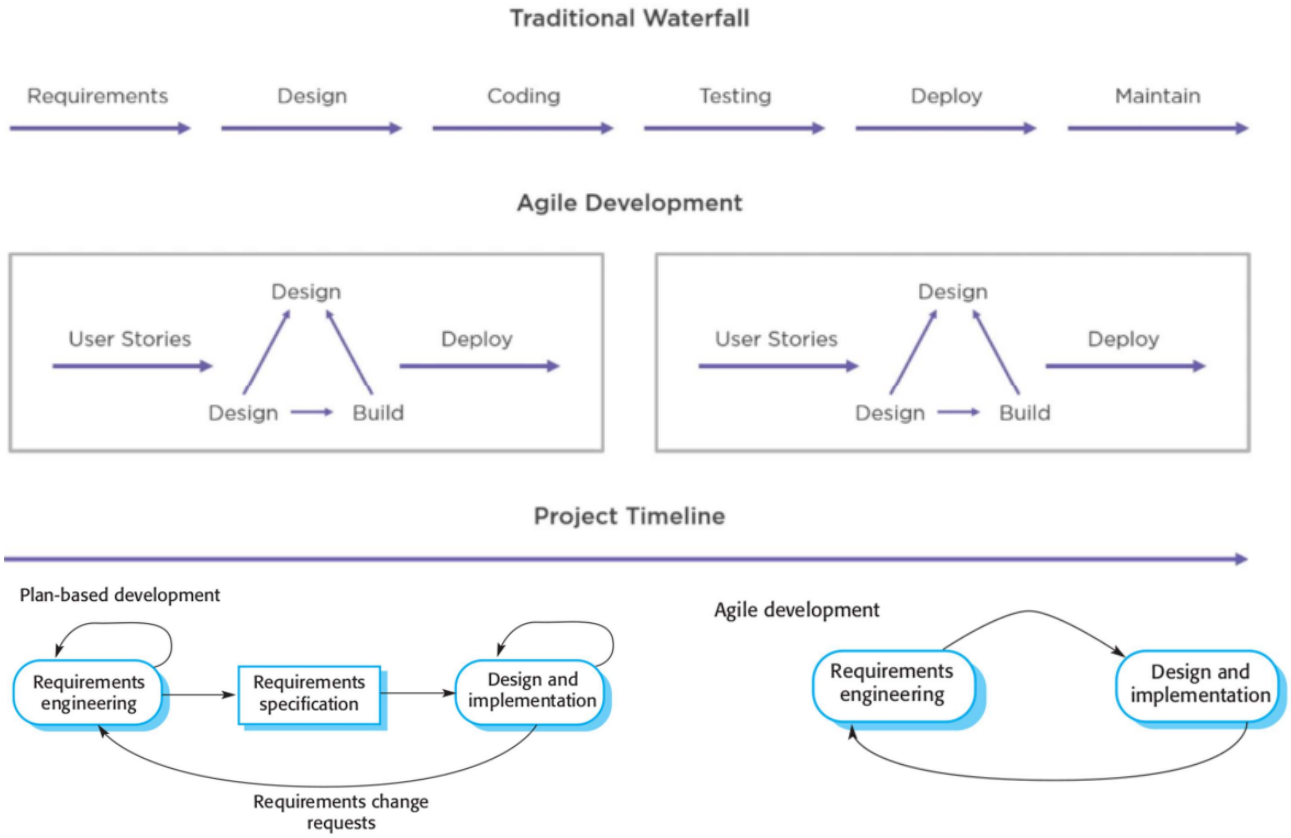
mientras que el *desarrollo ágil* se basa en la flexibilidad, la adaptación continua y la colaboración constante con los interesados..

Desarrollo Guiado por Plan:

- *Etapas bien definidas:* El proyecto se divide en etapas claramente definidas, como análisis, diseño, implementación y pruebas. Cada etapa tiene sus objetivos y actividades específicas.
- *Entregables por etapas bien definidos*: Cada etapa del proyecto tiene entregables claramente definidos que deben completarse antes de pasar a la siguiente etapa. Esto a menudo implica una revisión y aprobación formal antes de avanzar.
- *Entradas de una etapa es la salida de la etapa anterior:* Las salidas de una etapa se utilizan como entradas para la siguiente etapa. Esto asegura una secuencia lineal y predecible en el proceso de desarrollo.
- *Las iteraciones se producen dentro de las actividades:* Si bien puede haber iteraciones en cada etapa, estas se limitan a actividades específicas dentro de esa etapa.
- *Inflexible*: Tiende a ser menos flexible ante los cambios en los requisitos o las prioridades, ya que está basado en un plan predefinido. Los cambios significativos pueden requerir una reevaluación del plan y un proceso de cambio formal.

Desarrollo Ágil:

- *Especificación, diseño, implementación y pruebas son intercalados:* No se espera que todas las especificaciones se completen antes de la implementación. Estas actividades se realizan de manera intercalada y se adaptan según las necesidades.
- *Salidas del proceso decididas mediante negociación durante el proceso de desarrollo de software (feedback):* Los detalles y las prioridades pueden cambiar durante el proceso, y las decisiones se toman mediante la comunicación constante con los interesados.
- *Iterativo e incremental:* Se basa en ciclos cortos de desarrollo llamados iteraciones, donde se entregan partes funcionales del software en cada ciclo.
- *Responde al cambio:* La flexibilidad es fundamental. Los cambios en los requisitos se pueden abordar de manera más fluida, y el software se adapta a medida que se obtiene más información y feedback.



Manifiesto Ágil

Se valoran:

- Individuos e interacciones sobre procesos y herramientas
- Software funcionando sobre la documentación comprensible
- Colaboración con el cliente sobre negociación de contratos
- Responder al cambio sobre seguir el plan

Principios del manifiesto ágil

Son 12 principios separados en 3 grupos:

1. **Entrega del software en intervalos regulares y frecuentes**

La máxima prioridad es satisfacer al cliente a través de la entrega temprana y continua de software valioso. Entregar software funcional con frecuencia, desde un par de semanas hasta un par de meses, con preferencia por el plazo más corto. El software funcional es la medida principal del progreso. Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deberían ser capaces de mantener un ritmo constante indefinidamente.

2. **Comunicación en el equipo**

- Los profesionales de negocios y los desarrolladores deben trabajar juntos a diario a lo largo del proyecto.
- El método más eficiente y efectivo para transmitir información a un equipo de desarrollo y dentro de él es la conversación cara a cara.
- Las mejores arquitecturas, requisitos y diseños surgen de equipos autoorganizados.
- Construye proyectos alrededor de individuos motivados. Hay que brindarles el entorno y el apoyo que necesitan y confía en ellos para que hagan el trabajo.
- En intervalos regulares, el equipo reflexiona sobre cómo volverse más efectivo y luego ajusta y adapta su comportamiento en consecuencia.

3. **Excelencia en el diseño**

- La atención continua a la excelencia técnica y al buen diseño mejora la agilidad.
- La simplicidad, el arte de maximizar la cantidad de trabajo que no se hace, es esencial.
- Los procesos ágiles aprovechan el cambio en beneficio de la ventaja competitiva del cliente.

Roles en los temas ágiles

- Los equipos ágiles son principalmente equipos de desarrollo de software y miembros de un departamento en segundo lugar.
- Un equipo debería contar con un experto en el dominio del producto.
- Un equipo cuenta con miembros que poseen habilidades multidisciplinarias.
- Un equipo debería contar con algún tipo de rol de liderazgo.
- Un equipo puede beneficiarse de un coach o mentor ágil.

Ideas equivocadas sobre Agile

- Significa “sin compromiso”
- El desarrollo ágil no es predecible
- Es la solución mágica
- Solo hay una forma de hacer Agile
- No necesita diseño previo

- No causa problemas
- Estamos haciendo scrum, entonces no necesitamos programar en parejas, refactorizar o el desarrollo guiado por pruebas (TDD).

Errores que se comenten en los team agile nuevos

Pruebas deficientes: Descuidar las pruebas adecuadas puede llevar a problemas de calidad y retrasos en el desarrollo. Las pruebas son esenciales en el enfoque ágil para garantizar que el software entregado sea de alta calidad.

Ignorar la retroalimentación del cliente: En Agile, la retroalimentación del cliente es fundamental para ajustar y mejorar continuamente el producto. Ignorar esta retroalimentación puede resultar en un producto que no satisface las necesidades del cliente.

Falta de empoderamiento del equipo: En Agile, se espera que los equipos tomen decisiones y sean autoorganizados. La falta de empoderamiento puede obstaculizar la agilidad y la capacidad del equipo para tomar decisiones efectivas.

Falta de reuniones de retrospectiva y demostración: Las reuniones de retrospectiva son cruciales para identificar mejoras y ajustes en el proceso, mientras que las reuniones de demostración muestran el progreso del trabajo al cliente. La falta de estas reuniones puede dificultar la adaptación y la colaboración.

No tener un plan para abordar la resistencia de los empleados: La adopción de metodologías ágiles puede enfrentar resistencia por parte de algunos miembros del equipo. No tener un plan para abordar esta resistencia puede dificultar la transición a un enfoque ágil.

Es importante que los equipos ágiles nuevos reconozcan estos errores potenciales y trabajen en superarlos para tener éxito en su implementación de Agile. La comunicación efectiva, la formación y la adaptación constante son clave para evitar estos problemas.

Ventajas de Agile

Retorno temprano de la inversión (ROI): La entrega temprana y continua de funcionalidades permite a las empresas obtener un ROI más rápido, ya que los productos o características se entregan en ciclos más cortos, lo que puede generar valor más rápidamente.

Retroalimentación de clientes reales: El enfoque ágil se basa en la colaboración continua con los clientes. Esto permite obtener retroalimentación directa de los usuarios finales durante todo el proceso de desarrollo, lo que garantiza que el producto se alinee con las necesidades del cliente y se ajuste en consecuencia.

Construcción de los productos correctos: Agile pone un fuerte énfasis en la priorización y entrega de las características más valiosas y necesarias. Esto evita la construcción de características innecesarias o no solicitadas, lo que ahorra tiempo y recursos.

Entrega continua de mejor calidad: Los principios de Agile promueven la calidad desde el principio y alientan prácticas como el desarrollo guiado por pruebas (TDD) y la refactorización. Esto conduce a una mayor calidad del producto y la capacidad de mejorar continuamente la calidad a lo largo del tiempo.

En resumen, Agile ofrece un enfoque que se centra en la entrega temprana y continua de valor, la retroalimentación directa del cliente y la mejora constante de la calidad, lo que puede beneficiar tanto a las empresas como a los equipos de desarrollo.

Desventajas de Agile

- Difícil evaluar el esfuerzo requerido al comienzo del ciclo de desarrollo de software,
- Puede ser muy exigente en términos de tiempo para los usuarios
- Hay riesgo de que los requisitos o funcionalidades del proyecto se expandan o cambien de manera no planificada a lo largo del tiempo.
- Puede ser un desafío para nuevos miembros en el equipo.
- Los costos pueden aumentar en un entorno ágil en el que se requieren pruebas de manera continua a lo largo del proyecto en lugar de esperar hasta el final del desarrollo. Esto se debe a que en un enfoque ágil, se fomenta la realización de pruebas a lo largo del ciclo de desarrollo, lo que implica la participación de probadores de forma continua en lugar de en una fase posterior del proyecto.
- Puede ser intenso para el equipo.

Problemas con métodos ágiles

Puede ser un desafío mantener el compromiso y el interés continuo de los clientes a lo largo de un proyecto ágil. Los clientes pueden perder interés o compromiso si no ven resultados inmediatos o si el proyecto se extiende en el tiempo. La intensa participación y colaboración requerida en los métodos ágiles pueden ser abrumadoras para algunos miembros del equipo. No todos los miembros del equipo pueden estar preparados para la autogestión y la toma de decisiones activa.

En proyectos con múltiples partes interesadas, puede ser complicado priorizar los cambios, ya que diferentes partes interesadas pueden tener necesidades y objetivos diferentes. Esto puede llevar a conflictos y desafíos en la toma de decisiones. Si bien la simplicidad es un principio importante en Agile, mantenerla puede requerir esfuerzos adicionales. A veces, la simplicidad se ve amenazada por cambios constantes o por la inclusión de características innecesarias. En algunos casos, los contratos y las estructuras legales pueden no ser compatibles con los enfoques ágiles, lo que puede generar problemas y desafíos en la gestión de proyectos.

Principios métodos ágiles (resumidos)

Principio	Descripción
Participación de los Clientes	Los clientes deberían participar activamente en todo el proceso de desarrollo. Su función es proporcionar y dar prioridad a los nuevos requisitos del sistema y evaluar las iteraciones de la sistema
Entrega Incremental	El software se desarrolla en incrementos con el cliente especificando los requisitos que deben incluirse en cada incremento
Las personas no procesos	Tienen que reconocerse y aprovecharse las habilidades del equipo de desarrollo. Debe permitirse a los miembros del equipo desarrollar sus propias formas de trabajar sin procesos establecidos.
Aceptar el cambio	Esperara cambios en los requerimientos del sistema y diseñar el sistema para adaptarse a estos cambios
Mantenga la simplicidad *Ingeniería de Software III*	Enfoquese en la simplicidad tanto del software siendo desarrollado como el proceso de software. Siempre que sea posible, trabajar en actividades para eliminar la complejidad del sistema Capítulo 3 Desarrollo ágil de software 39

Aplicabilidad del método ágil

Los métodos ágiles se adaptan mejor a ciertos contextos de desarrollo de software:

Desarrollo de productos para la venta: Los métodos ágiles son adecuados para el desarrollo de productos que se venden en el mercado. Estos productos suelen ser de tamaño pequeño o mediano y pueden beneficiarse de la entrega temprana de características y la adaptación rápida a las necesidades cambiantes del mercado.

Desarrollo de sistemas a medida internos: Los métodos ágiles también pueden ser eficaces en el desarrollo de sistemas internos en organizaciones donde hay un compromiso claro por parte del cliente para participar activamente. En entornos con menos regulaciones externas y más flexibilidad, los métodos ágiles pueden brindar ventajas en términos de adaptabilidad y satisfacción del cliente.

Desafíos en sistemas grandes: Aunque los métodos ágiles se pueden aplicar en proyectos grandes, surgen desafíos en términos de coordinación, comunicación y escalabilidad. Mantener la agilidad en sistemas grandes puede ser más complicado debido a la necesidad de gestionar múltiples equipos y la complejidad que implica.

En resumen, los métodos ágiles son altamente efectivos en proyectos pequeños o medianos y en contextos donde la colaboración activa del cliente es factible. Sin embargo, en sistemas grandes, es fundamental abordar los desafíos de escala y coordinación para aplicar con éxito métodos ágiles. Cada contexto requiere una evaluación cuidadosa y la adaptación de las prácticas ágiles según las necesidades específicas del proyecto.

Cuestiones técnicas, humanas y organizacionales

La mayoría de los proyectos incluyen elementos del plan guiado y procesos ágil. La decisión sobre el equilibrio depende de:

¿Qué tipo de sistema se está desarrollando?
Los enfoques del plan guiado pueden ser necesarios para los sistemas que requieren una gran cantidad de análisis antes de la aplicación

¿Cuál es la expectativa de vida del sistema?
Los sistemas de larga vida tal vez requieran más documentación de diseño para comunicar las intenciones originales de los desarrolladores del sistema al equipo de soporte.

¿Qué tecnologías están disponibles para apoyar el desarrollo del sistema?

Los métodos ágiles se basan en buenas herramientas para realizar un seguimiento de la evolución de un diseño

¿Cómo se organiza el equipo de desarrollo?
Si el equipo de desarrollo se distribuye o si parte del desarrollo se subcontrata, entonces puede que tenga que desarrollar los documentos de diseño para comunicarse a través de los equipos de desarrollo.

¿Hay cuestiones culturales o de organización que puedan afectar al desarrollo del sistema?
Las organizaciones tradicionales de ingeniería tienen una cultura basada en el plan de desarrollo, ya que esta es la norma en la ingeniería.

¿Qué tan buenos son los diseñadores y programadores del equipo de desarrollo?
A veces se argumenta que los métodos ágiles requieren niveles más altos de capacitación que los enfoques basados en planes, en el que los programadores simplemente traducen un diseño detallado en código.

¿El sistema está sujeto a regulación externa?
Si un sistema tiene que ser aprobado por un regulador externo (por ejemplo, que el FAA aprueba el software, es crítico para el funcionamiento de una aeronave) entonces usted probablemente tendrá que producir detallada documentación como parte de la justificación de la seguridad del sistema.

PROGRAMACIÓN EXTREMA

Extreme Programming es una metodología de desarrollo de software la cual intenta mejorar la calidad del software y la velocidad de respuesta ante los requerimientos cambiantes del cliente.

Actividades

Escribir código: En XP, escribir código es una actividad fundamental. Los programadores se enfocan en crear código de alta calidad y funcionamiento.

Testear el sistema: XP promueve la escritura de pruebas (tests) para verificar el correcto funcionamiento del software. Esto incluye pruebas unitarias, de integración y de aceptación.

Escuchar a los clientes y los usuarios: La retroalimentación constante de los clientes y usuarios es esencial en XP. Se busca comprender sus necesidades y requerimientos para adaptar el software en consecuencia.

Diseñar para reducir el acoplamiento: La arquitectura del software se diseña de manera que los componentes estén lo menos acoplados posible. Esto facilita la flexibilidad y el cambio en el software.

Valores

La comunicación es esencial: Se promueve una comunicación abierta y constante entre los miembros del equipo de desarrollo, así como con los clientes y usuarios.

Simplicidad: XP favorece soluciones simples y elegantes en lugar de complejas. Se busca evitar la sobrecarga innecesaria.

Aprender del feedback: XP valora la retroalimentación como una oportunidad para aprender y mejorar continuamente el proceso de desarrollo.

Tener coraje: Se anima a los miembros del equipo a tomar decisiones valientes y enfrentar los problemas en lugar de ignorarlos.

Respetar al equipo y al proyecto: Se fomenta el respeto entre los miembros del equipo y hacia el proyecto en sí. Todos contribuyen al éxito del proyecto.

Principios

Feedback rápido: Se busca obtener retroalimentación rápida para identificar problemas y oportunidades de mejora en una etapa temprana del desarrollo.

Construir con simplicidad (DTSTTCPW, KISS, YAGNI): XP aboga por mantener el código simple y evitar la inclusión de características innecesarias. Las siglas se refieren a "Don't Repeat Yourself" (No te repitas), "Keep It Simple, Stupid" (Manténlo simple, tonto) y "You Aren't Gonna Need It" (No lo necesitarás).

Cambio incremental: XP promueve la implementación de cambios pequeños y frecuentes en lugar de cambios masivos y menos frecuentes. Esto facilita la adaptación a las necesidades cambiantes del proyecto.

Prácticas

Estas son algunas de las prácticas clave en XP que ayudan a implementar los valores y principios mencionados. Algunas de ellas incluyen:

Feedback detallado: Implica obtener comentarios específicos y concretos sobre el software, lo que ayuda a identificar áreas de mejora. Esto puede incluir revisiones de código, pruebas de usuario y evaluaciones de calidad para garantizar que el producto cumple con los requisitos y expectativas de los clientes.

Proceso continuo: XP promueve un proceso de desarrollo constante y sin interrupciones prolongadas. Los equipos trabajan en ciclos cortos y regulares para entregar incrementos de software funcional. Esto permite una respuesta ágil a los cambios en los requisitos y una entrega más rápida de valor al cliente.

Entendimiento común: La comunicación efectiva y la comprensión compartida de los objetivos y requisitos del proyecto son esenciales. Todos los miembros del equipo deben tener una visión clara y común del proyecto para colaborar de manera efectiva y alinear sus esfuerzos hacia los mismos objetivos.

Bienestar del programador: Se enfoca en cuidar la salud y el bienestar de los programadores. Esto implica evitar el agotamiento, promover un ambiente de trabajo equilibrado y apoyar a los miembros del equipo en su desarrollo profesional y personal, lo que a su vez mejora la productividad y la calidad del trabajo.

Principio o práctica	Descripción
Planificación Incremental	Los requerimientos se registran en story cards y las stories que se van a incluir en un release se determinan por el tiempo disponible y la prioridad relativa. Los desarrolladores desglosan dichas historias en "tareas" de desarrollo.
Releases pequeños	Al principio se desarrolla el conjunto mínimo de funcionalidad útil, que ofrece valor para el negocio. Las liberaciones del sistema son frecuentes y agregan incrementalmente funcionalidad a la primera liberación.
Diseño Simple	El diseño suficiente se lleva a cabo para satisfacer las necesidades actuales y no más.
Desarrollo pruebas primero	Un marco de pruebas unitarias automatizadas es usado para escribir pruebas para una nueva función antes de que la funcion sea implementada.
Reconstrucción	Se espera de todos los desarrolladores de reconstruir el código continuamente ni bien las mejoras se hallan. Esto mantiene el codigo simple y mantenible.
Programación en parejas	Los desarrolladores trabajan en parejas , revisando el trabajo mutuamente y dandose apoyo para hacer siempre un buen trabajo.
Propiedad colectiva	La pareja de desarrolladores trabaja en todas las áreas del sistema, para que no queden islas de conocimiento desarrolladas y todos los desarrolladores asumen responsabilidad por todo el código.
Integración continua	Tan pronto como el trabajo en una tarea esta completo, es integrado en el sistema. Luego de cada integración, se corren todas las pruebas unitarias.
Ritmo sostenible	Grandes cantidades de horas extras no se consideran aceptables como el efecto neto es a menudo para reducir la calidad del código y mediana productividad
Cliente en el sitio	Un representante de los usuarios finales del sistema (el cliente) debe estar disponible a tiempo completo para el equipo de XP. En un proceso de programación extrema, el cliente es miembro del equipo de desarrollo y es responsable de llevar requisitos del sistema para el equipo de implementación.

User Stories en XP

Requerimientos

Las User Stories son una herramienta fundamental en XP para gestionar los requerimientos de manera ágil y colaborativa. Permiten una comunicación efectiva entre el equipo de desarrollo y el cliente, priorizan las necesidades del usuario y ayudan a tomar decisiones informadas sobre qué construir en cada iteración de desarrollo. Los Requerimientos en XP se manejan mediante el uso de estas:

El **cliente o usuario** es considerado parte del equipo de desarrollo. Esto significa que están involucrados de manera activa y constante en el proceso de desarrollo. Son responsables de tomar decisiones sobre los requisitos y de proporcionar una dirección clara sobre lo que se debe construir.

Las **solicitudes de los usuarios** se expresan en forma de "User Stories" o historias de usuario. Estas son descripciones breves y simples de una funcionalidad o característica deseada desde la perspectiva del usuario. Por lo general, se escriben en un formato que sigue la estructura "Como [tipo de usuario], quiero [realizar una acción] para [lograr un objetivo]".

Todas las User Stories se agregan a un "**backlog**" o lista de tareas pendientes. El backlog se mantiene ordenado por prioridades, lo que significa que las User Stories más importantes o críticas se encuentran en la parte superior de la lista, mientras que las menos importantes se ubican en la parte inferior.

El equipo de desarrollo asigna **estimaciones de costo** a cada User Story. Estas estimaciones ayudan a cuantificar la cantidad de trabajo involucrado en implementar cada historia. Esto es útil para la planificación y programación.

El cliente es el responsable de elegir las User Stories que se incluirán en la próxima versión o release del software. Esta **selección** se basa en las prioridades definidas por el cliente y en las estimaciones de costos proporcionadas por el equipo de desarrollo. Las historias seleccionadas se convierten en el enfoque del trabajo para ese ciclo de desarrollo.

Las "User Stories" (historias de usuario) son parte de un enfoque ágil que ayuda a cambiar el enfoque de escribir sobre los requisitos a hablar de ellos. Son descripciones breves y simples de una funcionalidad, narradas desde la perspectiva de la persona que desea la nueva capacidad, generalmente un usuario o cliente del sistema. Suelen seguir una estructura sencilla, que comienza con "**Como un** <tipo de usuario>, **quiero** <algún objetivo> **para que** <alguna razón>".

Lo más importante es que las User Stories fomentan conversaciones sobre la funcionalidad deseada, lo cual es más relevante que el texto escrito en sí.

Ejemplos:

Un beneficio de las User Stories ágiles es que se pueden escribir a diferentes niveles de detalle. Se pueden escribir historias de usuario para abarcar grandes cantidades de funcionalidad, que generalmente se conocen como "***épicas***". Por ejemplo, una épica para un producto de respaldo de escritorio podría ser:

"Como usuario, puedo respaldar todo mi disco duro".

Dado que una épica generalmente es demasiado grande para que un equipo ágil la complete en una sola iteración, se divide en varias historias de usuario más pequeñas. En este caso, la épica podría dividirse en historias más pequeñas, como:

- "Como usuario avanzado, puedo especificar archivos o carpetas para respaldar según el tamaño del archivo, la fecha de creación y la fecha de modificación"
- "Como usuario, puedo indicar carpetas que no se deben respaldar para que mi unidad de respaldo no se llene con cosas que no necesito guardar".

¿Cómo se agrega detalle a las User Stories?

El detalle se puede agregar a las User Stories de dos maneras: ***dividiendo una historia en varias historias más pequeñas*** o ***agregando "condiciones de satisfacción/éxito"***. Cuando se divide una historia relativamente grande en varias historias más pequeñas, naturalmente se agrega más detalle, ya que se escriben más historias. Las "condiciones de satisfacción/éxito" son simplemente pruebas de aceptación a alto nivel que serán ciertas una vez que la User Story ágil esté completa. Estas condiciones ayudan a definir cuándo se ha cumplido con éxito la historia de usuario.

¿Quién escribe las User Stories?

Cualquier persona puede escribir User Stories. Si bien es responsabilidad del Product Owner asegurarse de que exista un backlog de productos con User Stories ágiles, no significa que el Product Owner sea quien las escriba. A lo largo de un buen proyecto ágil, se espera que cada miembro del equipo escriba ejemplos de historias de usuario. Es importante destacar que quién escribe una User Story es mucho menos importante que quién participa en las discusiones relacionadas con ella. La colaboración y la comunicación efectiva son esenciales en la escritura y definición de las User Stories.

¿Cuándo se escriben las User Stories?

Las User Stories se escriben a lo largo de todo el proyecto ágil. Por lo general, se lleva a cabo un taller de escritura de historias cerca del inicio del proyecto ágil, en el que participa todo el equipo. El objetivo de este taller es crear un backlog de productos que describa completamente la funcionalidad que se agregará a lo largo del proyecto o dentro de un ciclo de lanzamiento de tres a seis meses. Algunas de estas User Stories ágiles serán sin duda "épicas" (historias grandes), que luego se descompondrán en historias más pequeñas que se puedan incluir en una sola iteración. Además, nuevas historias se pueden escribir y agregar al backlog de productos en cualquier momento y por cualquier miembro del equipo. La flexibilidad en la escritura y la adición de User Stories es un principio clave en el desarrollo ágil.

Las "User Stories" no son lo mismo que los Casos de Uso ni que las Declaraciones de Requisitos. Tanto las User Stories como los Casos de Uso son herramientas para capturar y comunicar requerimientos que aportan valor al negocio, pero las User Stories son un enfoque ágil que se centra en las necesidades y objetivos del usuario, aportando flexibilidad y enfoque en el valor empresarial. Esto contrasta con los Casos de Uso y las Declaraciones de Requisitos, que tienden a ser más rígidos y centrados en los detalles del sistema.

Las **User Stories** tienden a ser más **pequeñas y específicas**, diseñadas para caber en una iteración de desarrollo, pueden ser más concisas y enfocadas en los aspectos más importantes. Mientras que los Casos de Uso pueden ser más extensos y abarcar un conjunto más amplio de funcionalidades, también suelen ser más detallados y completos en su descripción.

Los Casos de Uso tienden a ser más duraderos y pueden aplicarse a lo largo de todo el proyecto, incluir detalles de la interfaz de usuario y la interacción con el sistema en mayor medida, mientras que las **User Stories** están diseñadas para una **única iteración** y se centran más en los objetivos del usuario sin entrar en detalles de diseño de la interfaz.

Las declaraciones de requisitos se refieren a declaraciones específicas y detalladas como "El producto tendrá un motor de gasolina" o "El producto tendrá cuatro ruedas", y tienden a ser tediosas y pueden resultar en una carga de trabajo considerable para definirlas en su totalidad antes del desarrollo. Suelen centrarse en las características y funciones del sistema, mientras que las User Stories se enfocan en los objetivos y necesidades del usuario, lo que permite un enfoque más orientado al usuario y ágil.

Ejemplo:

En el caso de las Declaraciones de Requisitos, se especificará detalladamente que el producto debe tener un motor de gasolina, cuatro ruedas con neumáticos de goma montados en cada una, un volante y un cuerpo de acero. Esta especificación es exhaustiva y puede llevar a un enfoque rígido y poco ágil.

En contraste, una User Story podría ser: "Como un conductor, quiero un vehículo confiable para viajar a mi trabajo diario". Esta User Story se enfoca en el objetivo del usuario, dejando espacio para que el equipo de desarrollo proponga soluciones flexibles que cumplan con ese objetivo.

Diseño en XP

El diseño en la Programación Extrema (XP) se enfoca en varios principios clave. Estos principios se oponen al enfoque de "Implementar para hoy, diseñar para mañana" porque XP enfatiza la incertidumbre del futuro. En lugar de tratar de predecir y diseñar para todas las eventualidades futuras, XP se enfoca en diseñar de manera ágil y simple para abordar las necesidades actuales de manera efectiva.

Simplicidad: El diseño del sistema debe ser lo más simple posible. Esto significa evitar la complejidad innecesaria y mantener las soluciones elegantes y directas. La simplicidad facilita la comprensión del código y su mantenimiento.

Diseño Correcto: El diseño debe ser correcto, lo que significa que debe cumplir con los requisitos del usuario y funcionar como se espera. En XP, la corrección es fundamental.

*Superar Pruebas***:** El diseño debe pasar todas las pruebas. Esto implica que el software debe funcionar correctamente y ser robusto.

Eliminación de Duplicación: El diseño no debe tener lógica duplicada. La duplicación de código es propensa a errores y dificulta el mantenimiento. En XP, se busca eliminar esta duplicación.

Comunicar Intenciones: El diseño debe comunicar claramente las intenciones del desarrollador. Esto significa que el código debe ser fácil de leer y comprender para cualquier miembro del equipo.

Mínimas Unidades: El diseño debe tener la menor cantidad de unidades, como clases, métodos y funciones. Esto fomenta la simplicidad y evita la sobrecarga innecesaria.

Ventajas

El diseño en XP se enfoca en la simplicidad, la corrección y la eficacia a corto plazo, en lugar de tratar de anticipar y diseñar para un futuro incierto. Esto permite una mayor agilidad y adaptabilidad en el desarrollo de software.

- No se pierde tiempo en funcionalidad superflua: Se evita agregar características innecesarias que no aportan valor al usuario.
- Más fácil de entender: Un diseño simple es más fácil de comprender y de depurar, lo que facilita el trabajo en equipo.
- Más fácil de refactorizar y crear "propiedad colectiva" del código: La refactorización (mejora del código sin cambiar su funcionalidad) es más sencilla en un diseño simple. Además, promueve la propiedad compartida del código, lo que significa que cualquier miembro del equipo puede trabajar en él de manera efectiva.
- Ayuda a mantener a los programadores en el cronograma: Un diseño simple reduce la complejidad y el riesgo de retrasos en el desarrollo, lo que ayuda a mantener el equipo en el cronograma.

Refactoring en XP

Un precepto fundamental de la ingeniería de software tradicional es que se tiene que diseñar para cambiar. Esto es, deben anticiparse cambios futuros al software y diseñarlo de manera que dichos cambios se implementen con facilidad. Vale la pena el gasto de tiempo y esfuerzo anticipando los cambios ya que esto reduce los costos más tarde en la vida del ciclo XP, sin embargo, descarta este principio ya que sostiene que los cambios no se pueden prever de forma fiable

En su lugar, propone la mejora constante de código (Refactoring/Reconstrucción). El refactoring está motivado por los **smells**. Los términos ***"Design Smells"*** (malos olores de diseño) y ***"Code Smells"*** (malos olores de código)

se refieren a características o patrones en el diseño de software y el código que indican posibles problemas más profundos que pueden afectar la calidad del software.

Estos "smells" no son errores directos, es decir, el código puede funcionar correctamente, pero indican debilidades en el diseño que pueden conducir a problemas en el desarrollo. Sin embargo, son señales de advertencia que sugieren que el código o el diseño del software pueden necesitar ser refactorizados o mejorados para evitar problemas futuros y mantener un código limpio y mantenible.

Design Smells

Se centran en problemas más amplios relacionados con la arquitectura y estructura del software, y su impacto en la capacidad del software para adaptarse a cambios futuros.

- *Rigidez*: Ocurre cuando el diseño del software es difícil de cambiar o extender. Esto puede llevar a problemas si es necesario adaptar el software a nuevas necesidades o requisitos.
- *Fragilidad*: Se refiere a la facilidad con la que pequeños cambios en el software pueden causar la rotura de otras partes del sistema. Esto hace que el mantenimiento y la evolución del software sean complicados.
- *Inmovilidad*: Sucede cuando el código o las partes del sistema son difíciles de reutilizar en otros contextos o proyectos, lo que limita la flexibilidad y la eficiencia del desarrollo.
- *Viscosidad*: Se refiere a la tendencia de un sistema a resistir cambios que mejoran la calidad. Puede deberse a obstáculos en el proceso de desarrollo o decisiones arquitectónicas deficientes.

Code Smells

Los "Code Smells" son características específicas en el código que sugieren problemas potenciales. Se enfocan en problemas a nivel de código que afectan la legibilidad y mantenibilidad inmediata del software

Ejemplos:

- *Métodos muy largos*: Funciones o métodos con un exceso de líneas de código, lo que dificulta su comprensión y mantenimiento.
- *Métodos muy similares*: Varios métodos que realizan tareas similares pero no están refactored en un método común.
- *Clases grandes*: Clases que han crecido demasiado y se convierten en lo que se conoce como "God objects".
- *Dependencia inapropiada*: Clases que tienen dependencias indebidas de los detalles de implementación de otras clases.
- *Duplicación de código*: Porciones idénticas o muy similares de código que existen en múltiples ubicaciones del sistema.
- *Ciclomatic complexity*: Funciones con muchas ramificaciones y bucles que pueden requerir división en funciones más pequeñas.

Technical debt

Code smells son indicadores de factores que contribuyen al technical debt. Technical debt es un concepto en el desarrollo de software que refleja el costo implícito de rework causado por elegir una solución fácil en lugar de usar una mejor solución que hubiera tomado más tiempo en desarrollarse

Se puede comparar con las deudas financieras, si el technical debt no se paga acumula intereses, haciendo cambios futuros más difíciles.

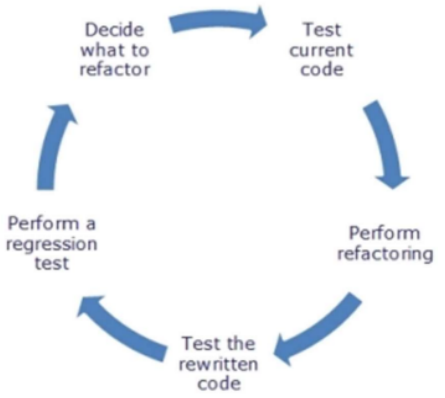
Refactoring

Los "Code Smells" son señales de advertencia que indican que una parte del código puede necesitar ser refactorizada. La refactorización es un proceso de mejora del código existente para que sea más limpio, más legible y más mantenible, sin cambiar su funcionalidad externa.

Los **beneficios** de la refactorización incluyen:

- La refactorización hace que el código sea más fácil de entender y de leer. Esto facilita la corrección de errores, ya que se vuelve más evidente dónde y cómo se deben realizar los cambios.
- Un código refacturado generalmente utiliza patrones de diseño reconocibles y es más flexible. Esto hace que sea más sencillo agregar nuevas funcionalidades o modificar el código existente sin introducir errores.
- La refactorización es una forma de abordar el "technical debt" (deuda técnica). El "technical debt" se refiere a la acumulación de código de baja calidad o no óptimo a lo largo del tiempo. La refactorización ayuda a pagar esa deuda técnica, mejorando la calidad del código y reduciendo los riesgos asociados con un código deficiente.

Un aspecto importante de la refactorización es que debe estar respaldada por un conjunto sólido de pruebas automatizadas con una suficiente cobertura. Estas pruebas garantizan que, a medida que se realiza la refactorización, no se introduzcan nuevos errores o problemas en el código existente. Por lo tanto, un "arnés de prueba automatizado" (conjunto de pruebas automatizadas) con una buena cobertura es esencial antes de emprender la refactorización.



La refactorización se lleva a cabo mediante ciclos iterativos de pequeñas transformaciones del programa y pruebas intercaladas. Esto significa que se realizan pequeñas mejoras en el código, seguidas de pruebas para garantizar que no se haya introducido ningún problema nuevo. Este enfoque iterativo garantiza que el código se mantenga en un estado funcional y de alta calidad a lo largo del tiempo, lo que facilita su mantenimiento y evolución.

Técnicas de Refactorización

Las técnicas de refactorización son acciones específicas que se pueden aplicar para mejorar la calidad del código y su estructura. Estas técnicas se utilizan para mejorar la calidad y la estructura del código, lo que a su vez facilita su mantenimiento y evolución a medida que se desarrolla el software.

Técnicas que permiten una mayor abstracción:

Encapsular Campo: Esta técnica implica forzar al código a acceder a un campo (variable) a través de métodos getter y setter en lugar de acceder directamente al campo. Esto permite un mayor control y encapsulación de los datos.

Generalizar Tipo: Consiste en crear tipos más generales o abstractos que permiten compartir más código. Esto fomenta la reutilización y la flexibilidad en el diseño.

Sustituir Código de Comprobación de Tipo con Estado/Estrategia: Esta técnica implica reemplazar bloques de código que realizan comprobaciones de tipo con un enfoque basado en estado o estrategia, lo que puede hacer que el código sea más limpio y extensible.

Sustituir Condicionales con Polimorfismo: Se refiere a reemplazar bloques de código condicionales (if-else) con un diseño basado en polimorfismo, donde se utilizan clases y métodos especializados para realizar diferentes acciones en lugar de múltiples condiciones.

Técnicas para dividir el código en piezas más lógicas:

Componentización: Esta técnica descompone el código en unidades semánticas reutilizables con interfaces claras y bien definidas. Esto facilita la reutilización y la comprensión del código.

Extraer Clase: Consiste en tomar parte del código de una clase existente y moverlo a una nueva clase. Esto se hace para mantener la cohesión y mejorar la organización del código.

Extraer Método: Se utiliza para dividir una parte de un método grande en un nuevo método separado. Esto hace que el código sea más fácil de entender y mantener, y también se aplica a funciones en lenguajes de programación que no son orientados a objetos.

Técnicas para mejorar nombres y ubicación del código:

Mover Método o Mover Campo: Estas técnicas implican mover un método o un campo a una clase o archivo fuente más apropiado. Esto se hace para mejorar la organización del código.

Renombrar Método o Renombrar Campo: Se utilizan para cambiar el nombre de un método o campo a uno que refleje mejor su propósito o funcionalidad.

Subir Método: En programación orientada a objetos, esto implica mover un método de una subclase a una superclase para compartirlo entre varias subclases.

Bajar Método: También en programación orientada a objetos, se trata de mover un método de una superclase a una subclase cuando es más relevante en ese contexto.

Pruebas en XP

En Extreme Programming (XP), las pruebas juegan un papel fundamental en el proceso de desarrollo de software, sirven para garantizar la calidad y la adaptabilidad del software. Los desarrolladores escriben pruebas unitarias y los clientes participan en la creación de pruebas de aceptación. La automatización de pruebas facilita la detección temprana de problemas y cambios seguros en el código.

En XP, los desarrolladores son responsables de escribir pruebas unitarias para las partes del código que están desarrollando. Estas pruebas se centran en probar unidades individuales de código para garantizar que funcionen correctamente.

Las pruebas unitarias deben pasar antes de que un desarrollador pueda considerar que ha completado una tarea o una funcionalidad. Esto asegura que el código desarrollado cumple con los requisitos y funciona como se espera. Los clientes (usuarios o representantes del cliente) colaboran con los desarrolladores para escribir pruebas de aceptación que describen el comportamiento deseado del sistema. Estas pruebas son utilizadas para determinar cuándo una tarea o una funcionalidad se considera finalizada.

Además, se hace hincapié en la automatización de las pruebas. Esto significa que las pruebas se pueden ejecutar de manera automática cada vez que se realiza una nueva compilación del software. Esta automatización es esencial para mantener un conjunto de pruebas completo y para detectar problemas rápidamente.

Las pruebas automatizadas se utilizan como un conjunto de regresión para asegurarse de que los cambios recientes no han introducido errores en el código existente, esto ayuda a mantener la calidad del software durante el proceso de desarrollo y refactorización.

Las pruebas en XP están diseñadas para que el sistema acepte el cambio. Esto significa que el código se puede modificar y evolucionar con confianza, ya que las pruebas garantizan que las funcionalidades existentes sigan funcionando correctamente después de cada cambio.

Ventajas de las Pruebas en XP:

- Promueven la creación de un conjunto completo de pruebas que cubren diferentes aspectos del software.
- Dan a los desarrolladores un objetivo claro y medible para verificar que su código cumple con los requisitos.
- La automatización de pruebas crea un conjunto de regresión que proporciona seguridad durante el proceso de refactorización y desarrollo continuo.

Desarrollo de Pruebas Primero

En XP, se prioriza el desarrollo de pruebas antes de escribir el código de la funcionalidad. Esto significa que antes de implementar una nueva característica o realizar cambios en el sistema, se definen las pruebas que verificarán que la funcionalidad cumple con los requisitos. Al desarrollar las pruebas primero, se aclara lo que se espera de la funcionalidad. Esto ayuda a garantizar que los requerimientos estén bien definidos y comprensibles tanto para los desarrolladores como para los clientes.

Las pruebas no son solo texto descriptivo; son programas ejecutables. Esto significa que se pueden ejecutar automáticamente para verificar el funcionamiento del sistema.

Pruebas Antiguas y Nuevas: Las pruebas antiguas y nuevas se ejecutan automáticamente. Cada vez que se agrega una nueva funcionalidad o se realiza un cambio, todas las pruebas, tanto las antiguas como las nuevas, se ejecutan para verificar que la funcionalidad existente siga funcionando correctamente y que no se hayan introducido nuevos errores (regresión).

Participación de los clientes

La participación de los clientes en el proceso de pruebas es fundamental. Los clientes colaboran en la definición de pruebas de aceptación que describen el comportamiento deseado del sistema. Estas pruebas ayudan a garantizar que el software cumple con las expectativas del cliente y que se ajusta a los requisitos del proyecto.

No obstante, las personas que adoptan el papel de clientes tienen tiempo disponible limitado y por lo tanto no pueden trabajar a tiempo completo con el equipo de desarrollo. Pueden pensar que la presentación de los requerimientos es suficiente contribución y por tanto pueden ser reacios a involucrarse en el proceso de prueba.

Automatización de pruebas

La automatización de pruebas implica que las pruebas se escriban como componentes ejecutables antes de que se implemente una tarea. Estas pruebas deben ser independientes y capaces de simular la entrada y verificar la salida del sistema. Se utiliza un marco de prueba automatizada, como JUnit, para facilitar la escritura y ejecución de pruebas.

Asegura que siempre haya un conjunto de pruebas que se pueda ejecutar rápidamente cada vez que se agrega nueva funcionalidad al sistema. Esto ayuda a identificar problemas introducidos por el nuevo código de manera inmediata, lo que contribuye a la calidad del software.

Dificultades en pruebas XP

Los programadores prefieren programación a las pruebas y a veces se toman atajos al escribir pruebas. Por ejemplo, pueden escribir ensayos incompletos que no comprueben todas las posibles excepciones que puedan ocurrir.

Algunas pruebas pueden ser muy difíciles de escribir de forma incremental. Por ejemplo, en una interfaz de usuario compleja, es a menudo difícil de escribir pruebas unitarias para el código que implementa la 'lógica de visualización "y flujo de trabajo entre las pantallas

Es difícil juzgar la integridad de un conjunto de pruebas. Aunque usted puede tener un montón de pruebas del sistema, la prueba de conjunto puede no proporcionar una cobertura completa.

Programación en parejas en XP

En Extreme Programming (XP), una práctica común es que los programadores trabajen en parejas. Esto significa que dos programadores trabajan juntos en el desarrollo de código. Es una práctica clave en XP que fomenta la colaboración, el aprendizaje compartido y la mejora continua del código. Aunque implica que dos programadores trabajen juntos en una tarea, sus beneficios en términos de calidad y productividad hacen que sea una lección valiosa en el desarrollo de software.

Trabajar en parejas ayuda a desarrollar una propiedad común del código. Esto significa que no solo un programador individual es responsable de una pieza de código, sino que dos personas comparten la responsabilidad. Esto fomenta la colaboración y la cohesión en el equipo.

Permite la difusión del conocimiento a través del equipo. Cada pareja de programadores tiene la oportunidad de aprender de las habilidades y el conocimiento del otro, lo que enriquece la capacidad del equipo en su conjunto.

La programación en pareja sirve como un proceso de revisión informal. Cada línea de código escrita es vista y revisada por más de una persona, lo que ayuda a identificar posibles errores y mejoras de manera temprana en el proceso de desarrollo.

La colaboración en parejas alienta la reconstrucción o refactorización del código. Cuando dos personas trabajan juntas, es más probable que discutan y mejoren el diseño y la calidad del código, lo que contribuye a mantener el código limpio y mantenible.

Aunque pueda parecer que dos personas trabajando en una tarea tomarían más tiempo, la experiencia ha demostrado que la productividad en la programación en pareja es similar a la de dos personas trabajando de forma independiente. Además, la calidad del código suele ser más alta debido a la revisión constante y a la colaboración.

Puntos clave

Los métodos ágiles son métodos incrementales de desarrollo que se centran en el desarrollo rápido, versiones frecuentes del software, reduciendolos gastos generales del proceso, e implican al cliente directamente en el proceso de desarrollo.

La decisión sobre si se debe utilizar un enfoque ágil o un enfoque dirigido por un plan para el desarrollo debe depender del tipo de software que está desarrollado, las capacidades del equipo de desarrollo y la cultura de la empresa que desarrolla el sistema.

La programación extrema es un método ágil bien conocido que integra una serie de buenas prácticas de programación tales como versiones frecuentes del software, el software de mejora continua y participación del cliente en el equipo de desarrollo.

SCRUM - Gestión de proyectos ágiles

Gestión de proyectos ágil

El enfoque ágil de gestión de proyectos, como se ejemplifica en Scrum, es una respuesta a los desafíos tradicionales de la gestión de proyectos de software. La gestión de proyectos ágil se caracteriza por un enfoque más flexible y adaptable que se alinea con la naturaleza incremental y cambiante del desarrollo de software. En lugar de seguir un plan detallado, se enfoca en valores como la colaboración, la comunicación y la entrega de software funcionando para satisfacer las necesidades del cliente. Scrum es un ejemplo de un marco de trabajo ágil ampliamente utilizado en la gestión de proyectos.

El enfoque tradicional para la gestión de proyectos es el direccionado por plan, los gerentes de proyectos elaboran un plan detallado que describe qué debe ser entregado, cuándo debe ser entregado y quién trabajará en el desarrollo del proyecto. Este plan a menudo se sigue rigurosamente a lo largo del proyecto.

El Manifiesto Ágil define los valores y principios que guían la gestión de proyectos ágil. Estos valores incluyen la prioridad de individuos e interacciones sobre procesos y herramientas, la entrega de software funcionando sobre documentación extensa, la colaboración con el cliente y la respuesta al cambio. La gestión de proyectos ágil destaca la colaboración entre el equipo y el cliente, la comunicación efectiva y la entrega continua de software.

A diferencia de enfoques como el PMBOK (Project Management Body of Knowledge), que proporciona pasos concretos para la gestión de proyectos, la gestión de proyectos ágil no se adhiere a un conjunto rígido de pasos. En su lugar, se busca adoptar una metodología o marco de trabajo ágil, como Scrum, y se adapta según las necesidades del proyecto.

Scrum

Scrum es un ejemplo de marco de trabajo ágil, define una estrategia flexible en la que un equipo de desarrollo trabaja de manera colaborativa y autónoma para alcanzar una meta común. Se basa en iteraciones y entrega incremental de funcionalidad.

Se utiliza para abordar problemas que son complejos y cambiantes, lo que es común en el desarrollo de software. Proporciona un enfoque ágil para manejar la incertidumbre y los cambios constantes. El objetivo de Scrum es entregar productos de la máxima calidad y valor de manera productiva y creativa. Esto se logra mediante la colaboración y la adaptación continua.

Es un marco de trabajo ligero, lo que significa que es fácil de entender y adoptar. No es un proceso o técnica específica, sino un marco en el que se pueden emplear diversas prácticas y procesos. A pesar de su simplicidad aparente, Scrum puede ser extremadamente difícil de dominar. Requiere una comprensión profunda y la práctica continua para implementarse de manera efectiva.

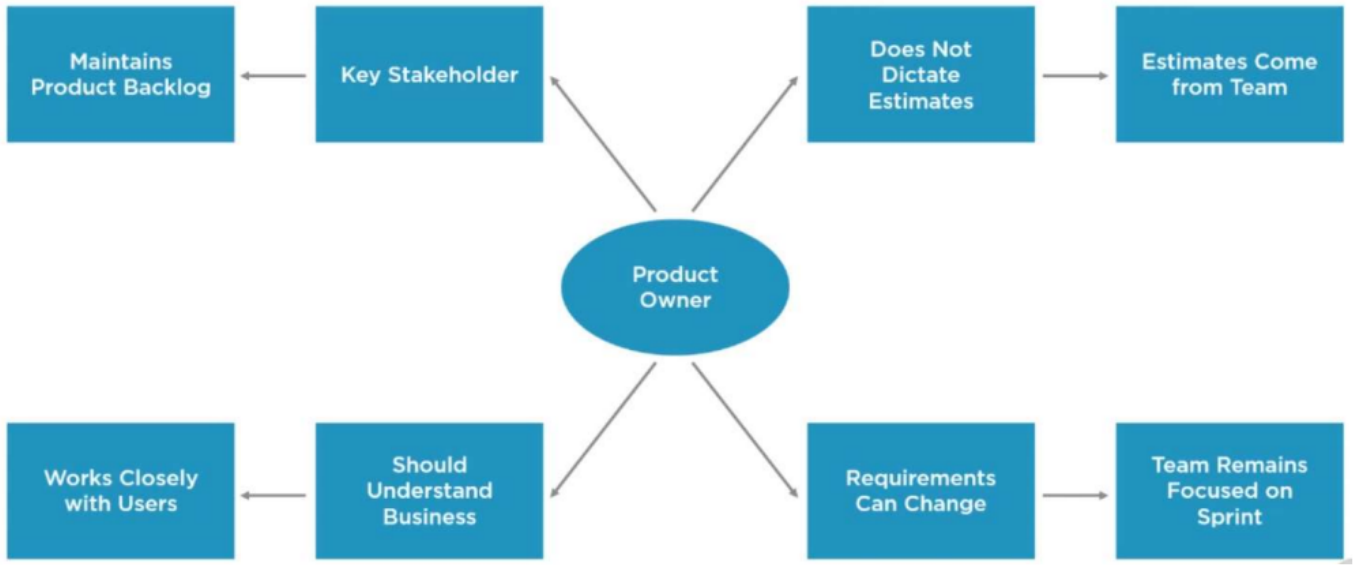
Scrum se basa en la teoría del control de proceso empírica o empirismo. Esto implica tomar decisiones basadas en hechos reales y experiencia en lugar de suposiciones. La gestión de proyectos Scrum es iterativa e incremental.

Los pilares son la transparencia, la inspección y la adaptabilidad. Estos principios guían la toma de decisiones en el marco de trabajo.

Además, Scrum utiliza ciclos de feedback para la mejora continua. El tiempo se divide en sprints, que son ciclos cortos. El producto debe estar en un estado "potencialmente entregable" en todo momento, y al final de cada sprint se realizan eventos formales para revisar los resultados y planificar el siguiente sprint. Los eventos formales en Scrum incluyen la Reunión de Planificación del Sprint, el Scrum Diario (Daily Scrum), la Revisión del Sprint y la Retrospectiva del Sprint. Estos eventos ayudan a mantener la transparencia, permiten la inspección y facilitan la adaptabilidad en el proceso de desarrollo.

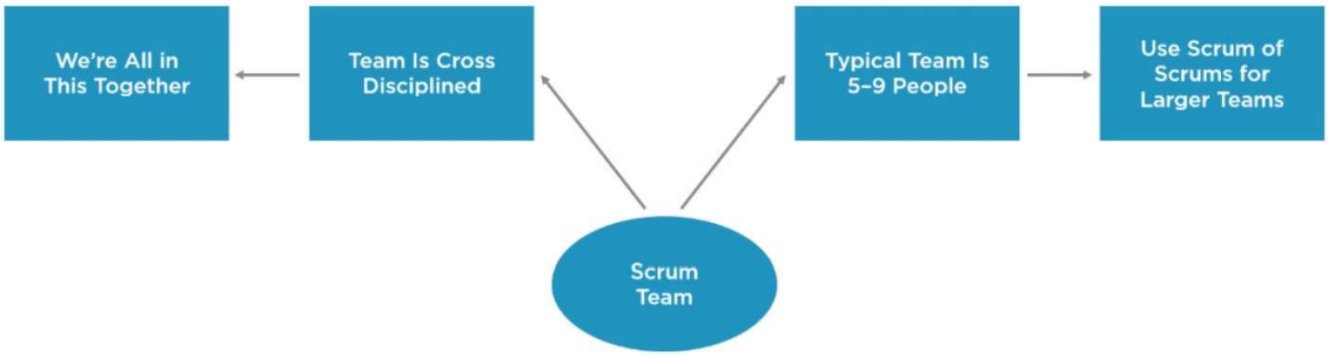
Scrum Roles

Product Owner



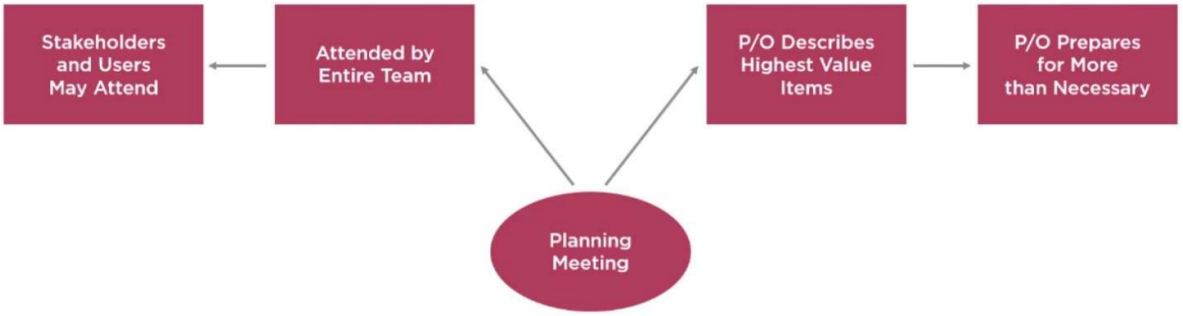
Scrum Master

Scrum Team

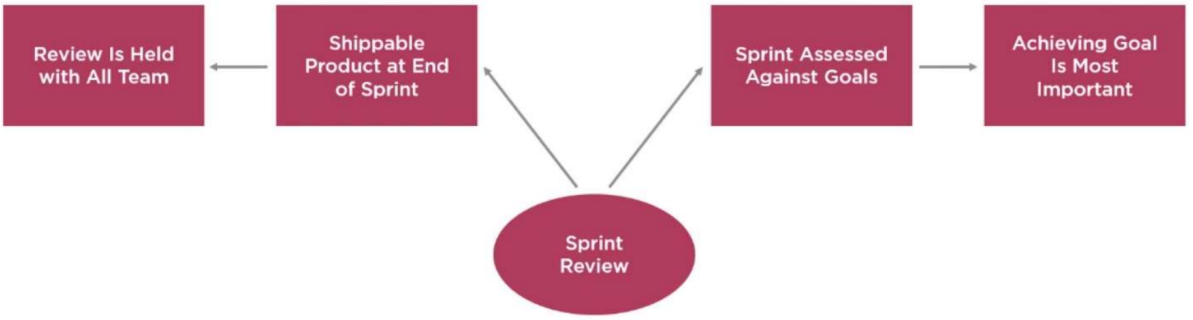


Scrum Eventos

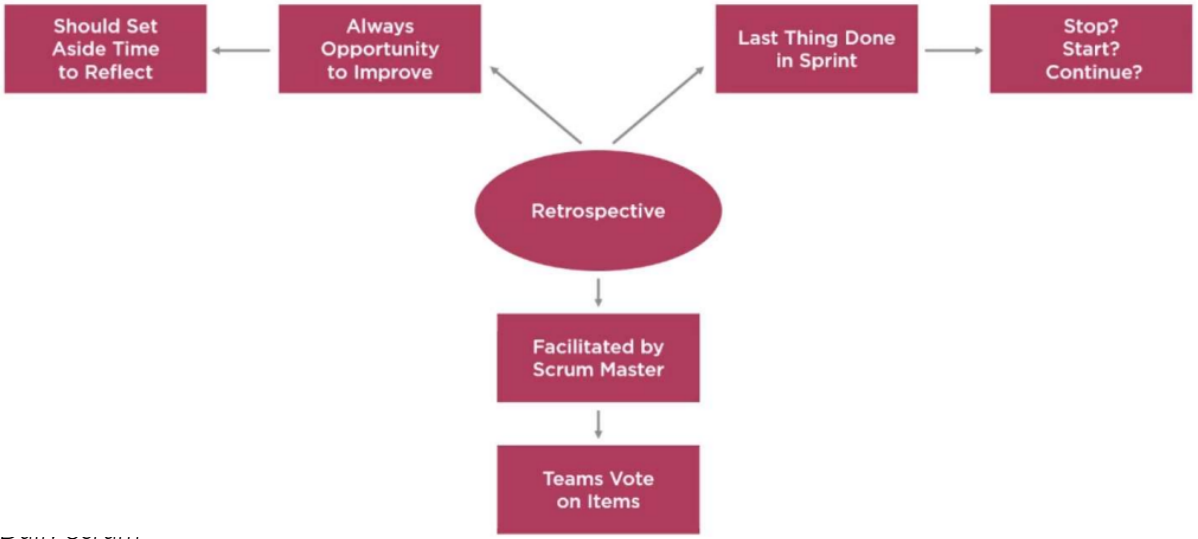
Planning



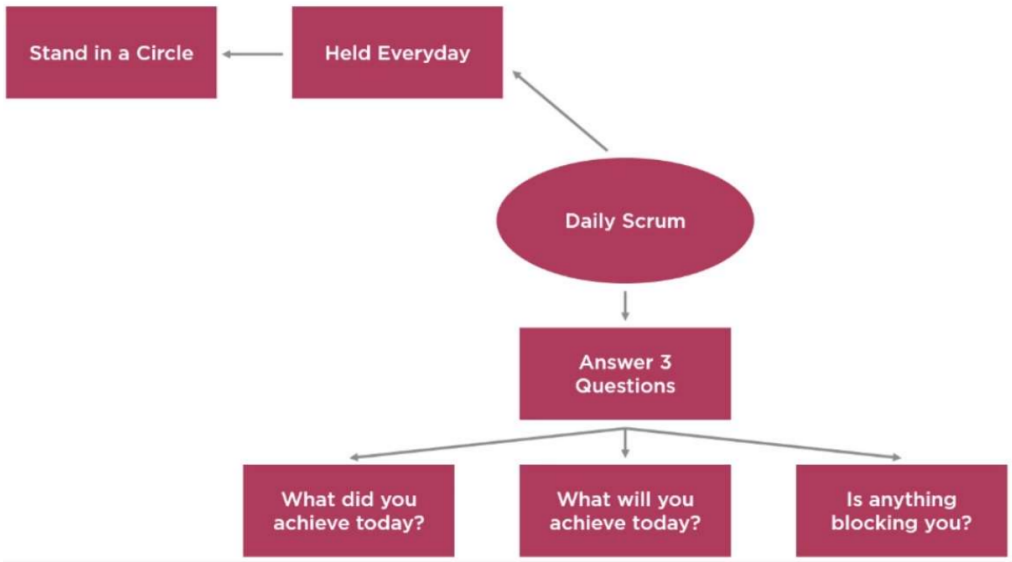
Sprint Review



Sprint Retrospective

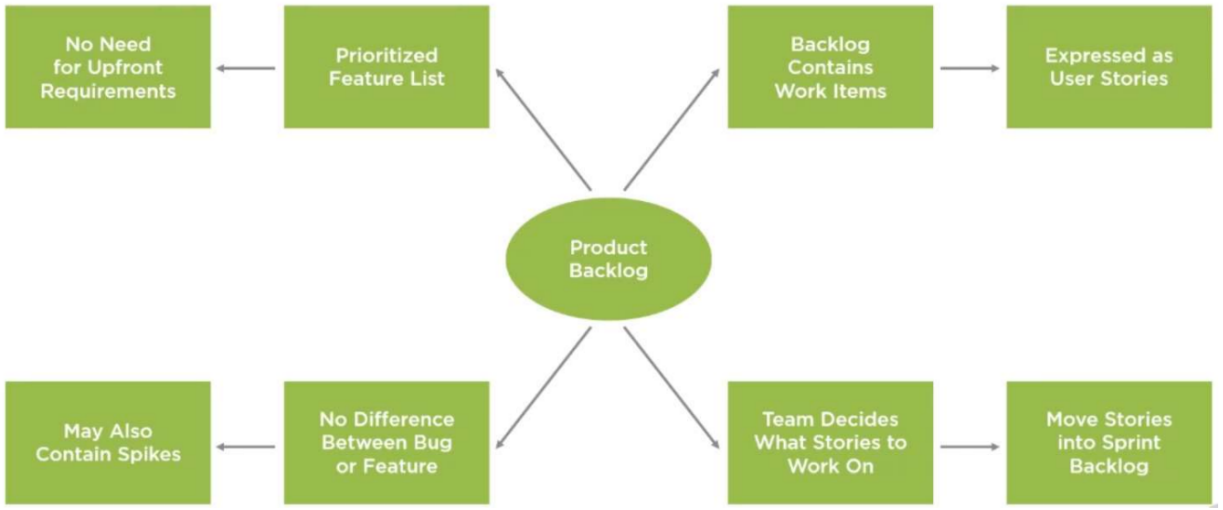


Daily Scrum

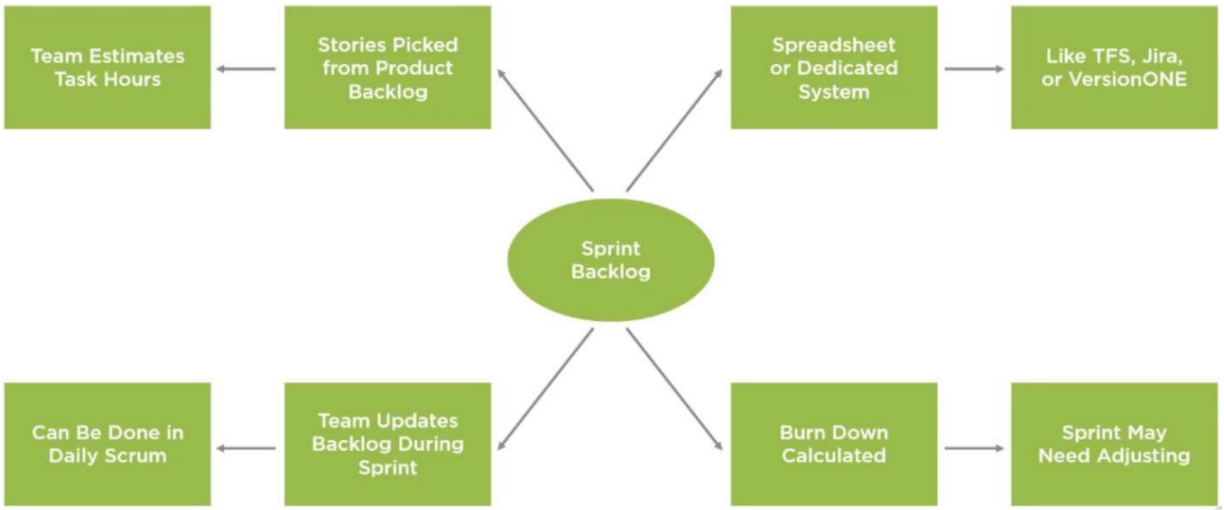


Scrum Artefacts

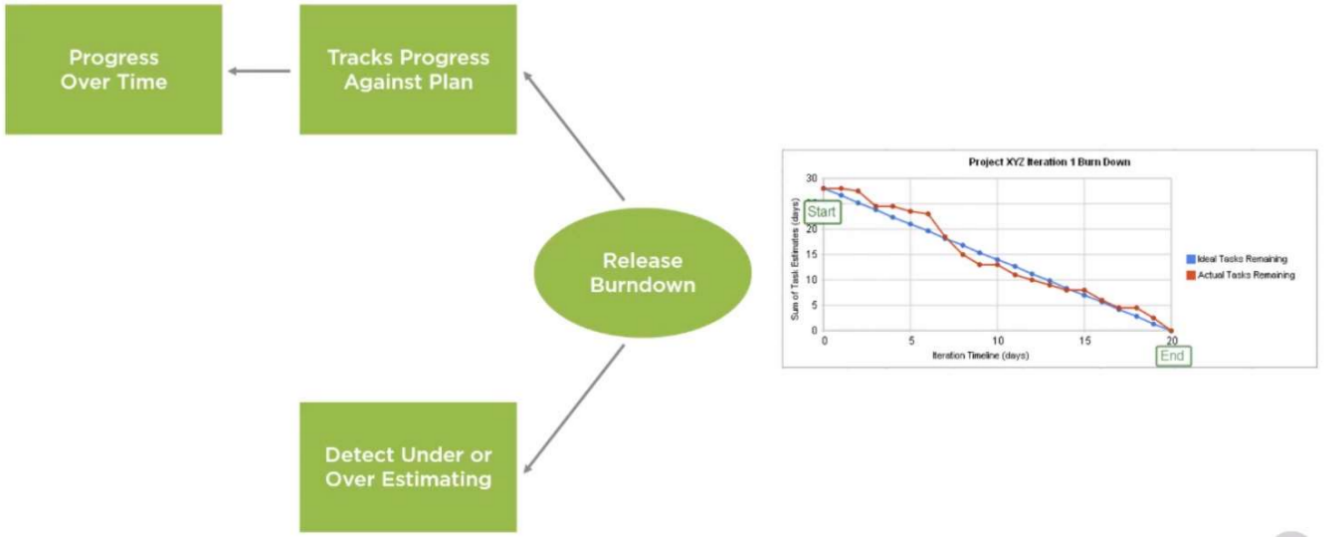
Product Backlog



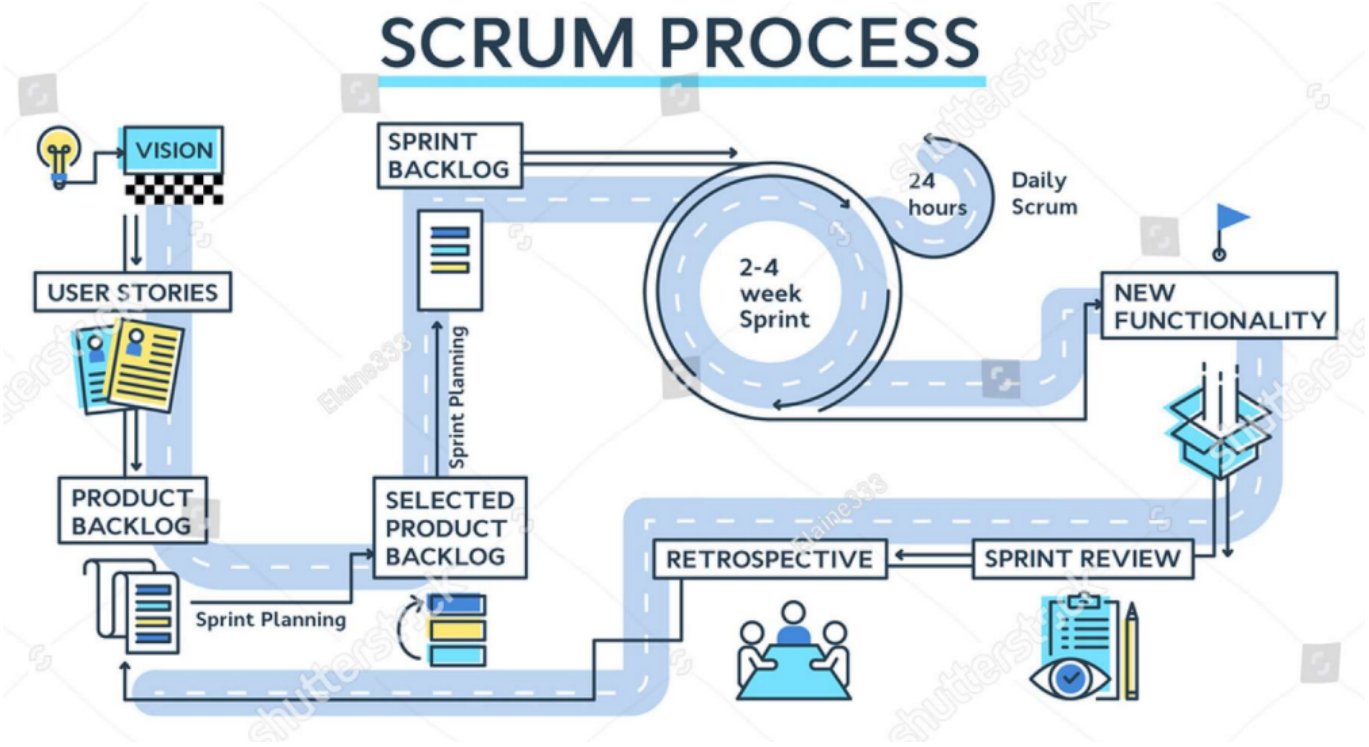
Sprint Backlog



Burndown Chart



El proceso de Scrum



Terminología

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	A measure of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

El ciclo de Sprint

Los sprints son de longitud fija, normalmente 2-4 semanas. Se corresponden al desarrollo de una versión del sistema en XP. El punto de partida para la planificación es la acumulación de stories, que es la lista de trabajo a realizar en el proyecto.

La fase de selección involucra a todo el equipo del proyecto, que trabajan con el cliente para seleccionar las funciones y funcionalidad que se desarrollará durante el sprint. Basados en: prioridades, esfuerzo estimado y la velocidad del team.

Planeamiento

- Estimaciones: Durante la planificación del sprint, el equipo realiza estimaciones para determinar cuánto trabajo se puede realizar durante el sprint. Las estimaciones se pueden realizar utilizando diferentes técnicas, como los puntos de historia o el poker planning.
- Puntos de Historia: Los puntos de historia son una unidad de medida comúnmente utilizada en Scrum para estimar el esfuerzo requerido para completar una historia. Estas estimaciones ayudan a determinar cuántas historias se pueden incluir en un sprint.
- Poker Planning: El poker planning es una técnica en la que los miembros del equipo asignan puntos de historia a las historias de usuario. Esto se hace de manera colaborativa y ayuda a establecer un consenso sobre las estimaciones.
- Velocidad: La velocidad del equipo se refiere a la cantidad de trabajo que el equipo puede completar de manera consistente durante un sprint. Se basa en la experiencia acumulada en sprints anteriores y se utiliza para predecir cuánto trabajo se puede realizar en futuros sprints.

Una vez que éstos están de acuerdo, el equipo se organiza para desarrollar el software. Durante esta etapa, el equipo está "aislado" del cliente y la organización, con toda comunicaciones canalizadas a través del denominado 'Scrum Master'. El papel del Scrum Master es proteger el equipo de desarrollo de las distracciones externas. Al final del sprint, el trabajo realizado es revisado y se presenta a las partes interesadas.Después, el siguiente ciclo comienza nuevamente.

Trabajo en equipo en Scrum y Daily Stand-ups

El Scrum Master es un facilitador que organiza reuniones diarias, rastrea la acumulación de trabajo por hacer, registra las decisiones, mide el progreso contra el atraso y se comunica con los clientes y la gestión fuera del equipo.

Todo el equipo asiste a las reuniones diarias cortas donde todos los miembros del equipo comparten información, describen su progreso desde la última reunión, los problemas que han surgido y que se ha previsto para el día siguiente. Esto significa que todos en el equipo saben lo que está pasando y, si surgen problemas, puede volver a planear el trabajo a corto plazo para hacer frente a ellos.

- Preguntas en los daily stand ups: *¿Qué hice ayer?¿Qué planeo hacer hoy?¿Tengo algún impedimento?*

Beneficios del Scrum

- El producto se divide en un conjunto de fragmentos manejables y comprensibles.
- Requerimientos inestables no retrasan el progreso.
- Todo el equipo tiene visibilidad de todo y por lo tanto se mejora la comunicación del equipo.
- Los clientes ven la entrega a tiempo de los incrementos y obtienen retroalimentación sobre cómo funciona el producto.
- La confianza entre los clientes y los desarrolladores se establece y una cultura positiva se crea en la que todo el mundo espera que el proyecto tenga éxito.

UNIDAD 5 - AUTOMATIZACIÓN DE PRUEBAS

Muchos proyectos solamente dependen del testing manual para verificar que el software cumpla con los requerimientos funcionales y no funcionales A veces, existen tests automatizados pero están desactualizados y pobremente mantenidos y requieren que se los suplemente con test manual extensivo

"Build quality in" (Construir la calidad desde el inicio) es un principio fundamental en el desarrollo de software ágil. Significa que en lugar de depender en inspecciones y pruebas masivas al final del proceso de desarrollo para garantizar la calidad, se debe enfocar en incorporar la calidad en el producto desde el principio del proyecto y mantenerla a lo largo de todo el ciclo de desarrollo. Esto implica que se integra en todas las etapas del proceso.

Para testing y la calidad del software, "building quality in" significa:

Escribir tests automatizados a diferentes niveles: Esto implica la creación de pruebas automatizadas en varios niveles del software, como pruebas de unidad para verificar funciones individuales, pruebas de componentes para evaluar módulos más grandes y pruebas de aceptación para verificar que el software cumple con los requisitos del usuario.

Ejecutar pruebas como parte del deployment pipeline: El deployment pipeline es un conjunto automatizado de pasos que se inicia cada vez que se realiza un cambio en el código, la configuración o el entorno del software. Como parte de este proceso, se ejecutan pruebas automatizadas para garantizar que los cambios no introduzcan nuevos errores y que el software siga funcionando como se espera.

La idea es que el testing automatizado se integre de manera continua en el proceso de desarrollo, lo que permite detectar y corregir problemas de manera temprana y garantizar que la calidad esté presente en cada versión del software que se construye. Esto ayuda a reducir la necesidad de correcciones masivas o inspecciones finales, ya que los problemas se abordan de manera proactiva a lo largo del ciclo de desarrollo.

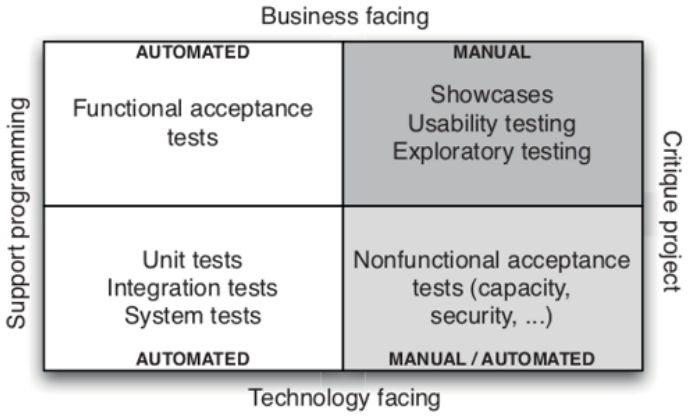
Un **proyecto ideal** implica varios aspectos clave para garantizar la calidad y la eficiencia en el proceso de desarrollo. Fomenta una colaboración cercana y continua entre diferentes partes interesadas, incluidos los testers, los desarrolladores y los usuarios. Esto permite una comprensión compartida de los requisitos y la calidad esperada del software.

Los tests automatizados se escriben desde el comienzo del proyecto, incluso antes de que los desarrolladores empiecen a trabajar en las características que los tests van a evaluar ya que describen en detalle cómo se supone que el sistema debe funcionar y qué resultados se esperan, esto asegura que la calidad sea una prioridad desde el principio.

Cuando los tests automatizados pasan, demuestran que la funcionalidad requerida por el cliente ha sido implementada completamente y correctamente. Esto garantiza que el software cumple con los requisitos del usuario. Los tests automatizados se ejecutan en un sistema de CI cada vez que se realiza un cambio en el código, la configuración o el entorno del sistema. Esto asegura que los tests se ejecutan de manera regular y ayuda a detectar problemas temprano.

Los tests automatizados no solo se utilizan para verificar nuevas características, sino que también sirven como tests de regresión. Esto significa que se ejecutan para garantizar que las actualizaciones no hayan introducido errores en las áreas existentes del software. Además de las pruebas de funcionalidad, un proyecto ideal también incluye tests automatizados para evaluar aspectos "no funcionales" del sistema, como rendimiento, seguridad y escalabilidad.

Tipos de Tests



Soporte del proceso de desarrollo, de cara al negocio

Los tests funcionales de aceptación desempeñan un papel fundamental en el soporte del proceso de desarrollo en un entorno ágil. Estos se centran en asegurar que los criterios de aceptación de una historia (o feature) se cumplan y que el software cumpla con las expectativas del usuario. Idealmente, los tests funcionales de aceptación se escriben y automatizan antes de que el desarrollo de una historia comience. Esto proporciona una especificación clara de lo que se espera del desarrollo y establece criterios claros para la finalización de la implementación. Estos tests responden a dos preguntas fundamentales en un entorno ágil:

- *¿Cómo sé cuándo terminé la implementación? (respondido por el desarrollador)*
- *¿Conseguí lo que quería? (respondido por el usuario)*

En un proyecto ideal, los usuarios, o aquellos que comprenden mejor las necesidades del negocio, deberían participar en la redacción del criterio de éxito de una historia. Este criterio se traduce en un test de aceptación. Una herramienta usada es “Cucumber”, permiten a los usuarios escribir scripts de prueba en un lenguaje más comprensible para ellos.

Tanto los testers como los desarrolladores trabajan en la implementación de los tests de aceptación. Los testers aportan su experiencia en pruebas y calidad, mientras que los desarrolladores trabajan en la implementación técnica. La herramienta, como Cucumber, actúa como un mecanismo para mantener sincronizados a todos los involucrados, ya que el mismo script de prueba se convierte en una especificación ejecutable que se ejecuta en el software.

El modelo "Given-When-Then" (GWT) es una forma estructurada de escribir pruebas o especificaciones, como pruebas unitarias, pruebas de aceptación y pruebas funcionales, ayuda a garantizar que se comprendan y validen claramente los comportamientos y resultados del sistema.

El modelo es eficaz para describir el "camino feliz" (**happy path**) de la prueba, que es el escenario ideal en el que todo funciona como se espera:

1. Given (Dado): Se establece el contexto inicial necesario para la prueba (configuración de datos, el estado del sistema). Es la preparación previa a la acción principal de la prueba.
2. When (Cuando): Se describen las acciones o eventos que se desencadenan, lo que se está probando o la acción principal que se está llevando a cabo. Esto representa la operación que se evalúa en la prueba.
3. Then (Entonces): Se establecen las expectativas o resultados esperados después de que se haya realizado la acción descrita en la sección "When", acá se verifica si el sistema se comporta de la manera esperada.

Además del "happy path," también es importante considerar escenarios alternativos (**alternative paths**), estos casos pueden incluir variaciones en el estado inicial, acciones alternativas o resultados inesperados. y "**sad paths**" que involucran situaciones de error o condiciones inesperadas.

La **automatización de los tests de aceptación** ofrece numerosos beneficios:

Costo: Puede requerir una inversión inicial de tiempo y recursos, pero a largo plazo, puede ahorrar dinero, ya que elimina la necesidad de realizar pruebas repetitivas manualmente en cada ciclo de desarrollo, lo que reduce los costos laborales y mejora la eficiencia.

Retroalimentación más rápida: Las pruebas automatizadas se pueden ejecutar rápidamente, lo que proporciona retroalimentación inmediata sobre la calidad del código. Esto permite identificar y solucionar problemas de manera más eficiente y, en última instancia, acelera el proceso de desarrollo.

Reducción de la carga de los testers: Al automatizar las pruebas de aceptación, los testers pueden enfocarse en pruebas más creativas y exploratorias en lugar de realizar pruebas repetitivas y manuales. Esto mejora la calidad general de las pruebas y aumenta la capacidad de encontrar problemas complejos.

Conjunto de tests de regresión poderoso: Cuando se realizan cambios en el código, las pruebas automatizadas pueden ejecutarse para asegurarse de que las nuevas características o correcciones no afecten negativamente las funcionalidades existentes. Si alguna modificación futura causa un problema en una funcionalidad existente, los tests automatizados lo identificarán.

Documentación del sistema: Al utilizar nombres legibles por humanos en los tests de aceptación automatizados, estos tests pueden servir como documentación viva del sistema. Los tests describen claramente cómo se espera que funcione el sistema, lo que facilita la comprensión del comportamiento del software.

Los **tests unitarios**, los **tests de componentes** y los **tests de despliegue** son tipos de pruebas que pueden ser escritos y mantenidos exclusivamente por desarrolladores.

Unit Tests: Evalúan unidades individuales de código, como funciones o métodos, para asegurarse de que funcionen correctamente. Está muy enfocado en una parte específica del código, necesitan simular otras partes del sistema (test doubles) y no deberían llamar a DB y otros sistemas. Generalmente lo escribe el mismo desarrollador que escribió el código, deben ser muy rápidas y actualizarse cuando se modifican las unidades de código correspondientes. Al realizarse en una sola parte del software, pierden la posibilidad de detectar bugs que provienen de la interacción entre diferentes partes del software.

Component Tests: También se conocen como tests de integración, evalúan componentes más grandes o módulos del sistema para garantizar que funcionen de manera efectiva juntos, pueden abarcar múltiples unidades de código o clases interconectadas. Son escritas por desarrolladores que trabajan en esas áreas específicas o en la integración de componentes. Deben actualizarse cuando cambian los componentes que están siendo evaluados.

Deployment Tests: Se ejecutan cada vez que se instala la aplicación y verifican que el deployment funcionó, que la aplicación fue correctamente instalada, correctamente configurada y capaz de contactar cualquier servicio que necesite y que este responda.

Critican el proyecto de cara al negocio

Las pruebas manuales son esenciales para verificar el proyecto desde la perspectiva del negocio y garantizar que la aplicación cumple con las expectativas del usuario.

Verifican que la aplicación le entrega al usuario el valor que este está esperando. No es solo validar que la aplicación cumpla con las especificaciones, es también validar que las especificaciones estén correctas; cuando los usuarios utilizan la aplicación en vida real siempre descubren que se puede mejorar. Tal vez se descubren nuevas features inspiradas en la entrega actual

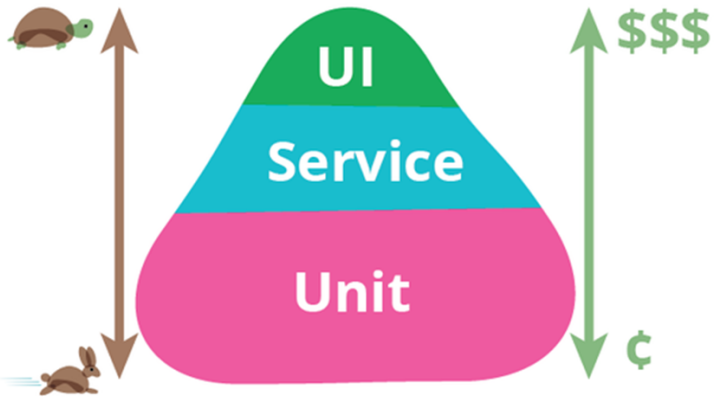
Las demostraciones y revisiones realizadas por los equipos ágiles al final de cada iteración son fundamentales. Estas demostraciones permiten a los usuarios y al equipo evaluar el resultado del desarrollo y proporcionar comentarios valiosos. La flexibilidad es clave, ya que tanto el usuario como el equipo pueden ajustar el plan en función de los comentarios recibidos. Solo cuando los tests pasan con éxito, podemos afirmar que se ha cumplido con éxito la tarea. Además, el uso de pruebas tipo "Canary" se utiliza para identificar posibles problemas o errores en etapas tempranas, lo que contribuye a la mejora continua del proyecto.

Critican el proyecto de cara a la tecnología

Los tests de aceptación se dividen en dos categorías principales: funcionales y no funcionales. Mientras que los tests funcionales se centran en las características directas del sistema, los tests no funcionales se enfocan en aspectos como la capacidad, disponibilidad y seguridad, entre otros. A menudo, los usuarios no especifican estos requisitos no funcionales desde el principio, pero se dan por sentados en muchos casos. La realización de tests no funcionales suele requerir recursos significativos, herramientas específicas y tiende a ser más lenta. Generalmente, se ejecutan hacia el final del proceso de implementación del sistema para garantizar que se cumplan los estándares requeridos.

Test en el Commit Stage

En el desarrollo de software, la mayoría de las pruebas deben ser pruebas de unidad. Estas pruebas son rápidas de ejecutar y deben cubrir al menos un 80% del código. Sin embargo, es importante recordar que las pruebas de unidad, por sí solas, no garantizan el funcionamiento completo de la aplicación. Para ello, se necesita el respaldo del resto del pipeline de implementación.



Para lograr que los tests de unidad sean rápidos, hay varias estrategias efectivas que se pueden seguir:

1. Evitar la interfaz de usuario: Las pruebas de unidad no deben involucrar interacciones con la interfaz de usuario, ya que los tiempos de respuesta de los humanos son extremadamente lentos en comparación con las máquinas. Además, trabajar con una gran cantidad de componentes en las pruebas puede requerir un esfuerzo considerable.

2. Inyección de dependencias o inversión de control: Al permitir que las dependencias sean proporcionadas desde fuera, se facilita la introducción de "test doubles" o sustitutos para las dependencias reales. Esto promueve un buen diseño que hace que las pruebas sean más eficientes.

3. Evitar la base de datos: Es recomendable evitar cualquier subsistema que no esté directamente relacionado con la funcionalidad que se está probando. Las pruebas de unidad no deben depender de la base de datos o cualquier otro componente externo.

4. Evitar el asincronismo: En la etapa de compromiso (commit stage) de las pruebas, es importante evitar el asincronismo, ya que puede hacer que las pruebas sean lentas. Si es necesario trabajar con operaciones asincrónicas, se pueden dividir los tests para que sean más manejables y rápidos.

5. Utilizar test-doubles: La creación de "test doubles" o sustitutos para las dependencias permite simular el comportamiento deseado sin involucrar sistemas complejos. Esto acelera la ejecución de las pruebas.

6. Minimizar el estado en los tests: En lugar de probar una amplia variedad de estados, se centra en probar el comportamiento. Evitar pruebas complejas que requieran un esfuerzo considerable para preparar los datos de entrada.

7. Simular el tiempo: Abstraer la gestión del tiempo en una clase separada y permitir la inyección de una implementación que simule el tiempo. Esto evita esperas innecesarias en las pruebas.

8. Fuerza bruta: Paralelizar la ejecución de las pruebas y separar los conjuntos de pruebas para poder ejecutar pruebas simultáneamente, lo que acelera el proceso de prueba.

Test doubles

En el contexto de las pruebas automatizadas, una estrategia clave es reemplazar partes del sistema con versiones simuladas para tener un mayor control sobre su comportamiento. Estas partes simuladas se conocen como "test doubles" o sustitutos de prueba, y existen varios tipos:

1. Dummy objects: Son objetos que se pasan como parámetros pero nunca se utilizan en las pruebas. Su propósito es llenar un requisito de argumento, pero su lógica interna no es relevante para la prueba.

2. Fake objects: Estos objetos tienen implementaciones funcionales, pero toman atajos que los hacen inutilizables en producción. Por ejemplo, se pueden utilizar bases de datos en memoria en lugar de bases de datos reales para acelerar las pruebas.

3. Stubs: Los stubs proporcionan respuestas predeterminadas a las llamadas que reciben durante las pruebas. Se utilizan para simular el comportamiento de componentes o servicios externos y permiten que las pruebas se centren en un código específico sin depender de componentes reales.

4. Spies: Los spies son objetos que registran información sobre cómo son llamados durante las pruebas. Se utilizan para verificar que ciertas acciones o llamadas hayan ocurrido durante la ejecución de una prueba.

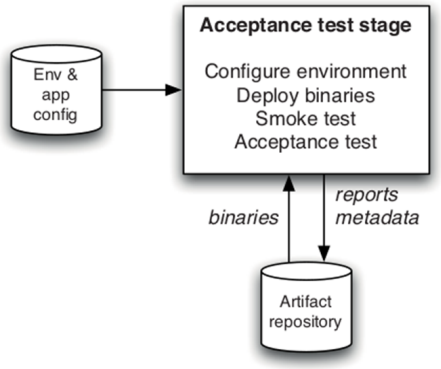
5. Mocks: Los mocks son objetos preprogramados con "expectativas" que definen una especificación de las llamadas que esperan recibir. Se utilizan para verificar que las interacciones entre objetos se ajusten a ciertas expectativas predefinidas.

El uso de estos "test doubles" permite aislar componentes y simplificar las pruebas al controlar el comportamiento de las partes del sistema que no son relevantes para la prueba actual. Esto facilita la identificación de problemas y garantiza que las pruebas sean efectivas y confiables.

Test de Aceptación Automatizado

Los tests de aceptación automatizados son una parte fundamental en el pipeline de implementación. Van más allá de la integración continua y se centran en evaluar el cumplimiento de los criterios de aceptación del

negocio. Estas pruebas se ejecutan contra cada versión de la aplicación que ha pasado la etapa de compromiso (commit stage), asegurando así que el software cumple con los estándares requeridos antes de su despliegue.



Los tests de aceptación automatizados son esenciales por varias razones. En primer lugar, reducen significativamente el costoso proceso de pruebas manuales, ya que no es necesario repetirlo en cada nueva versión del software. Además, al hacer pruebas manuales generalmente al final, los defectos se capturan tarde en el ciclo de desarrollo, lo que aumenta el riesgo.

Los tests de unidad y de componente, aunque importantes, no se centran en probar escenarios completos, por lo que no son suficientes para garantizar el cumplimiento de los criterios de aceptación del negocio. Además, los tests de aceptación actúan como una protección contra cambios importantes en el diseño o la arquitectura del software, ya que no es necesario rehacerlos cuando se realizan modificaciones.

En última instancia, los tests de aceptación automatizados fomentan la planificación adecuada y ejercen una presión saludable para mantener altos estándares de calidad en el desarrollo de software.

Test de Aceptación Automatizados Mantenibles

Los tests de aceptación automatizados mantenibles son esenciales en el desarrollo de software. Deben basarse en los criterios de aceptación y se deben escribir teniendo en cuenta la automatización desde el principio. Una vez que se establecen los Criterios de Aceptación (CoS), se deben automatizar.

Los tests de aceptación deben implementarse en capas. La primera capa se centra en los criterios de aceptación y se debe redactar en el lenguaje del dominio, evitando detalles de interacción con la aplicación para evitar que se vuelvan frágiles. Además, se recomienda que estos tests no se ejecuten contra la interfaz de usuario (UI), sino que prefieran utilizar la misma API que la UI.

El rol de Business Analyst, Product Owner o Representante del cliente y usuarios es esencial para identificar y priorizar los requerimientos, garantizar que los desarrolladores comprendan las necesidades del usuario y que las historias de usuario entreguen el valor de negocio esperado. Este rol es una función clave en el proceso y no necesariamente una persona específica.

Implementación de los tests de aceptación

La implementación de tests de aceptación es esencial en el desarrollo de software, pero presenta desafíos únicos. A diferencia de los tests de unidad, los tests de aceptación no pueden ser completamente stateless, ya que necesitan simular las interacciones del usuario y probar que el sistema cumple con los requerimientos del negocio. Para manejar el estado, se recomienda evitar cargar bases de datos completas de producción y en su lugar mantener un conjunto mínimo y controlado de datos. Además, es fundamental usar la API pública de la aplicación para configurar el estado en lugar de ejecutar scripts directamente contra la base de datos.

La encapsulación y el diseño adecuado son esenciales en los tests de aceptación, y se debe resistir la tentación de romper el encapsulamiento para fines de prueba. Es importante evitar la creación de "backdoors" o código específico para pruebas. Además, la asincronía y los tiempos de espera son desafíos en los tests de aceptación, ya que se deben probar interacciones completas. Se debe minimizar el tiempo de espera siempre que sea posible.

Los tests de aceptación deben ejecutarse en un entorno similar al de producción y no deben incluir integración con sistemas externos. En su lugar, se pueden utilizar "tests doubles" para controlar el estado y el comportamiento de sistemas externos. Un buen diseño minimiza el acoplamiento entre la aplicación y los sistemas externos, lo que permite implementar patrones como el "circuit breaker".

En cuanto a los puntos de integración externos, es fundamental probar la integración en lugar de los sistemas externos en su totalidad. Los tests deben centrarse en las interacciones necesarias y adaptarse a las circunstancias, ya sea que los sistemas externos estén en producción madura o en desarrollo activo. La estrategia es empírica, creando tests que aborden los problemas a medida que se descubren.

En resumen, los tests de aceptación requieren planificación cuidadosa, diseño eficiente y estrategias específicas para manejar el estado, la encapsulación, la asincronía y la integración con sistemas externos.

Deployment tests

Las pruebas de implementación, o deployment tests, son fundamentales para asegurarse de que el entorno donde se ejecutan los tests de aceptación sea lo más similar a producción posible, idealmente idéntico. Esto proporciona una excelente oportunidad para probar los instaladores en un entorno de producción simulado y garantizar que los componentes se puedan comunicar correctamente. Estas pruebas deben incluirse como una suite adicional al principio del pipeline de implementación para que cualquier falla en ellas cause un fallo inmediato, evitando esperas innecesarias de tiempo de espera.

UNIDAD 6 - MÉTRICAS

Medición y métricas del software

La medición del software se ocupa de derivar un valor numérico para un atributo de software como su complejidad o confiabilidad, comparando los valores medidos entre ellos y con los estándares de la organización se pueden extraer conclusiones acerca de: la calidad del software y evaluar la efectividad del proceso, herramientas o métodos

El objetivo final de la medición del software es usar la medición para hacer juicios sobre la calidad del software Idealmente, un sistema podría ser evaluado usando un rango de métricas para medir sus atributos de las medidas obtenidas inferir un valor de su calidad y compararlo contra un threshold para saber si la calidad requerida ha sido alcanzada Sin embargo la industria está aún lejos de la situación ideal

Una métrica del software es una característica de un sistema de software, de su documentación, o del proceso de desarrollo que puede ser medido de manera efectiva Las métricas pueden ser: de control o de predicción. Las métricas de control dan soporte a la gestión del proceso y las métricas de predicción ayudan a predecir características del software.

Ejemplo de métricas de proceso

El tiempo que lleva completar un proceso particular: calendar time, etc Los recursos requeridos para un proceso particular: cuántas personas/día, etc

El número de ocurrencias de un evento particular:

- Número de defectos promedio encontrados en una inspección
- Número de pedidos de cambio de requerimientos
- Número de líneas de código modificadas por pedidos en los requerimientos, etc.

Ejemplo de métricas de predicción o de producto

Están asociadas con el software en sí mismo: complejidad ciclomática, longitud promedio de los identificadores, número de atributos y operaciones que tiene un clase, etc.

Ambos tipos de métricas pueden influenciar en las decisiones del management Los managers usan mediciones de proceso para decidir si se debería cambiar el proceso métricas de predicción para decidir si el software tiene que ser cambiado o si está listo para ser liberado

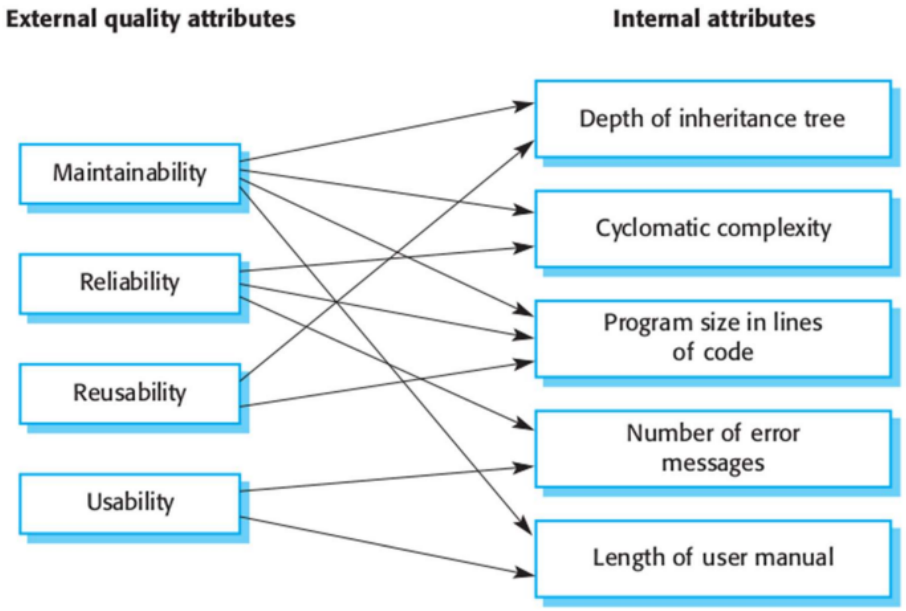
Métricas del Software

Las métricas del software deberían usarse de 2 maneras:

1. Midiendo características del sistema y agregándoles para evaluar atributos de calidad del sistema
2. Para identificar los componentes del sistema que están por debajo de los estándares.

Es difícil hacer una medición directa de los atributos de calidad del software (mantenibilidad, usabilidad, etc). Estos son atributos externos relacionados a cómo los desarrolladores y los usuarios experimentan el sistema. No pueden medirse objetivamente, pero se pueden medir atributos internos del software como el tamaño o la complejidad y asumir una relación entre ellos.

Relación intuitiva entre atributos internos y atributos externos de calidad



Condiciones para que una medida de un atributo interno pueda utilizarse como predictor de un atributo de calidad externo

El atributo interno debe poder medirse de manera precisa. Muchas veces se utilizan tools específicas, debe existir una relación entre el atributo interno que se mide y el atributo de calidad externo que se intenta predecir. La relación entre el atributo interno y el atributo de calidad externo debe ser entendida, validada y expresada en términos de una fórmula o modelo Identificar la forma funcional del modelo (lineal, exponencial, etc). Identificar los parámetros que se incluirán en el modelo, calibrar estos parámetros usando datos existentes, o se puede utilizar data-mining para descubrir relaciones y hacer predicciones sobre los atributos externos.

¿Por qué no es tan común el que se adopte un programa de métricas en una organización?

Es difícil cuantificar el ROI (retorno de inversión), los beneficios suelen ser a largo plazo, esto hace que las organizaciones sean reacias a invertir en métricas, ya que puede ser difícil demostrar un beneficio financiero inmediato. Además, algunas organizaciones pueden argumentar que han logrado mejorar la calidad del software en los últimos años sin necesidad de implementar un programa de métricas específico. Esto puede llevar a la percepción de que las métricas no son necesarias para lograr resultados positivos.

Introducir métricas puede requerir la implementación de herramientas especializadas, lo que puede aumentar la complejidad y los costos del proceso. Esto puede desalentar a algunas organizaciones, especialmente las más pequeñas, y con la adopción creciente de metodologías ágiles en el desarrollo de software, algunas métricas tradicionales pueden no encajar bien con estos enfoques. Las prácticas ágiles se centran en la entrega de software funcional de manera continua y la colaboración en equipo, lo que puede hacer que las métricas tradicionales sean menos relevantes.

Métricas de producto

Desempeñan un papel importante en la evaluación y mejora de la calidad del software, pero su aplicación y relevancia pueden variar según el contexto y los objetivos específicos del desarrollo de software. Estas métricas se dividen en dos categorías principales: métricas dinámicas y métricas estáticas.

Métricas dinámicas: Se recopilan midiendo un programa en ejecución. Suelen estar relacionadas con la eficiencia y la confiabilidad del software, ya que miden el comportamiento del programa en tiempo real. Por ejemplo, el número de errores o bugs reportados, el tiempo de respuesta del software y otros indicadores de

rendimiento en tiempo real que pueden relacionarse directamente con la eficiencia y la confiabilidad del software.

Métricas Estáticas: Se recopilan midiendo representaciones del sistema sin necesidad de ejecutar el programa. El análisis estático del código es una técnica común para recopilar métricas estáticas. Se utilizan para evaluar aspectos como la complejidad, la mantenibilidad y entendimiento del software. Métricas relacionadas con el diseño del software, la calidad del código y la documentación, el tamaño del programa o la complejidad del control pueden ayudar a predecir estos atributos de calidad, aunque la relación es más compleja y no siempre concluyente.

Estas son algunas métricas de rendimiento comunes utilizadas en ingeniería de software:

Tiempo de Respuesta (Response Time): Se refiere al tiempo que un sistema necesita para procesar una solicitud desde una perspectiva externa. En otras palabras, es el tiempo que un usuario o un sistema debe esperar para obtener una respuesta o resultado de una solicitud.

Tiempo de Reacción (Responsiveness): Se refiere a la velocidad con la que un sistema acepta o responde a una solicitud, incluso antes de completar todo el procesamiento de la solicitud. Es el tiempo que transcurre desde que se realiza una solicitud hasta que el sistema proporciona una confirmación o "acknowledge."

Latencia (Latency): Se refiere al tiempo mínimo requerido para obtener cualquier tipo de respuesta de un sistema. Puede variar según si el sistema es local (en la misma ubicación física) o remoto (accedido a través de una red).

Rendimiento (Throughput): Se refiere a la cantidad de trabajo que un sistema puede realizar en una unidad de tiempo determinada. Puede medirse en términos de bytes por segundo (por ejemplo, velocidad de transferencia de datos) o transacciones por segundo (por ejemplo, procesamiento de solicitudes).

Carga o Estrés (Load/Stress): Se utiliza para medir cuán estresado o cargado está un sistema en función de su capacidad para manejar múltiples usuarios o solicitudes simultáneas. Por lo general, se utiliza en el contexto de otras métricas de rendimiento, como el tiempo de respuesta. Por ejemplo, un sistema podría tener un tiempo de respuesta de 0.5 segundos con 10 usuarios y 2 segundos con 20 usuarios, lo que indica cómo responde el sistema bajo diferentes cargas.

Otras métricas de rendimiento:

Sensibilidad a la Carga (Load Sensitivity): Esta métrica expresa cómo varía el tiempo de respuesta de un sistema a medida que se incrementa la carga. Por ejemplo, un sistema A puede tener un tiempo de respuesta de 0.5 segundos con 10 usuarios y 2 segundos con 20 usuarios, lo que indica que su tiempo de respuesta aumenta significativamente con una mayor carga. Mientras que un sistema B podría tener un tiempo de respuesta de 0.2 segundos con 10 usuarios y 2 segundos con 20 usuarios, lo que muestra una mayor sensibilidad a la carga.

Eficiencia (Efficiency): La eficiencia se calcula dividiendo el rendimiento del sistema (por ejemplo, transacciones por segundo) entre los recursos utilizados para lograr ese rendimiento. Por ejemplo, el sistema A podría realizar 30 transacciones por segundo (tps) con 2 unidades centrales de procesamiento (CPUs), mientras que el sistema B realiza 40 tps con 4 CPUs. Ambos sistemas tienen un rendimiento similar, pero el sistema B utiliza más recursos para lograrlo.

Escalabilidad (Scalability): La escalabilidad mide cómo se ve afectada la performance de un sistema cuando se agregan nuevos recursos. Puede manifestarse de dos maneras:

- *Scale Up*: Agregar más o mejores recursos a un servidor existente para mejorar su rendimiento. Por ejemplo, aumentar la capacidad de memoria o CPU en un servidor.
- *Scale Out*: Agregar más servidores para distribuir la carga y aumentar la capacidad total. Por ejemplo, implementar una arquitectura en la nube o un clúster.

Capacidad (Capacity): La capacidad se refiere a la carga máxima o el máximo rendimiento que un sistema puede manejar de manera efectiva. Existe un punto en el que la performance se vuelve inaceptable debido a limitaciones de recursos o diseño. Identificar este punto es esencial para garantizar que el sistema pueda satisfacer las necesidades de la organización y los usuarios.

Un ejemplo: (filminas)

Recursos	Sistema A	Sistema B
1 servidor	20 tps	40 tps
2 servidores	35 tps	50 tps
3 servidores	50 tps	60 tps
4 servidores	65 tps	70 tps
5 servidores	80 tps	80 tps