GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

ENTORNOS DE DESARROLLO

UNIDAD 1. DESARROLLO DE SOFTWARE

1.3. METODOLOGÍAS ÁGILES

Departamento de Informática
Raúl Palao

Cicles Formatius

ÍNDICE

1 METODOLOGÍAS ÁGILES

Las metodologías ágiles en la ingeniería del software son enfoques de desarrollo de software que se basan en los valores y principios establecidos en el Manifiesto Ágil. Estas metodologías promueven la **flexibilidad, la colaboración, la entrega continua de software funcional y la capacidad de respuesta a los cambios en los requisitos del proyecto**.

1.1 Manifiesto ágil

El manifiesto ágil se refiere a un conjunto de valores y principios que guían la filosofía y las prácticas en el desarrollo de software y otros proyectos. El Manifiesto Ágil se originó en el mundo del desarrollo de software, pero sus principios se han extendido a diversas áreas, incluyendo la gestión de proyectos, el desarrollo de productos y más. Fue redactado por un grupo de expertos en desarrollo de software en **2001** y establece **cuatro valores y doce principios clave.**

1.1.1 Cuatro valores del manifiesto ágil

Los cuatro valores del Manifiesto Ágil son:

1. Individuos e interacciones sobre procesos y herramientas: Esto significa que se valora más la colaboración y la comunicación directa entre las personas involucradas en el proyecto que la adhesión estricta a procesos y herramientas formales.

2. Software funcionando sobre documentación extensiva: En lugar de crear una gran cantidad de documentación detallada, se da prioridad a la entrega de software funcional. La documentación sigue siendo importante, pero debe ser concisa y útil.

3. Colaboración con el cliente sobre negociación de contratos: Se enfatiza la colaboración continua con el cliente para comprender y satisfacer sus necesidades en lugar de enfocarse en la negociación de contratos detallados desde el principio.

4. Responder al cambio sobre seguir un plan: En lugar de aferrarse a planes rígidos, se valora la capacidad de adaptarse a cambios en los requisitos y circunstancias a medida que surgen.

1.1.2 Doce principios clave

Los 12 principios del Manifiesto Ágil son una ampliación de estos valores y proporcionan orientación adicional sobre cómo aplicar estos valores en la práctica. Algunos de los principios clave incluyen:

1. Satisfacer al cliente a través de la entrega temprana y continua de software valioso. En lugar de esperar hasta que todo el proyecto esté completo, se busca entregar partes valiosas del software de manera continua y frecuente.

2. Dar la bienvenida a los cambios en los requisitos, incluso en etapas avanzadas del desarrollo. La agilidad implica adaptarse a las necesidades cambiantes del cliente y estar dispuesto a realizar ajustes en cualquier momento.

3. Entregar software funcional con frecuencia, con una preferencia por entregas más cortas. La entrega continua de software permite obtener retroalimentación más rápido y ajustar el enfoque en consecuencia.

4. Colaborar con los clientes y las partes interesadas a lo largo del proyecto. La comunicación constante y la colaboración con los interesados ayudan a comprender y satisfacer sus necesidades.

5. Construir proyectos en torno a individuos motivados. Proporcionar a las personas el entorno y el apoyo que necesitan para hacer su trabajo y empoderarlos.

6. Utilizar conversaciones cara a cara como la forma más efectiva y eficiente de comunicación. La comunicación directa es valiosa para comprender los requisitos y resolver problemas de manera efectiva.

7. Entregar software que funcione es la medida principal de progreso. En lugar de enfocarse en documentación extensa o tareas completadas, el progreso se mide por la funcionalidad del software.

8. Mantener la simplicidad, maximizando la cantidad de trabajo no realizado. Evitar la adición innecesaria de características o funcionalidades complejas y centrarse en lo esencial.

9. Permitir que los equipos se autoorganicen. Los equipos deben tener la autonomía para tomar decisiones sobre cómo llevar a cabo su trabajo.

10. Reflejar de manera regular sobre cómo mejorar y ajustar el enfoque. La mejora continua es fundamental, y se deben realizar ajustes en función de la retroalimentación y la experiencia.

11. Alienta la sostenibilidad a largo plazo de los miembros del equipo. Evitar la sobrecarga de trabajo y mantener un ritmo constante de desarrollo.

12. Fomentar la atención técnica y una buena calidad del diseño. Invertir en prácticas técnicas sólidas para mantener la integridad del software a lo largo del tiempo.

1.2 Triángulo de Hierro

El Triángulo de Hierro, en el contexto de la ingeniería de software y la gestión de proyectos de desarrollo de software, se refiere a un concepto similar al Triángulo de Hierro en la gestión de proyectos en general, pero con un enfoque específico en los tres factores clave que afectan a un proyecto de software. Estos factores son:

- **Alcance**: Representa la funcionalidad y las características del software que se debe desarrollar. En otras palabras, son los requisitos y las expectativas del cliente en términos de lo que el software debe hacer.
- **Tiempo**: Este factor se relaciona con el cronograma del proyecto, es decir, el período de tiempo disponible para desarrollar y entregar el software.
- **Recursos** (Costos): Incluye los recursos humanos, financieros y técnicos necesarios para llevar a cabo el proyecto de desarrollo de software.

En el contexto de la ingeniería de software, la idea es que estos tres factores estén interconectados. Si se cambia uno de los elementos del Triángulo de Hierro, inevitablemente afectará a los otros dos. Algunos ejemplos de cómo esto se aplica en la ingeniería de software incluyen:

- Si se agrega más funcionalidad al alcance del proyecto, es probable que se requiera más tiempo y más recursos para completar el software, lo que podría aumentar los costos.

- Si se reduce el tiempo disponible para el proyecto, es posible que sea necesario reducir el alcance del software para cumplir con el nuevo plazo, o podría requerir más recursos para cumplir con el cronograma, lo que también aumentaría los costos.

- Si se reducen los recursos disponibles (por ejemplo, recortando el presupuesto), es posible que sea necesario reducir el alcance del proyecto o ampliar el tiempo necesario para desarrollar el software.

El Triángulo de Hierro en ingeniería de software es una herramienta útil para comprender las relaciones y restricciones entre alcance, tiempo y recursos en un proyecto de desarrollo de software. La gestión efectiva de estos factores es esencial para entregar proyectos de software con éxito y dentro de las restricciones establecidas.

2 EJEMPLOS DE METODOLOGÍAS ÁGILES

2.1 Extreme Programming (XP)

Extreme Programming (XP) es una metodología de desarrollo de software ágil que se centra en la mejora de la calidad del software y en la entrega de valor al cliente de manera rápida y continua. Fue creado por Kent Beck a **finales de la década de 1990** y se ha convertido en una de las metodologías ágiles más conocidas y utilizadas en la ingeniería de software. XP se caracteriza por una serie de prácticas y valores fundamentales:

Valores de Extreme Programming:

- **Comunicación:** Fomentar la comunicación efectiva y constante entre los miembros del equipo de desarrollo y los clientes.
- **Simplicidad**: Buscar la solución más simple que funcione para cada problema y evitar la complejidad innecesaria.
- **Retroalimentación:** Obtener retroalimentación continua de los clientes y las pruebas para guiar el desarrollo.
- **Coraje**: Tener la valentía de enfrentar los desafíos técnicos y las dificultades en el proceso de desarrollo.
- **Respeto:** Respetar a todos los miembros del equipo y sus contribuciones.

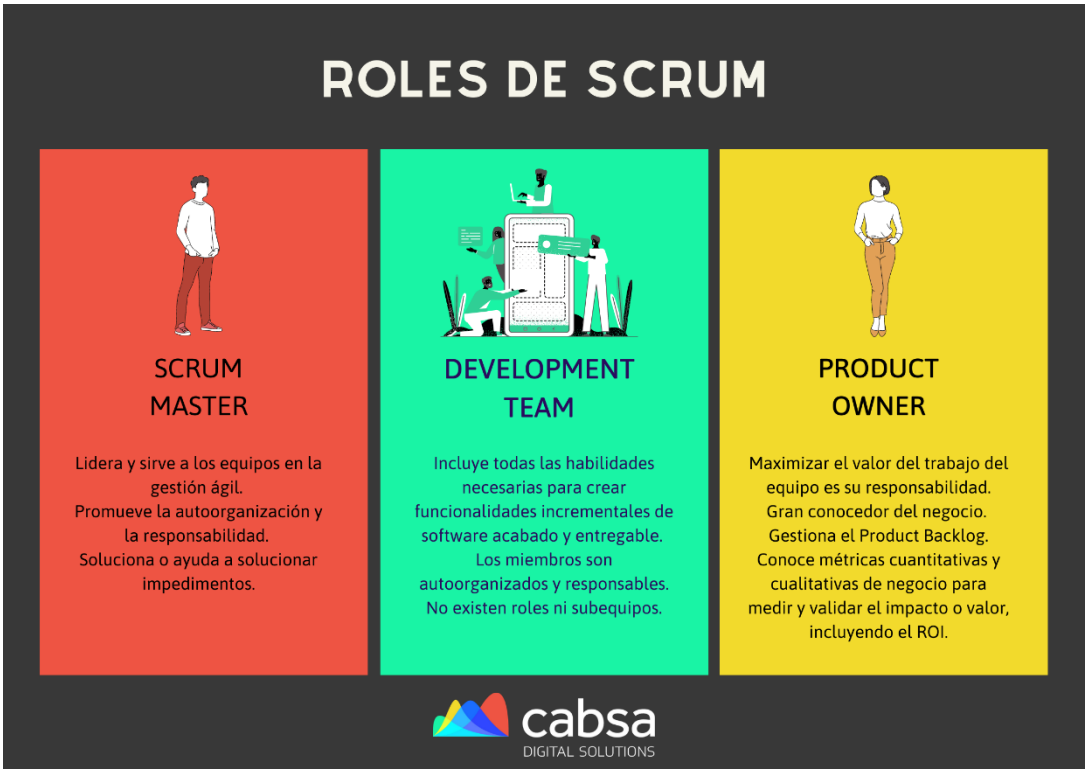
Imagen elaborada por oficinaproyectosinformatica.blogspot.com

2.2 Scrum

Scrum es un **marco de trabajo ágil** para la gestión de proyectos y el desarrollo de software. Se centra en la entrega de productos o proyectos de manera **iterativa e incremental** y se utiliza comúnmente en la industria del software, aunque también puede aplicarse a una variedad de otros campos. Scrum se basa en principios y roles claros, y utiliza una serie de eventos y artefactos para planificar, ejecutar y revisar proyectos de manera efectiva.

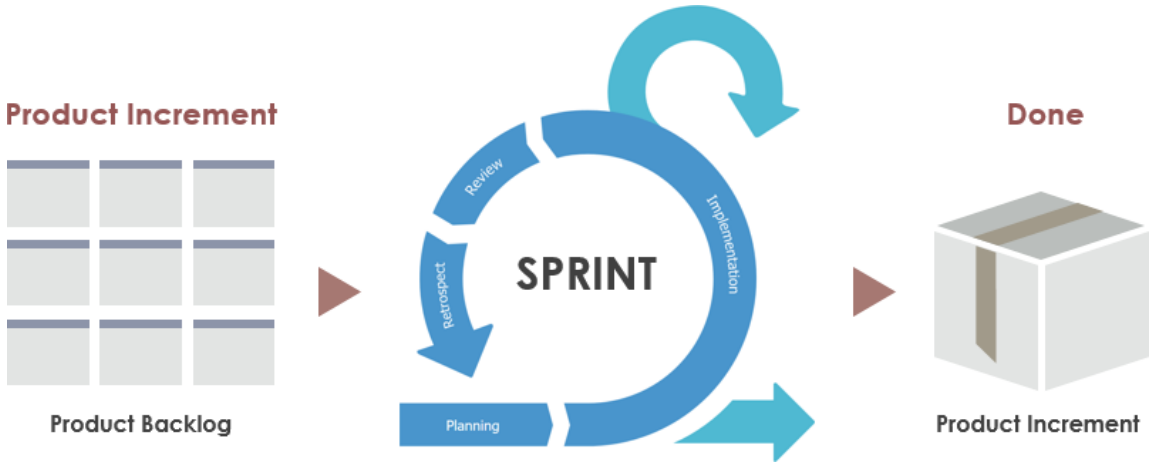
2.2.1 Roles

- **Product Owner:** Es el responsable de definir y priorizar los requisitos y características del producto. Representa los intereses del cliente y trabaja estrechamente con el equipo de desarrollo.
- **Scrum Master:** Actúa como un facilitador y líder del equipo Scrum. Su función principal es asegurarse de que el equipo siga los principios y prácticas de Scrum y elimine cualquier obstáculo que pueda estar impidiendo el progreso.
- **Equipo de Desarrollo:** Este es el equipo que trabaja en la implementación del producto. Es autoorganizado y multidisciplinario, y se compromete a entregar incrementos de producto al final de cada iteración (Sprint).



2.2.2 Conceptos

- **Product Backlog**: Es una lista priorizada de todas las características, requisitos y tareas que deben realizarse en el proyecto. Es propiedad del Product Owner.
- **Sprint Backlog:** Antes de comenzar un Sprint, el equipo selecciona elementos del Product Backlog y los coloca en el Sprint Backlog, que representa el trabajo que se realizará durante ese Sprint.
- **Incremento de Producto:** Es el resultado del trabajo completado al final de cada Sprint. Debe ser potencialmente utilizable, lo que significa que debería ser funcional y cumple con los estándares de calidad.



2.2.3 Eventos

- **Sprint:** Un Sprint es un período de tiempo fijo, típicamente de 2 a 4 semanas, durante el cual se desarrolla un conjunto de características o funcionalidades. Al final de cada Sprint, se debe entregar un incremento de producto potencialmente utilizable.
- **Reunión de Planificación del Sprint:** Al comienzo de cada Sprint, el equipo se reúne para seleccionar y comprometerse con las tareas a realizar durante ese Sprint.
- **Reunión Diaria de Scrum:** Es una breve reunión diaria en la que el equipo de desarrollo se actualiza mutuamente sobre el progreso, los obstáculos y las tareas para el día.

- **Revisión del Sprint:** Al final de cada Sprint, el equipo muestra el trabajo completado al Product Owner y a otras partes interesadas, obteniendo su retroalimentación.
- **Retrospectiva del Sprint:** Después de la Revisión del Sprint, el equipo realiza una retrospectiva para analizar lo que salió bien y lo que se puede mejorar en el próximo Sprint.

2.3 Kanban

Kanban es un método de **gestión y visualización del flujo de trabajo que** se originó en la industria manufacturera japonesa, específicamente en Toyota, pero se ha adaptado y aplicado en una amplia variedad de entornos, incluyendo el desarrollo de software, la gestión de proyectos, la fabricación, y más. El objetivo principal de Kanban es aumentar la eficiencia, mejorar la productividad y gestionar el trabajo de manera más efectiva. Algunos de los conceptos clave de Kanban son:

- **Visualización del trabajo:** En Kanban, se utiliza un tablero o **pizarra** para visualizar todo el trabajo que se está realizando. Cada tarea o elemento de trabajo se representa como una tarjeta o ficha, y estas tarjetas se mueven a través del tablero a medida que avanzan en el proceso. Esto permite una representación visual en tiempo real del progreso y el estado de todas las tareas.

- **Gestión del flujo:** El objetivo principal de Kanban es optimizar el flujo de trabajo y minimizar los cuellos de botella. **El trabajo se mueve a través del tablero** de manera constante y se ajusta según las necesidades, lo que permite una mayor adaptabilidad y respuesta a las demandas cambiantes.

2.4 Test-driven development

Test-Driven Development (TDD) es una metodología de desarrollo de software que se enfoca en escribir pruebas automatizadas antes de escribir el código real del programa. El proceso TDD generalmente sigue un ciclo repetitivo de tres pasos, conocido como el ciclo "Rojo-Verde-Amarillo":

- **Rojo (Red):** En esta fase, se escribe una prueba automatizada que describe el comportamiento deseado de una parte del software que aún no ha sido implementada. Esta prueba, en su estado inicial, fallará (mostrará un resultado "rojo") porque la funcionalidad aún no se ha desarrollado.

- **Verde (Green):** En esta etapa, el desarrollador implementa la funcionalidad necesaria para que la prueba automatizada pase y muestre un resultado "verde". El código se modifica o se escribe desde cero para cumplir con la prueba.

- **Amarillo (Refactor):** Después de que la prueba pasa (mostrando un resultado "verde"), el desarrollador puede mejorar el código existente o refinarlo (refactorizarlo) para hacerlo más limpio, eficiente y mantenible, sin cambiar su comportamiento.

Este ciclo se repite continuamente, con pruebas adicionales escritas antes de cada iteración. El enfoque de TDD es asegurarse de que cada parte del software esté respaldada por pruebas automatizadas que verifiquen su comportamiento. Esto tiene varios beneficios:

- **Calidad del código:** Las pruebas garantizan que el código cumpla con los requisitos y que no se introduzcan errores no deseados durante futuras modificaciones.

- **Documentación viva:** Las pruebas actúan como documentación viva del comportamiento del software. Otros desarrolladores pueden entender cómo se supone que funciona una parte del software al observar las pruebas asociadas.

- **Facilita el cambio**: Tener pruebas sólidas facilita la realización de cambios en el código sin temor a romper funcionalidades existentes, ya que las pruebas alertarán sobre posibles problemas.

TDD promueve la mejora continua y el desarrollo más confiable del software al fomentar la escritura de pruebas tempranas y la iteración constante. Se utiliza en conjunción con varias herramientas de pruebas unitarias y es una parte fundamental de la cultura de desarrollo ágil