TEMA 088. ANÁLISIS FUNCIONAL DE SISTEMAS, CASOS DE USO E HISTORIAS DE USUARIO. METODOLOGÍAS DE DESARROLLO DE SISTEMAS. METODOLOGÍAS ÁGILES.

Actualizado a 20/04/2023

1. ANÁLISIS FUNCIONAL DE SISTEMAS, CASOS DE USO E HISTORIAS DE USUARIO

El análisis es una actividad esencial en cualquier proyecto de desarrollo de software, ya que permite comprender a fondo el problema que se quiere resolver y establecer las bases para la creación de un diseño adecuado. Esta actividad implica aprender sobre las necesidades de los potenciales usuarios, identificar quién es realmente el usuario y entender todas las restricciones a la solución.

De acuerdo con Pressman, el modelo de análisis debe cumplir **tres objetivos principales**: describir lo que el cliente quiere, establecer una base para la creación de un diseño y definir un conjunto de requisitos. De esta forma, el equipo de desarrollo podrá comprender claramente las necesidades del cliente y los requisitos que deben cumplirse para satisfacer esas necesidades.

Las **tareas que comprende el análisis del sistema** son diversas y esenciales para asegurar el éxito del proyecto. Algunas de estas tareas incluyen la identificación de las necesidades y restricciones de coste y tiempo, la evaluación de la viabilidad del sistema desde el punto de vista técnico, económico y legal, la asignación de las funciones y compromisos al elemento hardware, software y el elemento humano, así como la especificación del sistema en sí mismo.

El análisis es una actividad fundamental en cualquier proyecto de desarrollo de software, ya que permite comprender a fondo el problema que se quiere resolver, establecer las bases para la creación de un diseño adecuado y definir un conjunto de requisitos esenciales para satisfacer las necesidades del cliente.

1.1. ANÁLISIS FUNCIONAL DE SISTEMAS

El análisis funcional de un sistema, según SWEBOK (Software Engineering Body of Knowledge), consiste en identificar los requisitos del mismo, describiendo QUÉ se va a desarrollar (aunque sin indicar CÓMO).

Los principales pasos que se llevan a cabo durante la ingeniería de requisitos son:

- **Educción, identificación o extracción de requisitos**: se obtienen las necesidades del cliente a partir de todas las fuentes de información disponibles.
- **Análisis de requisitos**: se analizan los requisitos obtenidos en la fase anterior para buscar conflictos, incoherencias, aspectos no resueltos etc. Después se clasifican, se evalúa su viabilidad y se integran los nuevos requisitos con los ya existentes. El objetivo final es lograr una lista de requisitos que defina las necesidades del cliente.
- **Representación de requisitos**: para ello se utilizan diferentes técnicas como el modelo Entidad Relación, casos de uso o diagramas de clases. Una vez que están los requisitos representados, es necesario que se reúnan los diversos participantes en el desarrollo para revisarlos y aprobarlos. El producto final con el que culmina esta fase es la Especificación de Requisitos Software, en donde está descrito con exactitud todo lo que el sistema debe hacer.
- **Validación de Requisitos:** se definen una serie de criterios y técnicas que permitirán, cuando el sistema esté construido, comprobar que cumple los requisitos.

El análisis funcional de sistemas es una actividad crítica en cualquier proyecto de desarrollo de software. Sin embargo, hay muchos problemas subyacentes que pueden dificultar su realización efectiva. Algunas de las causas más comunes incluyen una comunicación pobre entre los miembros del equipo, el uso de técnicas y herramientas inadecuadas y una tendencia a acortar la duración del análisis.

Para reducir estos problemas y la complejidad asociada, existen **tres técnicas principales que se pueden utilizar: la abstracción, la partición y la proyección**. La abstracción se utiliza para controlar la complejidad del problema y define la relación "general/específico" o "ejemplo de" entre objetos, funciones o estados en el dominio del problema. De esta manera, se pueden identificar patrones comunes y simplificar la comprensión del problema.

La partición es una técnica de descomposición que divide el problema en partes estructurales o funcionales más pequeñas. Esto ayuda a hacer el problema más manejable y a enfocarse en las partes críticas del sistema. Además, la partición también puede ayudar a detectar dependencias entre componentes y mejorar la modularidad del sistema.

Por último, la proyección es una técnica que tiene como objetivo "ver" un problema desde distintas perspectivas o puntos de vista. Esto puede ayudar a detectar problemas y soluciones que de otra manera no serían evidentes. Además, también puede ayudar a encontrar soluciones más adecuadas para los distintos usuarios y partes interesadas involucradas en el proyecto.

La Fase de Requisitos del Software se puede dividir en dos actividades principales: el análisis de requisitos del software y la especificación funcional.

El análisis de requisitos del software, también conocido como análisis del problema, se centra en comprender las necesidades del usuario y las limitaciones del sistema. Esta actividad implica la definición o especificación de requisitos y la modelización esencial, y su resultado final es el Documento de Requisitos del Sistema (DRS). Este documento describe con detalle los requisitos del sistema, tanto funcionales como no funcionales, y es utilizado como una guía para el diseño y la implementación del software.

La segunda actividad, la especificación funcional, se enfoca en la definición del comportamiento externo del sistema y la descripción del producto. Durante esta fase, se escribe la Especificación de Requisitos Funcionales o se define el comportamiento externo del sistema, lo que conduce a la creación del Documento de Diseño Funcional (DDF). Este documento describe la arquitectura del sistema, la funcionalidad y la interfaz de usuario, entre otros detalles relevantes.

Tanto el DRS como el DDF conforman el Documento de Especificación de Requisitos Software (ERS), que es una guía para el desarrollo del software y se utiliza para validar y verificar que el software cumpla con los requisitos establecidos. En resumen, la fase de requisitos del software es esencial para el éxito del proyecto y debe realizarse con la máxima atención y rigor.

Según Sommerville, los requisitos pueden clasificarse en:

- **Requisitos de sistema.**
    - Requerimientos Funcionales Abstractos. Son las funciones básicas que debe hacer el sistema a un nivel abstracto. Posteriormente, en cada uno de los subsistemas se concretan más.
    - Propiedades del Sistema: Propiedades no funcionales del sistema como por ejemplo disponibilidad, rendimiento, seguridad.
    - Características que no debe mostrar el sistema. En algunas ocasiones es importante especificar lo que el sistema no debe hacer.
- **Requisitos software.**
    - **Requerimientos Funcionales:** Son los requisitos que especifican el funcionamiento del software a construir.

- **Requerimientos No Funcionales:** Son los requisitos que especifican aspectos adicionales que no corresponden con el funcionamiento del sistema. Hacen referencia a las propiedades de éste como la fiabilidad, el tiempo de respuesta y la capacidad de almacenamiento. Pueden venir de las características requeridas del software (requisitos del producto), de la organización que desarrolla el software (requisitos organizacionales) o de fuentes externas

Las estrategias y técnicas de educción que se han desarrollado para la obtención de los requisitos se clasifican en:

- **Técnicas de alto nivel: relativas a entornos de trabajo.**
  - JAD (Joint Application Design) y JRP (Joint Requirements Planning) → métrica
  - Entorno de Bucles Adaptativo.
  - Prototipos.
  - Factores Críticos de Éxito.
  - Historias de usuario.
- **Técnicas de bajo nivel: permiten obtener detalles de una parte determinada del sistema.**
  - Entrevistas.
  - Brainstorming.
  - PIECES. (Performance, Information, Economy, Control, Efficiency, Services)
  - Casos de Uso.
  - Análisis de Mercado.

Revisar Técnicas y prácticas de Métrica v3

1.2. CASOS DE USO

El modelado de casos de uso es una técnica de obtención de requisitos que resulta adecuada para sistemas que están dominados por requisitos funcionales del usuario.

Los pasos que se llevan a cabo durante el modelado de casos de uso son:

1. Determinar el sujeto o **límite del sistema**, es decir, decidir qué es parte del sistema y qué es externo. Se dibuja como un cuadro, etiquetado con el nombre del sistema, con los actores dibujados fuera del límite y los casos de uso dentro.
2. Encontrar los **actores**: los actores representan roles de usuario o roles desempeñados por otros sistemas o hardware, y siempre son externos al sistema. Se suelen representar con la figura de hombre.
3. Encontrar los **casos de uso**: los casos de uso son secuencias de acciones que un sistema puede realizar al interactuar con actores externos. Se representan con un óvalo con el nombre del caso de uso en su interior.
4. Refinar hasta obtener un límite claro para el sistema, y un conjunto estable de actores y casos de uso.

Permite capturar los requisitos funcionales y expresarlos desde el **punto de vista del usuario**. Es una secuencia de acciones realizadas por el sistema, que actúan como cajas negras para el usuario.

Elementos que forman parte del diagrama de Casos de Uso:

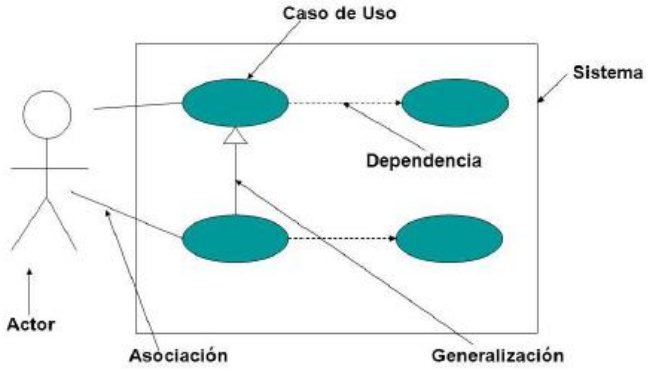
- **Actores:** Entidad externa al sistema que realiza algún tipo de interacción con el mismo.
- **Caso Uso:** Secuencia acciones realizada por Sistema que produce resultado. Requisito Funcional.

- **Relaciones entre actores y casos de uso:** de comunicación o solicitud.
- **Relaciones entre casos de uso:**
    - **<<extiende/extend>>:** u opcionales un caso de uso extiende o amplía a otro.
    - **<<usa/include>>:** un caso de uso "usa" a otro. Es decir, lo incluye siempre.
- **Contexto del sistema (System Boundary):** Cuadro que contiene las diferentes partes del sistema y los casos de uso que la forman.

Tipos de Casos de Uso:

- **Primarios:** representan los procesos comunes más importantes.
- **Secundarios:** representan procesos menores o raros.
- **Opcionales:** representan procesos que pueden no abordarse o que sólo se harán en caso de tener tiempo y recursos.

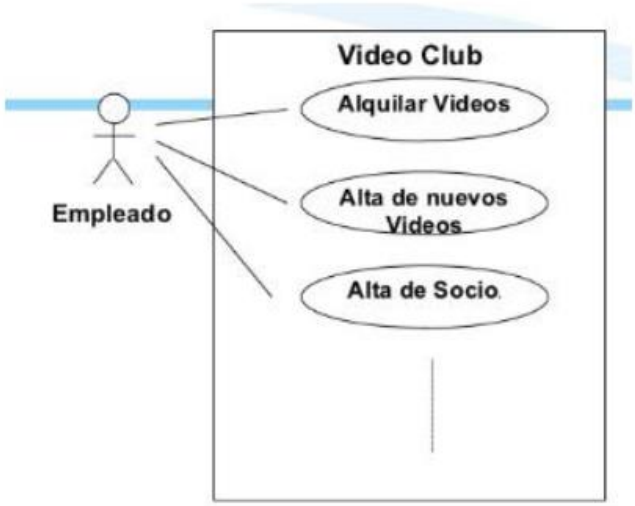
A continuación, se puede observar la representación gráfica de los elementos del diagrama de casos de uso.



¿Cómo hacerlo? Se determina el límite del sistema, con un cuadro con el nombre del sistema, los actores fuera y los casos de uso dentro del cuadro. Hay que:

- Definir actores. Representan los roles de usuarios. Son externos al sistema.
- Encontrar casos de uso. Secuencia de acciones del sistema que interactúan con actores externos.
- Refinar hasta tener conjunto de actores y casos de uso estables.

Un ejemplo de diagrama de casos de uso para un sistema de gestión de un videoclub es el siguiente:



Los casos de uso representados en el diagrama habrán de especificarse de modo que se obtenga una descripción de los pasos necesarios para completar la funcionalidad indicada en el caso de uso. A continuación, se muestra un ejemplo de especificación para el caso de uso "Alta de socio":

Nombre del caso de uso	Alta de socio
Descripción	Dar de alta a un socio en el videoclub
Actores	Empleado del videoclub
Precondiciones	El empleado se ha autenticado en el sistema
Flujo principal	1. El empleado del videoclub solicita al sistema comenzar el proceso de alta de un nuevo socio. 2. El sistema solicita los siguientes datos del nuevo socio: DNI, nombre, apellidos, fecha de nacimiento, sexo, dirección y teléfonos de contacto. 3. El empleado introduce los datos del socio y solicita al sistema que los almacene. 4. El sistema almacena los datos, imprime el carnet de socio e informa al empleado de que el proceso ha terminado con éxito.
Postcondiciones	El socio ha sido dado de alta en el sistema
Flujos alternativos	Si en el paso 3 el sistema detecta que el socio ya existe, muestra un mensaje informando de esta situación. Si en el paso 3 el sistema detecta que hay datos obligatorios que no han sido introducidos, muestra un mensaje informando de esta circunstancia.

ALGUNOS CONCEPTOS IMPORTANTES EN EL MODELADO DE CASOS DE USO

Generalización

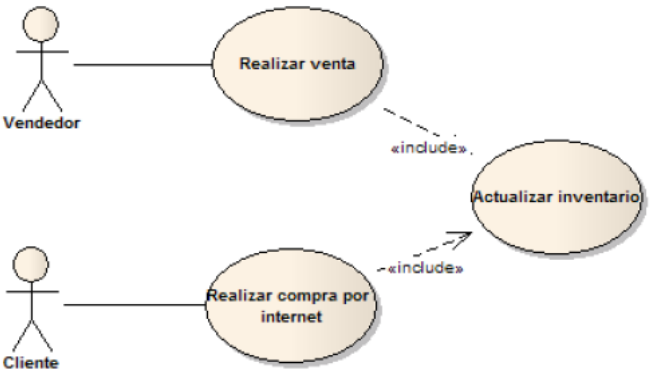
Generalización de actores: se produce cuando tenemos actores que heredan roles de su/s actor/es padre.

En el ejemplo que se ve a continuación, el actor "Funcionario habilitado" puede llevar a cabo todas las acciones que desarrollan tanto los "funcionario tramitadores" como los "ciudadanos".

Generalización de casos de uso: los casos de uso hijos heredan las características del caso de uso padre. Si el flujo del caso de uso padre está incompleto, se dice que es un caso de uso abstracto.

Relaciones

Include: permite incluir el comportamiento de un caso de uso en el flujo de otro caso de uso. El caso de uso base se ejecuta hasta que se alcanza el punto de inclusión, luego la ejecución pasa al caso de uso de inclusión. Una vez termina el caso de uso de inclusión, el flujo regresa de nuevo al caso de uso base.



Extend: mediante la relación extend, el caso de uso base proporciona un conjunto de puntos de extensión que son enganches donde se puede añadir nuevo comportamiento. Esta relación se suele utilizar para representar comportamiento que puede ser añadido al caso de uso en ciertas circunstancias, por ejemplo en la siguiente figura el caso de uso de extensión “Autorizar pago con tarjeta” solo se activa si el cliente paga con tarjeta, algo que no siempre ocurre, pues si paga en efectivo solo se activa el caso base “Realizar venta”.

1.3. HISTORIAS DE USUARIO

Las historias de usuario se suelen utilizar en las metodologías ágiles para obtener los requisitos. Representan los requisitos mediante frases simples que utilizan el lenguaje común del usuario.

El origen de las historias de usuario se remonta a la metodología de desarrollo ágil XP (eXtreme Programming). Se utilizan de forma habitual en todas las metodologías de desarrollo y frameworks ágiles para escribir los requisitos y las pruebas de validación.

Siguen la plantilla, según Mike Cohn: **COMO <rol> QUIERO <algo> PARA PODER <beneficio>**

Las historias de usuario se pueden escribir con diferentes niveles de detalle. Las grandes historias de usuario se conocen como épicas (**epics**), y puesto que serán demasiado largas como para poder ser completadas en una iteración, se pueden dividir en historias de usuario más pequeñas.

Las historias de usuario se componen de **3 partes** (conocidas como CCC):

- La descripción escrita, la cual se utiliza como recordatorio de una funcionalidad que está pendiente de implementar (**CARD**).
- Conversaciones sobre la historia que permiten refinar sus detalles (**CONVERSATION**).
- Criterios de aceptación que permiten determinar cuando la historia está completa (**CONFIRMATION**).

Las historias de usuario tienen las siguientes **características** (características INVEST):

- **Independent**: independientes unas de otras.
- **Negociable**: la historia en sí misma no es lo suficientemente explícita, por lo que es necesario un debate con los usuarios para esclarecer su alcance, el cual debe explicitarse mediante las pruebas de validación.
- **Valuable**: que tengan valor para los usuarios, que representen algo importante para alguno de ellos.
- **Estimable**: generalmente la estimación de una historia estará entre 10 días y 2 semanas.
- **Small**: que no sean demasiado largas, ya que eso dificulta la estimación.
- **Testable**.

Ventajas de las historias de usuario:

- Representan requisitos que pueden implementarse rápidamente.
- Permiten mantener una relación cercana con el cliente.
- Permite dividir los proyectos en pequeñas entregas.
- Facilitan la estimación.
- Son adecuadas para proyectos con requisitos volátiles o no muy claros.

Se pueden crear inicialmente historias de usuario de alto nivel para posteriormente ser reemplazadas por historias con mayor nivel de detalle.

Criterios de aceptación

Un criterio de aceptación es el criterio por el cual se define si una historia de usuario fue desarrollada según la expectativa del Product Owner y si se puede dar como hecha.

Deben ser definidos lo antes posible porque complementan la historia de usuario, ayudan al equipo de desarrollo a entender mejor cómo se espera que el producto se comporte y facilitan la estimación.

Además, sirven de guía para el desarrollo de los test. Deben cubrir los casos de uso positivos o comunes (el trayecto feliz) pero también los casos extremos o "corner case", por ejemplo, en una aplicación para grabar vídeos, el usuario está grabando un vídeo y se queda sin batería.

Las calidades de un criterio de aceptación eficaz se definen bajo el método SMART. SMART significa Specific (Especifico), Measurable (Medible), Achievable (Alcanzable), Relevant (Relevante) y Time-bound (Temporalmente limitado).

Existen dos técnicas para escribir criterios de aceptación:

- Técnica de comportamiento: se consigue una estructura consistente que se trasladará fácilmente a test automáticos: DADO <condiciones> CUANDO <evento> ENTONCES <resultado>.

- Técnica de escenarios:

  Suele definir "el trayecto feliz" y un trayecto alternativo de la funcionalidad y debe describir cómo el usuario ejecutaría o intentaría ejecutar los diferentes pasos en dichos trayectos. Al restringirse solo a la forma en que el usuario procedería y el sistema correspondería, elimina toda la información innecesaria.

Historias de usuario	Casos de uso
Más cortas, y por lo tanto estimables (normalmente entre 10h y 2 semanas) Ayudan a la estimación	**Más largos**. En muchas ocasiones es difícil dar una estimación para el caso de uso completo.
Se **pueden ir refinando y completando**. Inicialmente no son necesarios los detalles. Éstos se pueden ir añadiendo cuando se vaya a implementar la funcionalidad.	Debería **estar completa desde el principio**.
Se suelen desechar una vez finaliza la iteración en la que fueron implementadas.	Perviven en el tiempo (durante desarrollo y mantenimiento).
Están **incompletas sin las pruebas de validación.**	
Los escenarios alternativos se expresan como diferentes condiciones en las pruebas de validación.	Los escenarios alternativos se expresan en la sección de flujos alternativos.
Pueden especificar **requisitos no funcionales.**	**Siempre se refieren a requisitos funcionales**

2. METODOLOGÍAS DE DESARROLLO DE SISTEMAS

La ingeniería del software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software. Para lograr esto, se utilizan **metodologías** que son conjuntos de procedimientos, métodos y técnicas que ayudan a los desarrolladores a construir nuevo software de manera efectiva. Estas metodologías también incluyen herramientas que permiten la automatización de métodos y técnicas, así como un soporte documental que ayuda a los desarrolladores a seguir los pasos necesarios para construir el software.

Existen **ciertos requisitos que deben cumplir las metodologías**, como estar documentadas, ser repetibles, enseñables, estar basadas en técnicas probadas, ser validadas y ser apropiadas para el tipo de software que se está construyendo.

Es importante **distinguir entre una metodología y un ciclo de vida del software**. Una metodología puede seguir uno o más ciclos de vida y un ciclo de vida no es exclusivo de una metodología, por lo que varias metodologías pueden seguir el mismo ciclo de vida. Una metodología define los pasos, productos, técnicas y métodos para seguir, mientras que un ciclo de vida define etapas y actividades por las que pasa el desarrollo de un sistema.

Algunos tipos de metodologías:

- Para Sistemas de Tiempo Real. Permiten gestión de procesos concurrentes, manejo de interrupciones y prioridades, … En realidad son ampliaciones y modificaciones de otras metodologías.
- Metodologías estructuradas. Orientadas a datos (o función) o procesos. SSADM, Merise, Metrica 3, etc.
- Metodologías OO. Fomenta la reutilización y permite una forma de trabajo dinámica dado el alto grado de iteración y solapamiento. OOAD – BOOCH, OMT-Rumbaugh, OOSE- Jacobson. Estas 3 forman **RUP** (Rational Unified Process) o PUDS, que se puede considerar una evolución de las tres anteriores.
- Metodologías para un Dominio Específico. Por ejemplo, herramientas CASE y BPM.
- Metodologías Ágiles. Scrum, XP, Kanban, FDD…

3. METODOLOGÍAS ÁGILES

Las **metodologías ágiles de desarrollo de software** son un enfoque iterativo e incremental para la gestión y el desarrollo de proyectos de software. En lugar de planificar todo el proyecto de antemano, las metodologías ágiles se centran en entregar iterativamente pequeñas partes del proyecto a lo largo del tiempo, en ciclos de desarrollo cortos llamados "iteraciones" o "sprints".

Los equipos de desarrollo ágil trabajan en estrecha colaboración con los stakeholders y los clientes para identificar y priorizar los requisitos y características más importantes del software a desarrollar. Los miembros del equipo tienen roles y responsabilidades claramente definidos y trabajan juntos para garantizar la calidad del producto final.

En 2001, 17 desarrolladores reunidos en Snowbird (Utah, USA), publicaron el llamado "manifiesto ágil", que anuncia:

"Estamos poniendo al descubierto mejores métodos para desarrollar software, haciéndolo y ayudando a otros a que lo hagan. Con este trabajo hemos llegado a valorar:

- *A los **individuos y su interacción**, por encima de los procesos y las herramientas.*
- *El **software que funciona**, por encima de la documentación exhaustiva.*
- *La **colaboración con el cliente**, por encima de la negociación contractual.*
- *La **respuesta al cambio**, por encima del seguimiento de un plan.*

Aunque hay valor en los elementos de la derecha, valoramos más los de la izquierda."

Las metodologías ágiles se basan en los valores y principios del **Manifiesto Ágil**, que incluyen la satisfacción del cliente a través de entregas tempranas y continuas, la colaboración entre los miembros del equipo y los stakeholders, la respuesta al cambio en lugar de seguir un plan rígido, y el enfoque en individuos y su interacción por encima de los procesos y herramientas.

Se desarrolla en los siguientes 12 Principios:

1. Nuestra principal prioridad es **satisfacer al cliente** a través de la entrega temprana y continua de software de valor.
2. Son **bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo**. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente.
3. **Entregar con frecuencia software que funcione**, en periodos de un par de semanas hasta un par de meses, con preferencia en los períodos breves.
4. **Las personas del negocio y los desarrolladores deben trabajar juntos** de forma cotidiana a través del proyecto.
5. Construcción de proyectos en torno **a individuos motivados**, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realicen la tarea.
6. La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la **conversación cara a cara**.
7. El **software que funciona** es la principal medida del progreso.
8. Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un **ritmo constante de forma indefinida**.
9. **La atención continua a la excelencia** técnica enaltece la agilidad.
10. La **simplicidad** como arte de maximizar la cantidad de trabajo que no se hace, es **esencial.**
11. **Las mejores arquitecturas, requisitos y diseños emergen de** equipos que se auto-organizan.
12. En intervalos regulares, el **equipo reflexiona** sobre la forma de ser más efectivo y ajusta su conducta en consecuencia.

3.1. SCRUM

SCRUM (*melé*, en Ruby) es un conjunto de prácticas y recomendaciones para el desarrollo de un producto, basada en un proceso iterativo e incremental y que promueven la comunicación, el trabajo en equipo y el feedback rápido sobre el producto construido.

Sus objetivos principales son:

- Mostrar producto construido con frecuencia para obtener feedback por parte del usuario mediante entregas tempranas y planificadas.
- Conseguir equipos de alto rendimiento

Su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos, que pretende resolver el problema que producen los cambios en los requerimientos (desde la fase de análisis a la puesta en producción) y estimaciones de tiempos, costes y funcionalidades, muchas veces poco realistas al ser complicado su predicción al comienzo de los proyectos.

Especialmente indicado para:

- Proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos.
- Se utiliza para resolver situaciones en que no se está entregando al cliente lo que necesita, cuando las entregas se alargan demasiado, los costes se disparan o la calidad no es aceptable.

SCRUM define un conjunto de prácticas y roles

ROLES

El conjunto de roles participantes en el proceso puede dividirse en dos grandes grupos: cerdos (los comprometidos) y gallinas (implicados). El nombre de los grupos está inspirado en la siguiente frase:

En un plato de huevos con tocino el cerdo está comprometido, la gallina sólo está involucrada.

Los 'cerdos' están comprometidos a desarrollar el software de forma regular y frecuente

Las 'gallinas' sólo están interesadas en el proyecto. Si falla, fueron los cerdos los que se comprometieron.

Las necesidades, deseos, ideas e influencias de los roles 'gallina' se tienen en cuenta, pero no de forma que pueda afectar, distorsionar o entorpecer el proyecto Scrum.

Roles "Cerdo"

Son aquellos comprometidos con el proyecto y el proceso Agile, que se encargan de realizar las tareas necesarias para que el producto obtenga el valor necesario.

Product Owner (propietario del producto): representa la voz del cliente, es el responsable identificar las funcionalidades del producto, priorizando, decidiendo cuáles deberían ir al principio de la lista para el siguiente Sprint, y repriorizando y refinando continuamente la lista. Se asegura de que el equipo Scrum trabaja de forma adecuada desde la perspectiva del negocio. Sus funciones son:

- Representar a todos los interesados en el proyecto.
- Definir la visión y objetivos del proyecto.
- Colaborar con el equipo en el desarrollo y aceptar de manera regular el producto.
- Dirigir resultados y maximizar el ROI del equipo de desarrollo.

ScrumMaster (o Facilitador): no es el líder del equipo (porque ellos se auto-organizan), sino que actúa como una protección entre el equipo y cualquier influencia que le distraiga. El ScrumMaster es el responsable de la correcta implantación, seguimiento y utilización de Scrum en la organización y en el equipo de trabajo. Sus funciones:

- Asegurar el seguimiento del proceso.
- Difundir los valores de scrum.
- Gestionar los riesgos y problemas que el equipo se va encontrando.
- Proteger al equipo de interrupciones.

ScrumTeam o Equipo: Es el responsable de la construcción del producto siendo multi-funcional (tiene todas las competencias y habilidades necesarias para entregar un producto operativo en cada *sprint*) y auto-organizado (con un alto grado de autonomía y responsabilidad). Es un pequeño equipo de 5 a 9 personas con las habilidades transversales necesarias para realizar el trabajo (diseñador, desarrollador, etc). Sus funciones:

- Convertir el Product backlog en un producto potencialmente entregable en cada sprint
- Asegurar la calidad del producto
- Hacer las demostraciones del trabajo realizado
- Mejora continua

Roles "Gallina".

Roles implicados en el proceso Scrum, que no son parte del proceso, pero deben tenerse en cuenta, ya que su participación es muy valiosa para el correcto feedback del producto. Son los expertos del negocio y otros interesados (stakeholders) para quienes el proyecto producirá el beneficio acordado que lo justifica. Sólo participan directamente durante las revisiones del sprint.

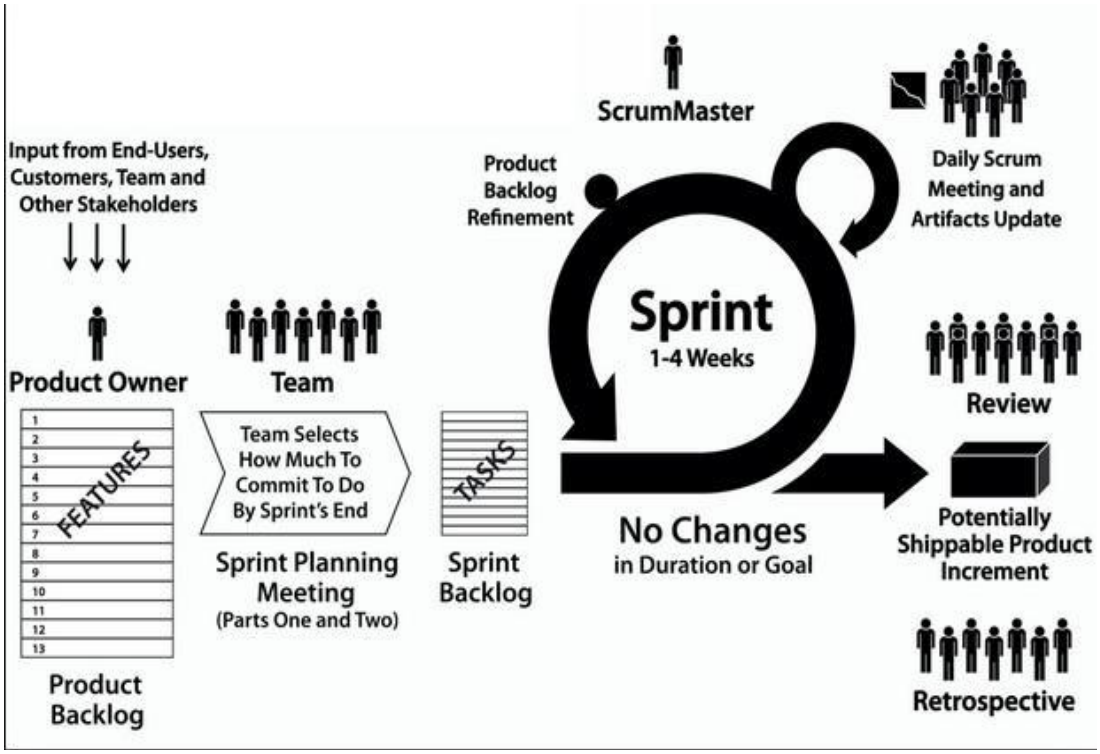
PRÁCTICAS

Durante cada sprint, un periodo entre 15 y 30 días, el equipo crea un incremento de software potencialmente entregable (utilizable). El conjunto de características que forma parte de cada sprint viene del Product Backlog, que es un conjunto de requisitos de alto nivel priorizados que definen el trabajo a realizar. Los elementos del Sprint Backlog se determinan durante la reunión de Sprint Planning. De este conjunto, el Product Owner identificará los elementos del Product Backlog que quiere ver completados y los hace del conocimiento del equipo. Entonces, el equipo determina la cantidad de ese trabajo que puede comprometerse a completar durante el siguiente sprint.

Durante el sprint, nadie puede cambiar el Sprint Backlog, lo que significa que los requisitos están congelados durante el sprint.

Un principio clave de Scrum es el reconocimiento de que durante un proyecto los clientes pueden cambiar de idea sobre lo que quieren y necesitan (a menudo llamado requirements churn).

El product backlog contiene descripciones genéricas de todos los requisitos, funcionalidades deseables, etc. priorizadas según su retorno sobre la inversión (ROI). Primero se realizarán aquellos requisitos más interesantes.



El proceso de desarrollo es el siguiente:

1. Al principio del proyecto se realiza el Sprint 0, con el objetivo de identificar los roles y compartir entre todos la misma visión y objetivos del proyecto.
   Las tareas de esta fase son:

- Realizar los preparativos necesarios para comenzar el proyecto, tanto organizativos como tecnológicos.
- Elaborar y priorizar la lista de requerimientos del producto a desarrollar, en forma de épicas e historias de usuarios, creando el producto backlog (la lista de requerimientos clasificados según su importancia), la cual es responsabilidad del Product Owner.
- Establecer el definition of ready (cuándo cualquier Historia de Usuario está lista y suficientemente definida para entrar en el Sprint Backlog) y definition of done (cuándo cualquier Historia de Usuario ha sido completada)

2. En el Sprint Planning Meeting, el Product Owner, durante la primera parte de la reunión, explica al Scrum Team los objetivos del proyecto y defiendo su backlog, para pasar a una segunda fase en la que el Scrum Team seleccionará los elementos que se desarrollaran en el siguiente sprint (de entre los de mayor prioridad) y construyen una primera planificación. Los elementos del backlog elegidos se descomponen en tareas, constituyendo el Sprint Backlog.
3. Se comienza en sprint, durante el cual no se podrán añadir nuevas tareas. De modo diario se celebran las Daily Scrum.
4. Al final del sprint se realizarán las tareas de cierre del sprint definidas en SCRUM:
    - Actualizar Product Backlog.
    - Sprint Review.
    - Entrega Documentación del sprint.
    - Planning meeting.
    - Aceptación.
    - Sprint Retrospective.

5. El proceso vuelve a comenzar con los elementos del producto backlog que quedan por hacer.

OTROS ELEMENTOS

Estos elementos conforman el conjunto de instrumentos que han de manejar los distintos roles de un equipo scrum.

Product Backlog: Es lista que contiene los requerimientos funcionales del producto que deberán desarrollarse a través de las sucesivas iteraciones.

Product Backlog Items: Son los objetivos, funcionalidades o requisitos de alto nivel que constituyen una pieza en el plan de producto (Product Backlog). Se escriben en forma de Historias de Usuario

Sprint**:** Es cada una de las iteraciones en Scrum. En cada proyecto, los sprints deberán tener una **duración fija** (no se debería modificar durante la vida del proyecto), recomendando que no sean inferiores a una semana ni superiores a cuatro semanas.

Sprint Backlog: Es un conjunto de historias de usuario que se seleccionan del Product Backlog durante la reunión de Sprint Planning, para ser abordadas durante un sprint.

Burn down Chart (Gráfica de "quemado") – de dos tipos: backlog: descendente, product: puede variar. La burn down chart es una gráfica mostrada públicamente que mide la cantidad de requisitos en el Backlog del proyecto pendientes al comienzo de cada Sprint. Progreso del proyecto, línea sea descendente hasta llegar al eje horizontal (proy. terminado). Si durante el proceso se añaden nuevos requisitos la recta tendrá pendiente ascendente en determinados segmentos, y si se modifican algunos requisitos la pendiente variará o incluso valdrá cero en algunos tramos (se refiere al "product backlog).

Reuniones

- **Sprint Planning Meeting**: se realiza al comienzo del sprint. El objetivo final es consensuar el Sprint Backlog.

  Los participantes:

  - Product Owner: Prioriza las historias de usuario más prioritarias y se las presenta al equipo
  - Equipo de desarrollo: Descompone las historias de usuario en tareas, estima las mismas e indica cuales entrarán en el sprint que va a comenzar
  - Scrum Master: Colabora con el equipo en la decisión de las tareas a realizar en el sprint y se asegura de que todo esté claro para poder comenzar.

  Duración máxima de 2 horas por cada semana de sprint

- **Daily scrum**: reunión diaria. entre todos los miembros del equipo. En la reunión, cada miembro del equipo explica al resto:
  - Qué tareas ha hecho desde la última reunión.
  - Qué tareas va a hacer hasta la próxima reunión.
  - Qué impedimentos tiene o prevé encontrar.

Los participantes (todos pueden asistir, pero solo los roles “cerdo” pueden hablar):

  - Scrum Master: Se encarga de recopilar los posibles impedimentos para tratar de solucionarlos y supervisa que la reunión siga los estándares de Agile (No más de 15 minutos).
  - Equipo de desarrollo: Indicará su evolución como se ha explicado en los objetivos
  - Product Owner: Su participación no es obligatoria, si bien es muy útil para estar al tanto de los impedimentos del equipo o de los posibles cuellos de botella.

- **Sprint Review**: revisión final del sprint cuyo objetivo es verificar si el incremento de producto satisface las expectativas del Product Owner e identificar si hay que hacer ajustes. Se trata de una demostración del producto realizado. Participantes:

  - Product Owner
  - Scrum Master
  - Equipo de desarrollo

  Duración máxima 1 hora por cada semana de sprint.

- **Sprint Retrospective**: Al final de cada sprint, asiste el equipo de desarrollo. El objetivo es que el equipo reflexione sobre la forma en que han trabajado durante el sprint que acaba e identifique acciones de mejora para siguientes iteraciones. Participantes:

  - Scrum Master: Encargado de gestionar la reunión y actuar como moderador.
  - Product Owner
  - Equipo de desarrollo

  Duración máxima 1 hora por cada semana del sprint.

La Guía Scrum tiene como objetivo ayudar a las personas de todo el mundo a entender Scrum. Esta guía contiene la definición de Scrum. Cada elemento del marco sirve a un propósito específico que es esencial para el valor global y los resultados realizados con Scrum.

La primera guía Scrum se publicó en 2010 y en el año 2020 se ha llevado a cabo su última revisión. En esta se han realizado cambios importantes en la terminología. La última actualización hace de Scrum un marco de trabajo **más real,** reduciendo la complejidad y brindando mayor claridad en el propósito de sus diferentes elementos. Entre los principales cambios cabe destacar:

- Los “Roles” se sustituyen por “Responsabilidades”.
- Se sustituye el 'Development Team’ por el término ‘Developers’.
- Al ‘Product Backlog’, ‘Sprint Backlog’ e 'Incremento', se le asocia un compromiso específico: ‘Product Goal’, ‘Sprint Goal’ y ‘Definition of Done’.
- Entre las prácticas que se han eliminado se encuentran las «3 preguntas» de la Daily Meeting.

El lenguaje de la guía se ha simplificado para alcanzar una mayor audiencia. Se han eliminado las referencias al software de forma que Scrum sea apropiado con equipos que realizan otras formas de trabajo complejo.

3.2. KANBAN

Kanban se basa en un sistema de producción que dispara trabajo solo cuando existe capacidad para procesarlo. Es una aproximación al proceso gradual, evolutivo y al cambio de sistemas para las organizaciones.

El disparador de trabajo es representado por *tarjetas kanban* de las cuales se dispone de una cantidad limitada. En el desarrollo de Software, Kanban fue introducido por David Anderson de la Unidad de Negocios XIT de Microsoft, en 2004, reemplazando el sistema de tarjetas por un tablero visual

Kanban puede ser también considerado como un método de organización del trabajo, y puede utilizarse como complemento de otras metodologías.

Utiliza un sistema de extracción limitada del trabajo en curso como mecanismo básico para exponer los problemas de funcionamiento del sistema (o proceso) y estimular la colaboración para la mejora continua del sistema. Un ‘tablero’ simple consistiría en 3 columnas: To-Do, Doing y Done, en las que se van colocando las tareas según la fase en la que estén. Esto permitirá observar los cuellos de botella (constraints).

Demuestra ser una de las metodologías adaptativas que menos resistencia al cambio presenta.

Este proceso de producción, donde un trabajo se introduce al sistema solo cuando existe disponibilidad para procesarlo, se denomina pull (tirar) en contrapartida al mecanismo push (empujar), donde el trabajo se introduce en función de la demanda.

A través del conocimiento aportado por la industria, podemos destacar 4 principios del método Kanban:

1. Comience con lo que hace ahora: El método Kanban se inicia con las funciones y procesos que ya se tienen y estimula cambios continuos, incrementales y evolutivos a su sistema.
2. Se acuerda perseguir el cambio incremental y evolutivo. La organización (o equipo) deben estar de acuerdo que el cambio continuo, gradual y evolutivo es la manera de hacer mejoras en el sistema y debe apegarse a ello. Los cambios radicales pueden parecer más eficaces, pero tienen una mayor tasa de fracaso debido a la resistencia y el miedo en la organización. El método Kanban anima a los pequeños y continuos cambios incrementales y evolutivos a su sistema actual.
3. Respetar el proceso actual, los roles, las responsabilidades y los cargos. Tenemos que facilitar el cambio futuro; acordando respetar los roles actuales, responsabilidades y cargos, eliminamos los temores iniciales. Esto nos debería permitir obtener un mayor apoyo a nuestra iniciativa Kanban (menor resistencia al cambio).
4. Liderazgo en todos los niveles. Se debe alentar hechos de liderazgo en todos los niveles de la organización de los contribuyentes individuales a la alta dirección.

En su aplicación al mundo del SW, según Anderson se basa en 5 Reglas:

Regla #1: Mostrar el proceso (o flujo de trabajo), consiste en la visualización de todo el proceso de desarrollo públicamente asequible. El objetivo es entender mejor el proceso, exponer sus problemas de funcionamiento y estimular la colaboración de los participantes.

Regla #2: Limitar el trabajo en curso (WIP≡Work In Progress), consisten en acordar anticipadamente la cantidad de ítems que pueden abordarse por cada proceso, con el objetivo de detectar los cuellos de botella.

Regla #3: Optimizar el flujo de trabajo, el objetivo es la producción estable, continua y previsible. Midiendo el tiempo que el ciclo completo de ejecución del proyecto demanda para tratar de minimizarlo, maximizando además el WIP por unidad de tiempo.

Regla #4: Hacer las Políticas de Proceso Explícitas, Las políticas definirán cuándo y por qué una tarjeta debe pasar de una columna a otra. Las reglas y directrices de su trabajo cambiarán conforme la realidad cambie.

Regla #5: Utilizar modelos para reconocer oportunidades de mejora: cuando los equipos tienen un entendimiento común de las teorías sobre el trabajo, el flujo de trabajo, el proceso y el riesgo, es más probable que sea capaz de construir una comprensión compartida de un problema y proponer acciones de mejora que puedan ser aprobadas por consenso. El método Kanban sugiere que un enfoque científico sea utilizado para implementar los cambios continuos, graduales y evolutivos.

Además de los 4 principios y las 5 reglas, posteriormente se añaden 2 reglas más:

Regla #6: Sistemas Adaptativos Complejos: Los principios del método Kanban están diseñados para sentar las bases de una organización que puede mejorar de forma incremental mediante el establecimiento de las condiciones que estimulen la mejora. Los Sistemas Adaptativos Complejos utilizan reglas simples y condiciones de semillas para estimular un cambio incremental, todo ello basándose en los principios de Kanban.

Regla #7: El papel del diseñador de sistemas: El sistema del apdo. anterior deberá ser controlado y adaptado. Deberá haber alguien que asuma el papel del diseñador del sistema, o facilitador del diseño del sistema, para controlar el proceso de cambio.

Herramientas para implementar Kanban: utilizando notas adhesivas, o tableros con ranuras. Otras veces, se puede usar software de seguimiento de trabajos, tales como: Kanban Tool, JIRA, Greenhopper, Trello, Cardmapping o Tablero Kanban online

3.3. XP (EXTERME PROGRAMMING)

La programación extrema (XP) fue creada por Kent Beck en la segunda mitad de los años 90 del siglo pasado, apareciendo oficialmente en su libro "Extreme Programming explained ", publicado en 1999, donde se define la Programación Extrema como:

"una metodología ligera para el desarrollo de software en un entorno donde los requerimientos del sistema son vagos o cambian rápidamente, adaptada a equipos de pequeño y mediano tamaño"

Lleva al extremo diversas prácticas consideradas buenas:

- Revisión constante del código (programación por parejas)
- Probar constantemente (tests unitarios y funcionales), incluso varias veces al día (integración continua)
- Si el diseño forma parte del trabajo diario (reorganización, del inglés refactoring)
- Deberá desarrollarse lo mínimo para que el sistema funcione (principio de simplicidad)
- Todo el equipo trabajará en la definición de la arquitectura.
- Si las iteraciones cortas son buenas, serán realmente cortas, segundos, minutos y horas, no semanas, meses y años (The Planning Game)

Se centra en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo.

XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios.

XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

El mayor objetivo de XP es la reducción de costes mediante una adecuada estimación de la velocidad del proyecto.

Los proyectos comienzan obteniendo User Stories y desarrollando soluciones Spike Solutions sobre una idea arquitectural general de la solución conocida como Architectural Spike. Posteriormente se mantiene una reunión con clientes/usuarios, desarrolladores y gestores para acordar una planificación entre todos de lo que hay que hacer.

3.4. FDD (Feature Driven Development)

El Desarrollo basado en funcionalidades se centra en iteraciones cortas que entregan funcionalidad tangible. Al contrario de otras metodologías, FDD afirma ser conveniente para el desarrollo de sistemas críticos.

FDD basa su proceso en funcionalidades o features, que son pequeñas funciones del producto.

El ciclo de FDD consta de los siguientes pasos:

- Análisis y Desarrollo de un modelo del producto
- Construir una lista de funcionalidades
- Planificación basada en las funcionalidades identificadas
- Diseño basado en las funcionalidades, realizado en iteraciones
- Construcción basada en las funcionalidades, realizado en iteraciones

3.5. TDD (Test Driven Development)

El Desarrollo guiado por pruebas es una metodología ágil. Promueve los test unitarios de cada una de las nuevas funcionalidades que se vayan a realizar.

Involucra otras dos prácticas de ingenieria de software (ciclo):

- **Escribir las pruebas primero (Test First Development)**
- **Refactorización (Refactoring).**

ATDD (Acceptance Test Driven Development) es una técnica de escritura de pruebas en la cual se espera que el usuario final defina y diseñe las pruebas que den el OK a su uso. También se le llama STDD - Story Test Driven Development.

3.6. BDD (Behavior Driven Development)

En la Ingeniería de Software, behavior-driven development o desarrollo guiado por el comportamiento (DGC), es un proceso de desarrollo de software que surgió a partir del desarrollo guiado por pruebas (DGP). Surge como respuesta a los problemas al enseñar el desarrollo guiado por pruebas: qué probar y qué no probar, qué tanto abarca una prueba, …

El corazón del BDD es la reconsideración de la aproximación a la prueba unitaria y a la prueba de validación.

3.7. DDD (Domain Driven Design)

El diseño guiado por el dominio es un enfoque para el desarrollo de software con necesidades complejas mediante una profunda conexión entre la implementación y los conceptos del modelo y núcleo del negocio. Las premisas del DDD son las siguientes:

- Poner el foco primario del proyecto en el núcleo y la lógica del dominio.
- Basar los diseños complejos en un modelo.
- Iniciar una creativa colaboración entre técnicos y expertos del dominio.

3.8. LEAN

La metodología ágil LEAN se centra en la entrega rápida y continua de valor al cliente, eliminando el desperdicio y mejorando continuamente el proceso de desarrollo. El enfoque está en la eficiencia, reducción de costos y mejora de la calidad, con énfasis en el aprendizaje continuo y la colaboración abierta entre el equipo de desarrollo y el cliente.

En el desarrollo de software, los siete principios de la metodología Lean son:

1. Eliminar el desperdicio: cualquier actividad que no agregue valor al cliente debe ser eliminada.
2. Aumentar el aprendizaje: la mejora continua se logra al aumentar el conocimiento y habilidades del equipo de desarrollo.
3. Decidir lo más tarde posible: las decisiones importantes deben ser tomadas solo cuando se tiene la información necesaria y suficiente.
4. Entregar lo más rápido posible: el software debe entregarse al cliente lo más pronto posible para obtener retroalimentación.
5. Empoderar al equipo: los miembros del equipo de desarrollo deben tener el poder de tomar decisiones y resolver problemas.
6. Crear integridad en el proceso: los procesos deben ser diseñados para evitar errores y asegurar la calidad.
7. Ver el todo: el equipo debe ver el panorama completo del proyecto y no solo centrarse en una parte del mismo para obtener una perspectiva clara y global.