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

Análisis: actividad que incluye **aprender** sobre el problema, **entender** las necesidades, **identificar** al usuario y **entender** todas las restricciones a la solución. Según Pressman, debe cumplir tres objetivos:

- Describir lo que el cliente quiere.
- Establecer una base para la creación de un diseño.
- Definir un conjunto de requisitos.

Las **tareas** que comprende el **Análisis del Sistema** son:

- Identificar las necesidades y restricciones de coste y tiempo.
- Evaluar la viabilidad del sistema desde el punto de vista técnico, económico y legal.
- Asignar las funciones y compromisos al elemento hardware, software y el elemento humano.
- La especificación del sistema.

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**.
- **Análisis de requisitos**.
- **Representación de requisitos**.
- **Validación de Requisitos**.

Para reducir los problemas y la complejidad se pueden emplear tres técnicas principales:

- **Abstracción**: controlar la complejidad, define la relación “general/específico” o “ejemplo de”.
- **Partición**: técnica de descomposición que divide el problema, estructural o funcionalmente,.
- **Proyección**: su objetivo es “ver” un problema desde distintas perspectivas o puntos de vista.

Fase de Requisitos del Software se puede descomponer en dos actividades:

- **Análisis de requisitos del software → Documento de Requisitos del Sistema (DRS).**
- **Especificación funcional o Descripción del Producto**, **Documento de Diseño Funcional (DDF**).

Ambos documentos conforman el **Documento de Especificación de Requisitos Software (ERS)**.

Según Sommerville, los requisitos pueden clasificarse en:

- **Requisitos de sistema.**
  - Requerimientos Funcionales Abstractos.
  - Propiedades del Sistema.
  - Características que no debe mostrar el sistema.
- **Requisitos software.**
  - **Requerimientos Funcionales:** especifican el funcionamiento del software.
  - **Requerimientos No Funcionales:** especifican aspectos adicionales que no corresponden con el funcionamiento del sistema.

Las estrategias y técnicas de educción se clasifican en:

- **Técnicas de alto nivel: relativas a entornos de trabajo.** JAD (Joint Application Design), JRP (Joint Requirements Planning), Entorno de Bucles Adaptativo, Prototipos, Factores Críticos de Éxito, Historias de usuario.
- **Técnicas de bajo nivel:** Entrevistas, Brainstorming, PIECES. Casos de Uso, Análisis de Mercado.

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. Pasos que se llevan a cabo:

1. Determinar el sujeto o **límite del sistema**, decidir qué es parte del sistema y qué es externo.
2. Encontrar **actores**: representan roles de usuario o roles desempeñados por otros sistemas.
3. Encontrar **casos de uso**: secuencias de acciones que un sistema puede realizar al interactuar con actores externos.
4. Refinar hasta obtener un límite claro para el sistema.

Permite capturar los requisitos funcionales y expresarlos desde el **punto de vista del 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:** Cuadro que contiene las diferentes partes del sistema y los casos de uso.

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.

ALGUNOS CONCEPTOS IMPORTANTES EN EL MODELADO DE CASOS DE USO

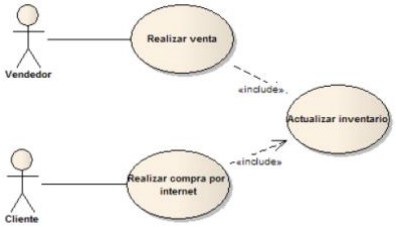
Generalización

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

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.



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.

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.

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

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.

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

- Descripción escrita, recordatorio de un funcionalidad pendiente de implementar (**CARD**).
- Conversaciones sobre la historia que permiten refinar sus detalles (**CONVERSATION**).
- Criterios de aceptación, determinan cuando la historia está completa (**CONFIRMATION**).

Las historias de usuario tienen las siguientes características **(INVEST)**: Independent, Negociable, Valuable, Estimable, Small y Testable.

Ventajas:

- 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.

Criterios de aceptación

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: DADO <condiciones> CUANDO <evento> ENTONCES <resultado>.
- Técnica de escenarios: Suele definir “el trayecto feliz” y un trayecto alternativo.

Historias de usuario	Casos de uso
Más cortas, estimables (entre 10h y 2 semanas)	**Más largos**. En ocasiones difícil dar estimación.
Se **pueden ir refinando y completando**..	Debería **estar completa desde el principio**.
Se suelen desechar una vez finaliza la iteración en la que fueron implementadas.	Perviven en el tiempo (desarrollo y mantenimiento).
Están **incompletas sin las pruebas de validación.**	
Los escenarios alternativos se expresan como diferentes condiciones en las pruebas	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

Metodología: Conjunto de **procedimientos** (pasos que hay que dar para construir el SW), **métodos o técnicas** (cómo se debe construir el SW) y **herramientas** (permiten automatización de métodos/técnicas) y un **soporte documental** que ayuda a los desarrolladores a realizar nuevo software.

Requisitos: Documentada, Repetible, Enseñable, Basada en Técnicas Probadas, Validada y Apropiada

Metodología vs Ciclo de Vida → una metodología puede seguir uno o más ciclos de vida. Una metodología define los pasos, productos, técnicas y métodos para seguir. 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. En realidad son ampliaciones y modificaciones de otras metodologías.
- Estructuradas. Orientadas a datos (o función) o procesos. SSADM, Merise, Metrica 3, etc.
- Metodologías OO.. OOAD – BOOCH, OMT-Rumbaugh, OOSE- Jacobson. Estas 3 forman **RUP** (Rational Unified Process) o PUDS, 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

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

- *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.*

Partiendo de los 4 postulados del **Manifiesto Ágil**, se desarrollan los 12 Principios del Manifiesto Ágil

1. Principal prioridad **satisfacer al cliente** → entrega temprana y continua de software de valor.
2. Son **bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo**.
3. **Entrega frecuente software que funcione**, en periodos de 2 de semanas a 2 meses,.
4. **Las personas del negocio y los desarrolladores deben trabajar juntos** de forma cotidiana.
5. Construcción de proyectos en torno **a individuos motivados**.
6. La forma más eficiente y efectiva de comunicar información → **conversación cara a cara**.

7. El **software que funciona** es la principal medida del progreso.
8. Se debe 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 cómo ser más efectivo y ajusta su conducta.

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

ROLES

El conjunto de roles participantes en el proceso puede dividirse en dos grandes grupos: cerdos (los comprometidos) y gallinas (implicados).

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.

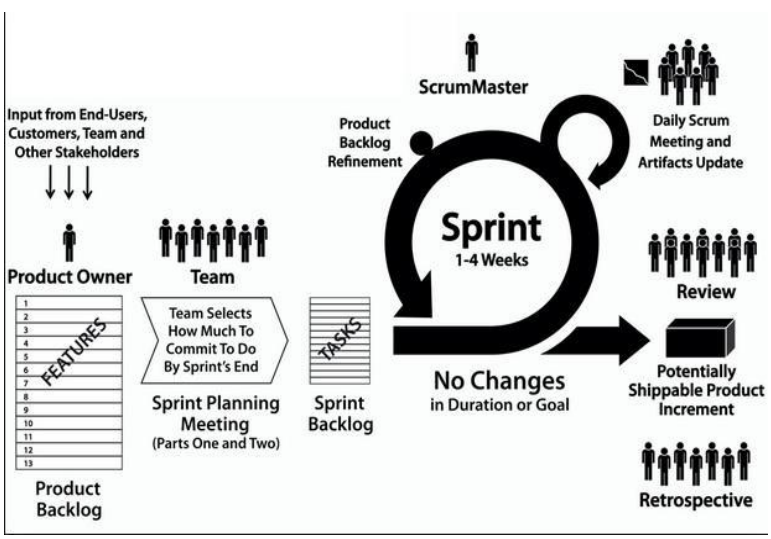
Roles "Gallina". Roles implicados en el proceso Scrum, que no son parte del proceso, pero deben tenerse en cuenta, ya que su participación es muy valiosa para el correcto feedback del producto. Son los expertos del negocio y otros interesados (stakeholders) para quienes el proyecto producirá el beneficio 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. 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, identificar los roles y compartir visión y objetivos. Las tareas de esta fase son:
   - Realizar preparativos necesarios para comenzar el proyecto, organizativos y tecnológicos.
   - Elaborar y priorizar lista de requerimientos del producto, en forma de épicas e historias de usuarios, creando el product backlog (requerimientos clasificados según su importancia).

- Establecer el definition of ready (cuándo Historia de Usuario está lista y suficientemente definida para entrar en el Sprint Backlog) y definition of done (cuándo 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 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 y 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. Objetivo: consensuar el Sprint Backlog. Los participantes: Product Owner, Equipo de desarrollo y Scrum Master. 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 y Qué impedimentos tiene o prevé encontrar. Los participantes (todos pueden asistir, pero solo los roles "cerdo" pueden hablar): Scrum Master, Equipo de desarrollo y Product Owner.

- **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 y 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, Product Owner y Equipo de desarrollo. Duración máxima 1 hora por cada semana del sprint.

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.

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

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 se denomina pull (tirar) en contrapartida al mecanismo push (empujar), donde el trabajo se introduce en función de la demanda. Podemos destacar 4 principios del método:

1. Comience con lo que hace ahora: estimula cambios continuos, incrementales y evolutivos.
2. Persigue el cambio incremental y evolutivo, mediante pequeños y continuos cambios.
3. Respetar el proceso actual, los roles, las responsabilidades y los cargos.
4. Liderazgo en todos los niveles..

En su aplicación al mundo del SW, según Anderson se basa en 5+2 Reglas:

- Regla #1: Mostrar el proceso (o flujo de trabajo).
- Regla #2: Limitar el trabajo en curso (WIP≡Work In Progress),.
- Regla #3: Optimizar el flujo de trabajo.
- Regla #4: Hacer las Políticas de Proceso Explícitas.
- Regla #5: Utilizar modelos para reconocer oportunidades de mejora.
- Regla #6: Sistemas Adaptativos Complejos.
- Regla #7: El papel del diseñador de sistemas:.

Herramientas para implementar Kanban: notas adhesivas, o tableros con ranuras. Kanban Tool, JIRA, Greenhopper, Trello, Cardmapping o Tablero Kanban online.

3.3. XP (EXTERME PROGRAMMING)

Creada por Kent Beck, apareciendo oficialmente en su libro "Extreme Programming explained ", publicado en 1999, donde se define 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), 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.
- Iteraciones 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.

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

Especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, con alto riesgo técnico.

El mayor objetivo es la reducción de costes mediante 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)

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)

Promueve los test unitarios de cada una de las nuevas funcionalidades que se vayan a realizar. Involucra otras dos prácticas de ingeniería de software (ciclo): **Escribir las pruebas primero (Test First Development)** y **Refactorización (Refactoring).**

ATDD (Acceptance Test Driven Development) técnica 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)

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 enfoca en la entrega continua de valor al cliente y la mejora constante del proceso de desarrollo, eliminando el desperdicio y aumentando la eficiencia y calidad. Los siete principios de la metodología LEAN en el desarrollo de software son:

- eliminar el desperdicio,
- aumentar el aprendizaje,
- decidir lo más tarde posible,
- entregar lo más rápido posible,
- empoderar al equipo,
- crear integridad en el proceso
- y ver el todo.

Estos principios buscan aumentar la eficiencia, reducir costos y mejorar la calidad, fomentando la colaboración entre el equipo de desarrollo y el cliente y fomentando la toma de decisiones informada y la resolución de problemas.