ELEMENTOS DE INGENIERÍA DE SOFTWARE

UNIDADES TEÓRICAS

UNIDAD 1: INGENIERÍA DE SOFTWARE

1. Introducción a la Ingeniería de software
2. Conceptos elementales de la ingeniería de software
3. Introducción a los métodos de gestión: tradicionales y ágiles

UNIDAD 2: CICLO DE VIDA DEL SOFTWARE

1. Etapas, Roles y Entregables

UNIDAD 3: MÉTODOS DE GESTIÓN

1. Tradicionales vs ágiles.
2. Manifiesto ágil: principios y valores
3. Ley de Paretto
4. Método práctico: Cascada
5. Método práctico: Scrum

UNIDAD 4: DOCUMENTACIÓN

1. Cascada: Casos de usos y UML
2. Agile: Documentación inicial
    - Elevator's pitch
    - User Story Mapping (USM)
    - Producto mínimo viable (MVP) (Inception)
3. Agile: Documentación de iteraciones
    - Tipo de incidencias
    - Slicing de US
    - Patrones de slicing de US
4. UX y Mockups gráficos (ambas)

UNIDAD 5: CALIDAD

1. Introducción a la calidad
2. Tipos de pruebas
3. Caja negra
    - Diseño de casos de prueba
    - Técnicas de diseño de pruebas
4. Reporte de bugs
5. Caja blanca: TDD
6. Integración continua

UNIDAD 6: MÉTRICAS Y ESTIMACIONES

1. Casada: Diagrama de Gantt (Ms Project). Estimación por tiempo.
2. Agile: Burndown Chart. Estimación por puntos (Planning poker). Velocidad del equipo

UNIDAD 1: INGENIERÍA DE SOFTWARE

1. Intro a la Ingeniería de software

El arte de aplicar los **conocimientos científicos para** la **invención** de herramientas necesarias para la sociedad, mediante técnicas y procedimientos de la industria y otros campos de aplicación científicos.

¿Qué es la ingeniería de software?

Es una **rama** de la **ingeniería** que desarrolla y gestiona **sistemas informáticos** utilizando técnicas y experimentos de la informática, la gestión de proyectos y otras disciplinas.

2. Conceptos elementales de la ingeniería de software

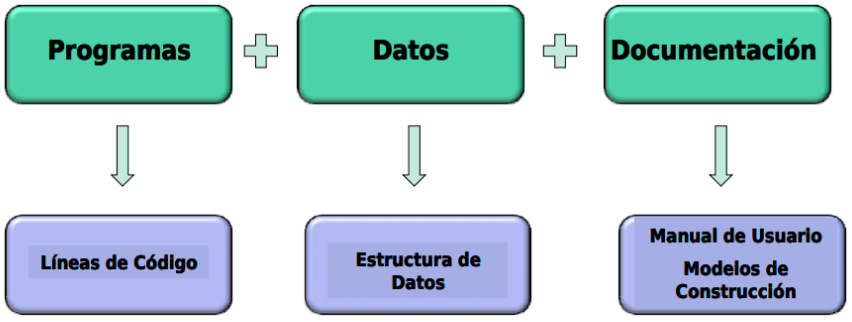
¿Qué es un sistema informático?

Es un sistema que nos permite almacenar y procesar información de manera automática, mediante una serie de partes interrelacionadas, formadas por: el hardware, el **software** y el personal.

¿Qué es un software?

Es un un conjunto que involucra los siguientes elementos:

1. Producto de software: programas
2. Datos: estructuras de datos
3. Documentación: manuales



Características

- Es un elemento lógico y no físico -> Usable mediante una interfaz
- Es desarrollado, no se "fabrica"
- No se estropea, pero se degrada -> Mantenible
- No hay piezas de repuesto
- Se construye a medida -> Reusabilidad

¿A qué se parece el software?

- A un casa (que se construye)
- A un libro (que se idea y se escribe)
- A una receta de cocina (que se inventa y se anota)
- A un servicio de un/a abogado/a en un juicio (que nos ayuda con su conocimiento especializado)

¿Producto o Servicio?

Problemáticas

Ya sabemos lo que es y sus características. Pero, ¿qué problemas conlleva su desarrollo según lo analizado hasta ahora?

- **Planificación imprecisa**: ¿cuándo se entrega?
- **Baja productividad:** ¿es rentable?
- **Calidad dudosa**: ¿cuán bueno es el software?
- **Insatisfacción del cliente**: ¿es lo que realmente quería?

Entonces, será necesario incorporar las siguientes tareas:

- Planificación
- Control y seguimiento: métricas
- Calidad
- Gestión

Por lo tanto, la gente que hace software, ¿qué tipo de habilidades y capacidades debe tener?

- Creatividad
- Lógica
- Estructura
- Capacidad de resolución
- Ley del mínimo esfuerzo
- Aprovechar los recursos ya existentes - no reinventar la rueda

¿Qué es un proyecto de software?

Es cumplir con el **ciclo de vida** del **software**, desde el relevamiento de los requisitos, hasta su puesta en producción (incluyendo las etapas de mantenimiento), llevado a cabo con ciertas **metodologías de gestión**, a entregar en el tiempo concreto para lograr el producto **software** deseado.

3. Metodologías de gestión: tradicional y ágil

¿Qué es una metodología para el desarrollo de software?

Es un conjunto de técnicas y métodos (frameworks) que ayudan a la organización de un equipo de trabajo para gestionar un proyecto de software. Cada metodología tiene un enfoque y cuenta con herramientas adecuadas para su aplicación.

¿En qué tipo de industrias se pueden implementar proyectos de desarrollo?

Básicamente en cualquiera que requiera un producto de software. La industria y el mercado están creciendo cada vez más, lo que conlleva a contar con varios tipos de empresas:

- De producto: aquellas que desarrollan específicamente productos de software. Ejemplos: Mercadolibre, Despegar, Facebook, etc
- Consultoras: aquellas que brindan servicios relacionados al desarrollo de software. Ejemplos: Accenture, globant, Practia, etc
- De industria: aquellas que construyen productos para otras industrias pero que tienen un área de sistemas destinada al desarrollo de software. Ejemplos: industria de medios, industria de bancaria, industria automotriz, industria de la salud, etc

¿Qué metodologías se pueden aplicar al desarrollo de software?

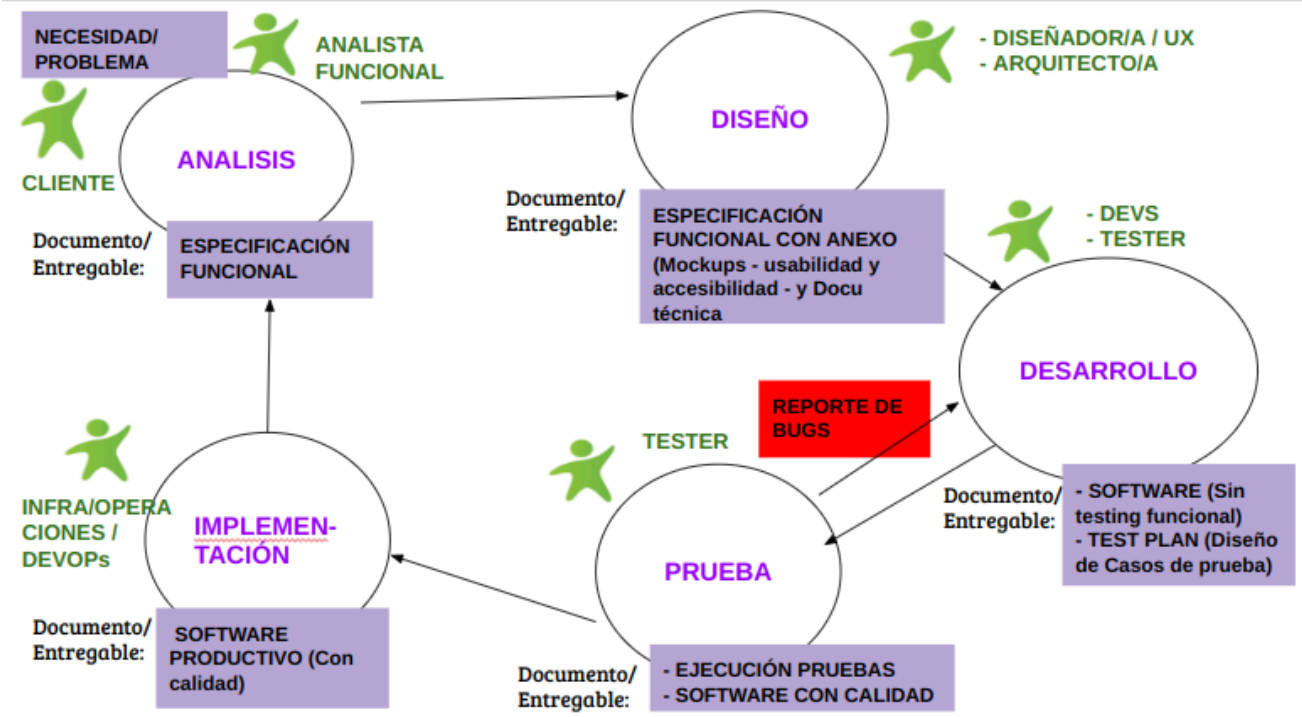
Como metodologías de desarrollo de software, filosóficamente hablando, se conocen 2 tipos:

- La tradicional
- La ágil

Cada una tiene un enfoque diferente, el cual se implementa en la industria del software, mediante frameworks o métodos, que han ido evolucionando a lo largo de las décadas. Muchos de ellos fueron combinando técnicas y prácticas tradicionales, con ideas nuevas e innovadoras que resolvían las falencias del método tradicional, hasta llegar a los frameworks más ágiles conocidos y utilizados en el mercado.
La **metodología tradicional** tiene un enfoque **secuencial** o también llamado **lineal**, mientras que la **metodología ágil** tiene un enfoque **iterativo e incremental**.

UNIDAD 2: CICLO DE VIDA DEL SOFTWARE

Antes de aprender las metodologías de gestión para el desarrollo del software, será necesario entender cómo es el ciclo de vida del software.
A continuación se mostrará un diagrama que contempla sus etapas, sus roles y la documentación / entregable que se genera en cada una de ellas.



Todo inicia con una idea, problema o necesidad, planteada por un cliente, quien solicita cubrir dicha necesidad o resolver dicho problema mediante un producto de software.

1. **Análisis**: para entender dicha necesidad (input) se procede con la primera etapa, la cual implica realizar un Análisis del problema, relevando cada requerimiento del cliente. Esta tarea la realiza el rol "Analista Funcional", el cual, como resultado de su relevamiento crea una Especificación funcional (output) con los requerimientos / requisitos que deberá contener el producto de software.

2. **Diseño**: como 2da etapa, se procede al Diseño de lo mencionado en la especificación funcional (input). Este diseño involucra 2 tipos de enfoques, y por lo tanto de roles: diseño técnico de la tecnología y arquitectura sobre la que se desarrollará el producto, y el diseño gráfico y visual (mockups) que contendrá el producto, el cual, además contempla las propiedades de usabilidad (UX) y accesibilidad. Estas tareas las desenvuelven los roles de "Arquitecto/a" y "Diseñador/a" respectivamente. El resultado de estas tareas se anexan a la especificación funcional, enriqueciendola con material técnico y visual (**output**).

3. **Desarrollo**: una vez completada la especificación (input), y en base a la misma se comienza, por un lado, con la etapa de Desarrollo, en la cual el Equipo de desarrollo comienza a programar lo requerido, y por otro (en paralelo), el Equipo de testing realiza el test plan con el diseño de los casos de prueba. El resultado de esta etapa resulta en el producto de software desarrollado pero sin testing funcional y un plan de pruebas: test plan (output).

4. **Prueba**: con el producto de software ya desarrollado (input), el Equipo de testing comienza a ejecutar las pruebas del test plan. Como resultado de dicha ejecución, se obtiene el test plan actualizado con el estado de las pruebas. Los estados posibles son:
    a. **Success**: la prueba fue exitosa
    b. **Fail**: la prueba falló. En este caso, se crea un reporte de bugs (errores) (**output**), en el cual se van agregando todos los defectos encontrados en las pruebas que fallaron. Esto implica crear la incidencia de tipo bug, y asignarla al equipo de desarrollo, el cual deberá resolver cada error en el código.

   En esta etapa se genera un breve ciclo de interacción entre el equipo de desarrollo y el equipo de testing. Una vez que ambos equipos llegaron a un acuerdo de calidad aceptable, se procede a pasar a la siguiente etapa, entregando así el producto de software desarrollado con calidad (output).

5. **Implementación**: contando ya con un producto de software con calidad (**input**), el mismo deberá ser implementado en el ambiente de producción. Para esto, se solicita al equipo, comúnmente denominado "Operaciones o Infraestructura", que realice el deploy del producto. Lo cual implica pasar la versión del último ambiente en el que se testeó, al ambiente productivo. Como resultado de esta última etapa, obtenemos un producto de software productivo con calidad (output).
   Como se mencionó en un principio, al tratarse de un ciclo, el mismo vuelve a comenzar ante una necesidad de cambios en las funcionalidades del producto, una evolución del mismo, o una nueva versión con corrección de errores (Bug fixing).

Una breve aclaración

El ciclo de vida del software inicia luego de concretarse un acuerdo entre el cliente y el proveedor, siendo que previamente se lleva a cabo una fase de negociación, la cual involucra un contrato legal y formal entre ambas partes; en éste se detallan los acuerdos mínimos, tanto de gestión como sobre las características del producto. Esta fase suele ser gestionada por el área comercial o de ventas de la empresa, en la cual no se involucra el equipo de desarrollo. Aunque es común que intervenga un líder técnico que analice la viabilidad del requerimiento.

Ambientes

Recordemos el concepto de ambiente:
Es un “espacio” donde se realiza una tarea determinada en función de la etapa en la que se encuentra el producto de software. Técnicamente, no es más ni menos, que una url que nos brindarán para poder operar en dicho ambiente.
Los ambientes mínimos con los que se deben contar son:

1. Ambiente de desarrollo
2. Ambiente de testing / calidad
3. Ambiente productivo (disponible al público)

Dependiendo del tamaño del producto / proyecto o de la empresa, para el equipo de testing se puede utilizar más de un ambiente de QA (Quality Assurance). Es muy común que coexistan el ambiente de testing (o QA), el de integración y uno de UAT (pre-productivo), además de los ya mencionados.

UNIDAD 3: MÉTODOS (FRAMEWORKS) DE GESTIÓN

Ahora que ya sabemos en qué consiste el ciclo de vida de un software, necesitaremos aprender cómo gestionarlo, llevando a la práctica cada etapa mediante la aplicación de algún método / framework.
En este espacio nos vamos a enfocar sólo en los 2 framework más conocidos de cada metodología:

- Metodología tradicional: cascada (waterfall)
- Metodología ágil: Scrum

Pero antes de profundizar en cada uno de ellos, realicemos un breve repaso histórico sobre los distintos métodos que han contribuido a lo largo del tiempo y evolucionaron hasta el día de hoy.

Historia de cada metodología

Métodos tradicionales

- Cascada (waterfall): creado por **Winston Royce en 1970**. Desarrollo del producto de forma secuencial que atraviesa cada una de las etapas del ciclo de vida del software.

- INCREMENTAL: creado por **Harlan Mills en 1980**. Es una extensión del método de cascada, dado que mantiene las 5 etapas, donde a partir de todos los requisitos funcionales ya definidos, el producto se va desarrollando de manera progresiva con nueva funcionalidad en cada incremento. Surgió como una manera de reducir la repetición del trabajo, y retrasar la toma de decisiones en los requisitos, hasta adquirir experiencia.

  PROS:
  - ➢ El cliente se involucra más
  - ➢ Se entrega software con más frecuencia
  - ➢ Se adapta mejor a proyectos chicos y a los cambios
  - ➢ Se agrega el concepto de "entrega de **algo** de valor" en el análisis

  CONTRAS:
  - ➢ Difícil de evaluar el costo total
  - ➢ Difícil de aplicar a los sistemas transaccionales que tienden a ser integrados y funcionar como un todo
  - ➢ Requiere de gestores experimentados
  - ➢ Los errores en los requisitos se siguen detectando tarde

- PROTOTIPADO O ITERATIVO: creado por **Gomaa en 1984**. Nace como una propuesta para identificar mejor los requisitos poco claros de cascada, a partir de los cuales, se construye un prototipo en poco tiempo y con poco dinero, para que el cliente pueda tener una vista preliminar del producto, probarlo y aportar feedback. Es un modelo ITERATIVO que se basa en el método de prueba y error, y con foco en la interfaz. Se entiende por error, cuando el cliente solicita una modificación. Se realizan todos los cambios necesarios hasta que el cliente quede satisfecho, en cuyo caso la prueba es exitosa. A partir de entonces, se puede comenzar con el desarrollo real del producto. Este modelo utiliza las etapas del ciclo de vida del software, pero extiende la etapa del diseño para el prototipo:
  - ➢ Análisis
  - ➢ Diseño
    - ○ Diseño rápido del prototipo
    - ○ Construcción del prototipo
    - ○ Evaluación del prototipo con el cliente
    - ○ Refinamiento del prototipo
  - ➢ Desarrollo (del producto)
  - ➢ Prueba
  - ➢ Implementación

  Este modelo es útil cuando el cliente conoce los objetivos generales, pero no identifica los requisitos detallados de entrada, procesamiento y/o salida.
  Hay dos tipos de prototipos:

- El desechable: sirve para eliminar dudas sobre lo que realmente quiere el cliente mediante el dibujo de una interfaz gráfica. Sólo se ejecuta la etapa de Diseño. ***Desventaja***: hay que bajar las expectativas del cliente para que entienda que se trata sólo de un boceto, y el producto aún no se desarrolló.
- El evolutivo: es un modelo parcialmente construido que puede pasar de ser prototipo a ser software, pero sin contar con buena documentación y calidad.
- Se ejecutan todas las etapas previamente mencionadas.

Desventaja: el equipo de desarrollo puede caer en la tentación de avanzar en el producto sin tener en cuenta los compromisos de calidad y prioridades del cliente.

- Espiral: creado por **Barry W. Boehm en 1986.** Este método se focaliza en proyectos grandes y complejos, para lo cual necesita una combinación de los métodos anteriores. Lo que diferencia en gran medida este modelo de los demás es que reconoce explícitamente los riesgos al construir software; esto genera una reducción considerable de las fallas en los proyectos grandes, ya que evalúa repetidamente los riesgos utilizando prototipos, y verifica cada vez el producto en desarrollo, mediante iteraciones.

El modelo se divide en **4 etapas** que se desarrollan de manera espiralada, desde el centro hacia afuera, tantas veces hasta finalizar el producto; donde en cada iteración se obtiene un producto mejorado del anterior, una versión.
Durante las primeras iteraciones, la versión incremental podría ser solamente un prototipo en papel, pero durante las últimas iteraciones, ya se producen versiones cada vez más completas del producto final.

Mapeo con el ciclo de vida del software:

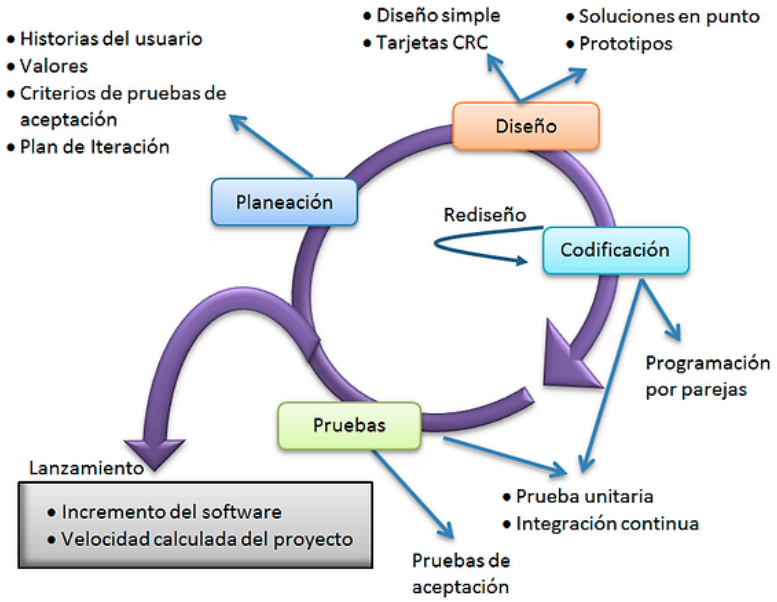
- **Análisis**: etapa de Análisis
- **Evaluación**: se **analizan los riesgos** y se procede a la etapa de Diseño mediante el/los prototipos
- **Desarrollo**: etapas de Desarrollo y Prueba del producto
- **Planeamiento**: planificación de la siguiente iteración

Métodos Ágiles

- RAD (Rapid Application Development): creado por **James Martin en 1991**. Presentado como el desarrollo rápido de aplicaciones que permite construir sistemas en poco tiempo. Se puede decir que es el primer acercamiento a lo que hoy conocemos como filosofía ágil, aunque conserva el modelo de Prototipado, al cual se le incorporaron las herramientas CASE, que lo convierten en un framework interactivo.

- RUP (Rational Unified Process / Proceso Unificado de Rational): es el framework formal y oficial del método "Proceso Unificado", publicado por **Ivar Jacobson, Grady Booch y James Rumbaugh en 1998**. Originalmente se diseñó un proceso genérico y de dominio público, el Proceso Unificado, y luego una especificación más detallada, RUP, de la empresa IBM que se vende como producto independiente.
  RUP no es un sistema con pasos firmemente establecidos, sino un conjunto de metodologías adaptables al contexto y necesidades de cada organización. Este framework comprende tres principios claves: fusión del concepto iterativo e incremental con el espiral, centrado en la arquitectura y dirigido por los casos de uso. Es el primero en incorporar el concepto de casos de usos mediante UML como comunicación gráfica, debido al auge de la programación orientada a objetos.

- XP (eXtreme Programming): creado por **Kent Beck en 1999**. Método iterativo e incremental focalizado principalmente en la etapa de desarrollo que se caracteriza por ser flexible y de apertura al cambio. Surgió como respuesta al auge de Internet y de las "punto.com" que enfatizaron la velocidad de comercialización y crecimiento de las empresa como factores comerciales competitivos; cuyos requisitos, rápidamente cambiantes, exigían ciclos de vida del producto cada vez más cortos. "eXtreme" proviene del concepto de que si algo funciona bien, por qué no llevarlo al límite y usarlo al máximo: como la programación de a dos (en pareja) funciona, hagámosla siempre; como las pruebas tempranas son buenas, probemos siempre antes de escribir el código (TDD), etc. Este método, propone maximizar las acciones y tareas que son indispensables y funcionan bien, dejando de lado aquellas que no aportan valor. Así, el método se basa en valores, principios y prácticas. Los valores y principios representan un propósito, pero son abstractos, mientras que las prácticas son concretas y ayudan al equipo a responsabilizarse de ellos. Para poder llevar a cabo estas prácticas, fue necesario aplicar un cambio que derivó en un punto de inflexión con respecto a la metodología tradicional: dividir los requerimientos en funcionalidades pequeñas, creando así Historias de Usuarios (User Stories).
  A continuación veamos de manera gráfica el flujo de trabajo propuesto por este método:



Bonus: https://www.digite.com/es/agile/programacion-extrema-xp/

KANBAN: creado en **Toyota (Japón) a inicios de los 50'** y **adaptado al software a principios del 2000**. Es un sistema iterativo e incremental para el desarrollo de proyectos basado en objetivos bien definidos. Este método incorpora la división de tareas, de aquellas que sólo aportan valor, y que están organizadas de manera visual en un tablero disponible para todo el equipo; proporcionando así un ritmo de trabajo continuo. Su lema es "Stop starting, start finishing" que significa "Deja de comenzar, comienza a terminar". Una particularidad es que no se aplica sólo a un proyecto, sino que el tablero puede combinar diferentes proyectos y tareas de manera independiente pero manteniendo siempre un flujo constante de trabajo. Con este método se hace lo justo y necesario pero bien hecho, es decir, **no se premia la rapidez, sino la calidad**. Para que esto sea posible es necesario eliminar o reducir lo que es secundario en el devenir del proyecto.

LEAN: creado por **Sakichi Toyoda y Taiichi Ohno en la década del 50' y 60' en Japón**, pero el **término oficial** se presenta en los **80'.** Su filosofía se enfoca en minimizar las pérdidas de los sistemas, al mismo tiempo que maximiza la creación de valor para el cliente. "Lean" significa magro. Para ello utiliza la mínima cantidad de recursos, es decir, los estrictamente necesarios para el crecimiento, eliminando así desperdicios que reducen el tiempo de producción y costo. Esta filosofía, más tarde aplicada a la ingeniería del software, con el libro **"Lean Software Development"** de **Mary y Tom Poppendieck en 2003**, se unifica con otras prácticas ágiles, dando como resultado, un desarrollo iterativo e incremental pero con procesos constantes de análisis y mejora continua (Kaizen) que maximizan el aporte de valor.
Bonus:https://www.progressalean.com/origen-y-evolucion-del-lean-manufacturing/#:~:text=A%20finales%20del%20siglo%20XIX,cuando%20se%20romp%C3%ADa%20un%20hilo.

Scrum: originalmente creado por **Ken Schwaber y Jeff Sutherland en 1995**, quienes se basaron en el libro "Wicked problems, righteous solutions" de Peter DeGrace and Leslie Hulet Stahl, quienes a su vez, tomaron el concepto de un artículo de Hirotaka Takeuchi e Ikujiro Nonaka en 1986 en Japón. Si bien Scrum, como framework, sólo se enfoca en la gestión de proyectos con filosofía ágil; su práctica ha evolucionado a través de los años derivando en una combinación de métodos ágiles ya existentes y formalizados con el surgimiento del Manifiesto ágil en 2001. Así es que aplica el método iterativo e incremental para la organización de las funcionalidades del producto, aprovechando las prácticas de XP para su desarrollo, gestionando las tareas mediante un tablero Kanban y maximizando el aporte de valor propuesto por Lean.

Frases populares y mitos:

- XP: es trabajar en parejas
- Kanban: es un tablero con tareas
- Scrum: es desarrollo rápido

Como hemos visto, cada uno de estos métodos tiene bastante más para ofrecer y es más complejo que lo que se conoce popularmente, por lo cual me veo en la obligación de desmentir dichos mitos y frases tan poco profundas.

Ahora sí vamos a profundizar en los 2 métodos que trabajaremos en la materia:

Método cascada (waterfall)

Un poco de historia y sus inicios

El desarrollo de los sistemas mediante un método de construcción por etapas se originó en la década del 70', para producir los sistemas de negocio en una época de grandes conglomerados empresariales. La idea principal era utilizar un proceso estructurado y metódico, que respete cada una de las etapas del ciclo de vida del software, tal y como se produce en las industrias automotrices. De ahí que este método / framework es el que más representa la metodología tradicional con su enfoque secuencial.
En 1985, el Departamento de Defensa de los Estados Unidos publicó el Estándar 2167 (DoD-STS-2167) que establecía un proceso estandarizado para el desarrollo de software: **el modelo en cascada**. Este estándar se basó en un paper llamado *"Managing the Development of Large Software Systems"*, escrito en 1970 por Winston Royce.

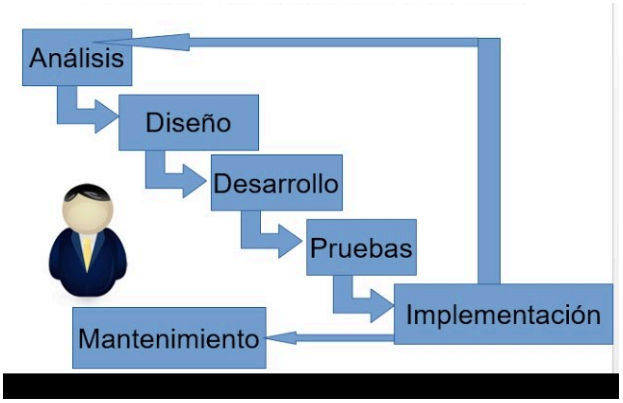
Lo llamativo del caso, es que este método se institucionalizó tras una mala interpretación del Departamento de Defensa sobre el paper de Royce; en el cual el autor explicita literalmente que dicho modelo es *"arriesgado y una invitación al fracaso"*. Como consecuencia de dicha conclusión, a continuación propone un modelo más propicio para la industria del software, basado en una construcción iterativa e incremental (*para más información leer el siguiente artículo* (1)). De esta manera, amén del error, el método de "Cascada" quedó en la historia como el primer modelo introducido en la ingeniería del software (de hecho Royce ni siquiera lo nombró de esa manera).

(1): La historia detrás del error

Características:

Tal y como hemos mencionado, este método se inspiró de la industria automotriz o las manufactureras, en las cuales, una vez creada una pieza del producto, no se puede volver atrás, necesariamente se debe pasar a la siguiente fase. A su vez, cada fase debe completarse en una secuencia y orden determinado, *"No se puede hacer el techo de una casa, si aún no tiene cimientos"*.
En la ingeniería de software este modelo también es conocido por su **enfoque secuencial o lineal**, esto se debe a que cada una de las etapas no puede iniciarse hasta que la etapa o fase anterior esté completa. Con cada etapa se genera un nuevo documento, que al finalizar la misma, se debe revisar, validar y aprobar; el cual sirve como punto de partida para la siguiente etapa. Como aprobación final se necesita la validación del cliente, y como puntapié inicial, se parte de la especificación funcional surgida de la primera etapa de análisis, a la cual se le van anexando el resto de los documentos.

Veamos un repaso gráfico de las etapas del ciclo de vida del software:



Problemáticas:

A continuación veremos las desventajas que posee este método y como consecuencia los posibles problemas que pueden derivar a raíz de su enfoque.

1. Dependencia: las etapas al ser secuenciales generan mucha dependencia entre sí. Lo que no permite paralelizar el trabajo y es propenso al arrastre de errores.

2. Documentación incompleta y errónea: cada etapa genera su propia documentación, la cual es anexada a la anterior; este exceso de documentación convierte al método en un proceso muy burocrático y propenso al arrastre de errores entre cada etapa. Asimismo, todos los requerimientos se deben determinar en la especificación funcional al inicio del proceso, por lo que una vez finalizada esta etapa no hay momento para su revisión y modificación, es decir que todo lo relevado en dicha etapa es fundamental para el éxito del proyecto. No hay espacio para el olvido o el malentendido de las funcionalidades. Lo cual termina generando documentación incompleta y/o errónea en la mayoría de los casos.
3. Rigidez: los desarrollos de este tipo pueden durar meses **sin que los clientes vean un producto final**, lo que provoca muchos problemas para actualizar los requisitos o recibir comentarios adicionales que afectan el proyecto. Puede resultar difícil cambiar las funcionalidades y características principales del software. Se caracteriza por definir **total** y **rígidamente** los requisitos al inicio de los proyectos. Los ciclos de desarrollo son poco flexibles y no permiten realizar cambios.

4. Costoso: ante los errores detectados, el costo en la corrección del producto puede ser altísimo debido al:
    a. Desconocimiento del origen del error: se desconoce dónde se originó el error, por lo cual hay que realizar ingeniería inversa de cada etapa hasta detectarlo.
    b. Corrección del error: si en el mejor de los casos se tratase de un error (bug) de programación, su resolución podría ser relativamente sencilla pero en cambio, si el error fuese producido en la etapa de diseño o peor aún, en la etapa de análisis, el costo de su corrección es realmente alto.

5. Expectativa vs realidad: dado los plazos tan largos, la poca participación del cliente durante todo el proceso, y la extensa documentación generada sólo al inicio del proyecto, es muy probable que al momento de la entrega del producto, el cliente se sienta insatisfecho por contar con un producto no deseado dado que no expresa lo que realmente tenía en mente.

6. ROI TARDÍO: el cliente desconoce la rentabilidad del producto hasta el momento en el cual es entregado, y dado los largos períodos que conlleva este método, es muy probable que, en el mejor de los casos, incluso ya sea demasiado tarde para que su producto sea competitivo en el mercado. Como contraposición, en caso de que el producto no haya cumplido con sus expectativas y deba ser modificado, es muy probable que el dinero aportado deje de ser una inversión para ser una pérdida; siendo que no obtuvo un retorno de inversión (Return on investment - ROI) a tiempo como para cubrir gastos que puedan retroalimentar los errores o incluso continuar con la evolución del producto.

Como conclusión, este método no funciona para proyectos comerciales cambiantes. Sólo puede aplicarse a proyectos, cuyo producto de software tenga una cantidad de funcionalidad estática, conocida y/o bien definida desde el inicio.
Pero como sabemos, la mayoría de los proyectos del mercado actual no tienen estas características y es por este motivo que el método en cascada está cada vez más en desuso, siendo reemplazado por los métodos ágiles.

El agilismo como solución

Como respuesta a estas problemáticas es que han surgido métodos (frameworks) más apropiados para el desarrollo de un producto de software y a las características de su industria, cuyo objetivo principal es agilizar el proceso de desarrollo para un mercado cada vez más dinámico, competitivo y cambiante. Con el fin de complir con dicho propósito es que los frameworks tradicionales fueron evolucionando con el tiempo, creando métodos con enfoques cada vez más ágiles, hasta prácticamente ser reemplazadas por el agilismo.

Proceso iterativo y el incremental

Como primera medida, se dejó de desarrollar de manera secuencial para pasar a un desarrollo por incrementos, lo cual generó la necesidad de obtener feedback del cliente en una etapa más temprana, incorporando así bocetos iniciales del producto mediante prototipos. Al contar con un producto mejor identificado desde el inicio y dividido en incrementos, surgió la necesidad de iterar todo el proceso.
De esta manera, si bien cada método representaba una evolución del anterior, sus desarrollos seguían dependiendo de una especificación funcional inicial; característica que sigue sosteniendo a este grupo de métodos dentro de la metodología Tradicional.

El punto de inflexión: dividir los requerimientos para el desarrollo
XP fue el primer método en unificar el proceso de desarrollo por incrementos pero manera iterativa, lo cual lo llevaría a romper con esta característica de la especificación funcional “completa” y al inicio del proyecto. De esta manera fue uno de los primeros que implementó el enfoque de la metodología ágil.

Motivo por el cual propone generar tantas funcionalidades chiquitas como requiera el producto. Pero el foco de XP no se basa precisamente en la gestión, sino que es un método con prácticas netamente técnicas que aportan calidad y agilidad a la etapa de desarrollo del ciclo de vida.

Gestión: organización de las tareas

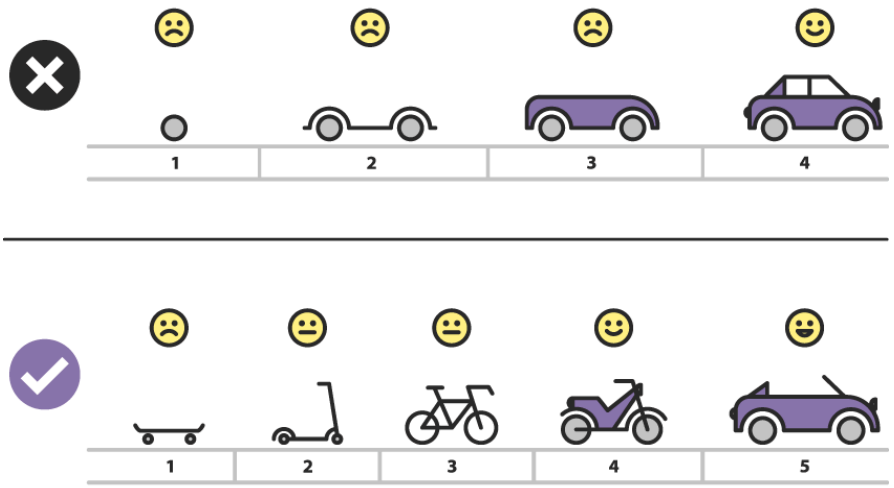
Casi en paralelo, aparece el método Kanban, el cual sí se focaliza en temas de gestión, para lo cual utiliza tableros que organizan las tareas (US y todas las necesarias) y visibilizan su estado, aportando calidad y valor también a la manera de organizar el trabajo.

Método Ágil: Scrum

El método aglomerante

Finalmente hemos llegado al método (framework) de trabajo ágil más utilizado en el mercado actual: **Scrum**. Éste aprovechó y tomó lo mejor de los métodos ágiles ya existentes, creando así un **marco de trabajo simple y liviano** que ayuda a las personas, equipos y organizaciones a generar valor e impacto mediante soluciones adaptativas para problemas de cierta complejidad. Así fue que utilizó el enfoque iterativo e incremental del agilismo, utilizado por XP, y sobre el mismo generó su propio esquema de trabajo.
Si bien los inicios de Scrum fueron contemporáneos al resto, su aplicación tal cual se la conoce en la actualidad deriva del surgimiento del Manifiesto ágil, dado que Scrum se basa en el empirismo y el pensamiento Lean. El empirismo afirma que el conocimiento proviene de la experiencia y la toma de decisiones basadas en lo que se observa. El pensamiento Lean reduce los desperdicios y se centra en lo esencial.

Resumen gráfico



A modo de conclusión veamos en un ejemplo gráfico, la diferencia entre el desarrollo de un producto, en este caso un autito, utilizando el método Cascada vs el método Scrum.
En la primera imagen se muestra un **enfoque secuencial**, desarrollado con metodología tradicional, donde el cliente recién conoce y obtiene el producto en la última etapa del ciclo de vida del software, sin tener una visión de su evolución, y a la espera de calcular su retorno de inversión. En la segunda imagen se observa un **enfoque iterativo e incremental**, desarrollado con **metodología ágil**, en el cual se obtiene un producto desde el inicio del proyecto. Si bien es una versión bastante minimizada del producto final, el cliente ya puede comercializarlo y así recuperar un poco de lo invertido, para volver a invertir en su respectiva evolución. De esta manera, incluso, de necesitar hasta podría decidir modificar su idea inicial de producto.
Básicamente la metodología ágil se diferencia de la metodología tradicional enfatizando sobre la **adaptabilidad**, buscando adaptarse fácilmente a los cambios, en lugar de la **previsibilidad**, que busca planificar y tener "todo" controlado desde el inicio.
Aquí se comparte una interesante comparación gráfica entre el desarrollo mediante el método incremental, el iterativo, y el iterativo e incremental.

El manifiesto ágil

Ya hemos pasado por el conjunto de métodos ágiles pero **¿cuándo se considera ágil a un método?**, **¿qué características debe cumplir?**

Pues bien, como una manera de estandarizar y normalizar los métodos ágiles es que se crea, de manera oficial, un **manifiesto** firmado por quienes ya venían aportando ideas a esta metodología. Por lo que el manifiesto no es más ni menos que un espacio unificado que conglomera varias ideas ágiles basadas en la filosofía Lean.

El manifiesto se compone de **4 valores y 12 principios**, los cuales proveen los pilares filosóficos y culturales del agilismo. El sitio oficial se encuentra en el siguiente link: https://agilemanifesto.org/iso/es/manifesto.html

Analicemos los valores planteados:

1. Individuos e interacciones **sobre** procesos y herramientas
2. Software funcionando **sobre** documentación extensiva
3. Colaboración con el cliente **sobre** negociación contractual
4. Respuesta ante el cambio **sobre** seguir un plan

Los valores mencionados sobre la izquierda tienen mayor preponderancia que los mencionados sobre la derecha. Esto no implica que se trate de un reemplazo de elementos, sino que los primeros se entienden como una mayor valoración sobre los segundos.

Aquí un breve análisis:

1. Los **procesos** y las **herramientas** deberán estar al servicio de las personas y sus objetivos, no al revés. Esto no quiere decir que deban ser descartados, todo lo contrario, se trata de un cambio de enfoque, dado que son elementos sumamente necesarios para llevar a cabo el método ágil en la práctica.
2. Se valora más contar con un **software funcionando** que "prometer" el software mediante una **documentación** súper extensa y "completa" del mismo.
3. No alcanza con el sólo hecho de firmar un **contrato**, la **colaboración** constante entre equipo y cliente será una pieza fundamental para el éxito del proyecto.
4. Se valora la adaptación a los **cambios**, tomando como certeza la evolución del producto mediante sus constantes cambios. Seguir un **plan**, si éste no responde a su evolución, no aporta valor.

Ley de Pareto

El agilismo como parte de su manifiesto focaliza constantemente en el concepto de **"valor"**, desde las funcionalidades que contendrá el producto hasta las acciones que toma el equipo. En cada decisión, con sus respectivos matices, está planteado e involucrado el aporte de valor que le será redituado al cliente en cada caso. Para llevar a cabo este concepto, se utiliza la **Ley de Pareto**, también conocida como la **ley del 80/20**.

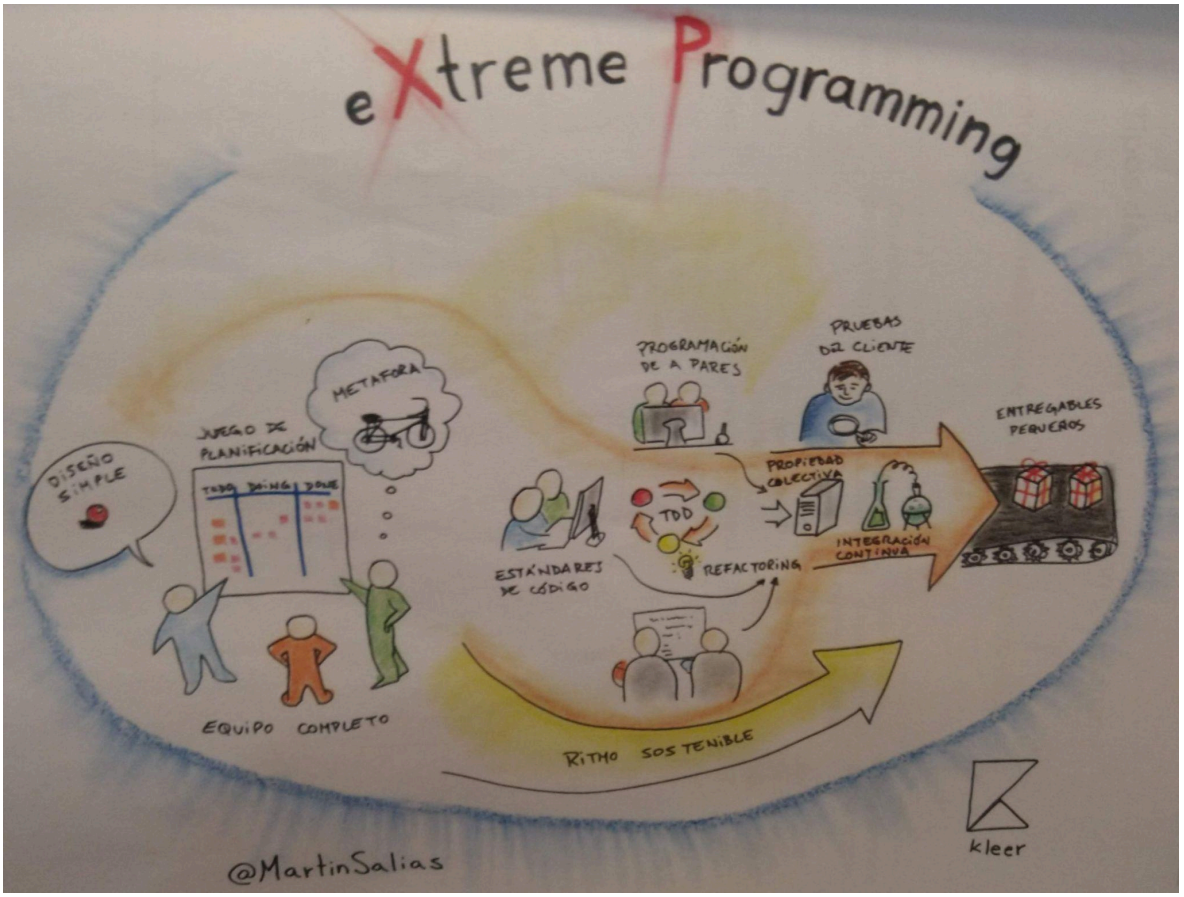
> **Ley de Pareto**: es una proporción entre 2 variables que nos permite medir el valor aportado / ganancia obtenida en función del esfuerzo / costo invertido.

Scrum: el framework

Retomemos nuevamente el framework **Scrum**, para así poder relacionar la gestión de proyectos ágiles, con este marco de trabajo, que permite generar valor e impacto en las organizaciones construyendo productos de calidad.

¿En qué consiste Scrum?

Scrum es **simple**, no es más que un **marco** (tal como el de una puerta), es decir que sólo brinda una **estructura general** pero sin indicar detalles técnicos ni de micro gestión. Scrum es un marco **incompleto** de manera intencional, sólo define las partes necesarias para implementar, a grandes rasgos, la teoría del manifiesto ágil. Scrum se basa en la inteligencia colectiva de las personas que lo utilizan. En lugar de proporcionar instrucciones detalladas, sus **reglas** son sólo una **guía** para la mejora continua y la interacción humana, lo cual trae como consecuencia la organización del trabajo. Es por este motivo, que será sumamente importante ser **conscientes** de las cualidades humanas que cada persona debe aportar para que el marco sea exitoso.
La oportunidad de mejora que propone Scrum y su simplicidad permite, **a medida que se vaya necesitando**, combinar procesos, técnicas y prácticas; ya sean existentes o definidas por el propio equipo.



Por ejemplo, una manera de aprovechar el resto de los métodos ágiles y no reinventar la rueda, sería **combinar las prácticas de gestión de Kanban** con las **técnicas de desarrollo de XP**, todo esto **dentro** de este marco. Esta combinación suele ser la más común. De ahí que se utiliza, para cada iteración, el **tablero Kanban** para gestionar las tareas, la técnica **pair-programming** para el desarrollo, y las prácticas **TDD, BDD e integración continua** para la calidad del producto.
De esta manera es que Scrum se trata de una propuesta experimental y empírica, basada en la prueba y error que genera la propia experiencia de aplicar el marco una y otra vez en cada iteración, produciendo así mejoras en el proceso y en el entorno de trabajo. El empirismo afirma que el conocimiento surge de la experiencia y que las decisiones se deben tomar sobre la base de los hechos conocidos.
Para esto Scrum cuenta con los siguientes **pilares empíricos**:

Pilares de Scrum

- Transparencia: los aspectos visibles y entendibles por todos los responsables del resultado. Por ejemplo: lograr un lenguaje común con respecto al proceso, estar de acuerdo en las expectativas entre quienes construyen y quienes aceptan el producto. La **transparencia** permite la **inspección**. La inspección sin transparencia genera engaños y desperdicios

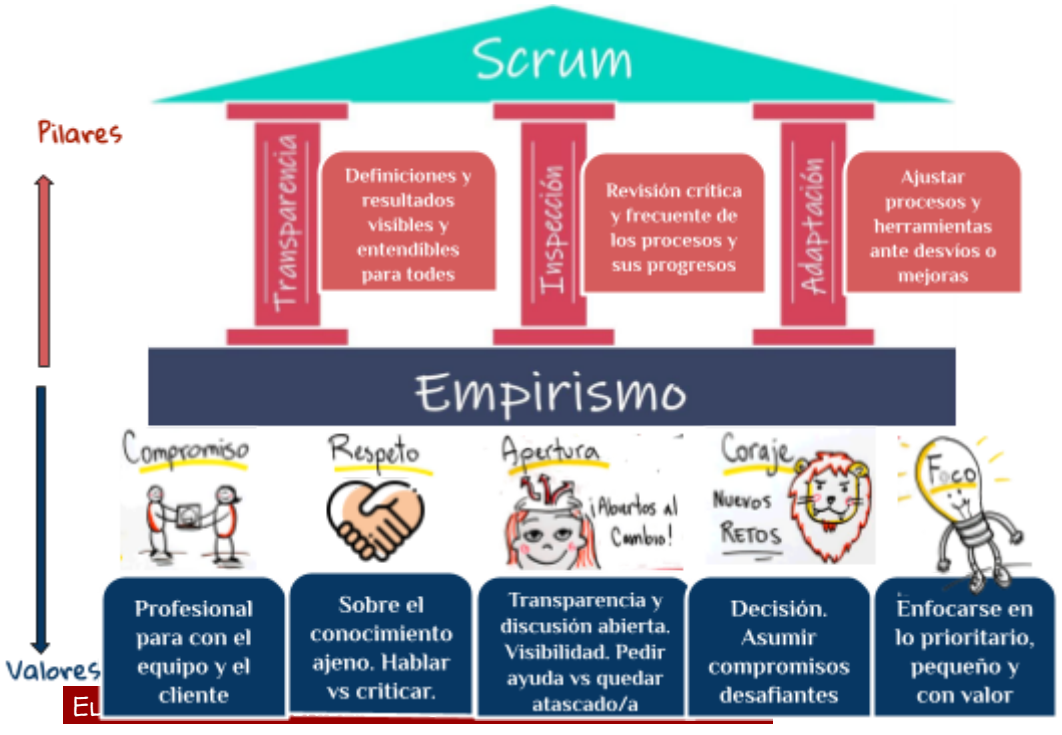
- Inspección: les responsables del resultado deben inspeccionar frecuentemente los artefactos y el progreso hacia el objetivo para detectar desvíos indeseados. La **inspección** permite la adaptación. La inspección sin **adaptación** se considera inútil. Scrum está diseñado para provocar cambios

- Adaptación: si en una inspección se detectan oportunidades o ideas que agregan mayor valor que aquello ya planificado, el proceso o el producto pueden ajustarse para maximizar el valor entregado. La adaptación se vuelve

más difícil cuando las personas involucradas no están empoderadas o no poseen capacidad para autogestionarse. Se espera que un equipo de Scrum se adapte en el momento en que aprenda algo nuevo por medio de la inspección. Para que los pilares surtan efecto, será necesario que las personas se comporten de manera consecuente con ciertos valores que le den sentido, y de esta manera aporten a la parte cultural que el método requiere. Básicamente el uso exitoso de Scrum depende de que las personas sean más competentes en vivenciar los siguientes 5 valores.

Valores de Scrum

- Compromiso: los equipos Scrum tienen mayor control sobre sus actividades, por eso se espera de su parte el compromiso profesional para el logro del éxito.
- Respeto: debido a que los miembros de un equipo Scrum trabajan de forma conjunta, compartiendo éxitos y fracasos, se fomenta el respeto mutuo, y la ayuda entre pares es una cuestión a respetar
- Apertura / Franqueza: los equipos Scrum privilegian la transparencia y la discusión abierta de los problemas. No hay agendas ocultas ni triangulación de conflictos. La sinceridad se agradece y la información está disponible para todes, todo el tiempo.
- Coraje: debido a que los equipos Scrum trabajan como verdaderos equipos, pueden apoyarse entre compañeres, y así tener el coraje de asumir compromisos desafiantes que les permitan crecer como profesionales y como equipo
- Foco: los equipos Scrum se enfocan en un conjunto acotado de características por vez. Esto permite que al final de cada iteración se entregue un producto de alta calidad y, adicionalmente, se reduce el tiempo de salida a producción.



Sprints: ¿cómo se gestionan las iteraciones en Scrum?

Para scrum **una iteración es un período de tiempo**, es decir que tiene una fecha de inicio y una de fin, ambas bien definidas, y donde cada iteración tiene la misma duración. Una iteración en scrum se denomina: Sprint. Un sprint puede durar entre 1 y 2 semanas. Pero tener en cuenta que no se puede ***"alargar o acortar un sprint"*****,** dado que es un período de tiempo.

¿Qué significa esto?

Es muy común que al no terminar las tareas planificadas, se necesite "alargar el sprint", pero esto no es posible si tenemos en cuenta la definición de iteración… no podríamos alterar el paso del tiempo. El sprint finalizó sólo con el transcurso de los días. En términos más subjetivos, a lo sumo, se podrá **"alterar la duración del sprint"**, pero **no es una buena práctica**, dado que sería muy difícil identificar, por un lado la velocidad del equipo, y por otro la real capacidad del trabajo comprometido. De esta manera se debe finalizar cuando se cumple el período correspondiente, pasando al Product Backlog las incidencias adeudadas para retomar en el próximo sprint (o en el que el/la cliente crea necesario).
En dicho caso se podría decir que se trató de un sprint complicado, del cual habrá que aprender y accionar para evitar que vuelva a suceder, basándonos en los pilares de inspección y adaptación.

Elementos de Scrum: ¿Qué se realiza dentro de los sprints?

Scrum cuenta con 3 elementos formales para sus pilares dentro de un **elemento contenedor: el Sprint**. Estos elementos funcionan porque implementan todos los pilares empíricos de Scrum basándose en los valores.

Scrum cuenta con 3 elementos:

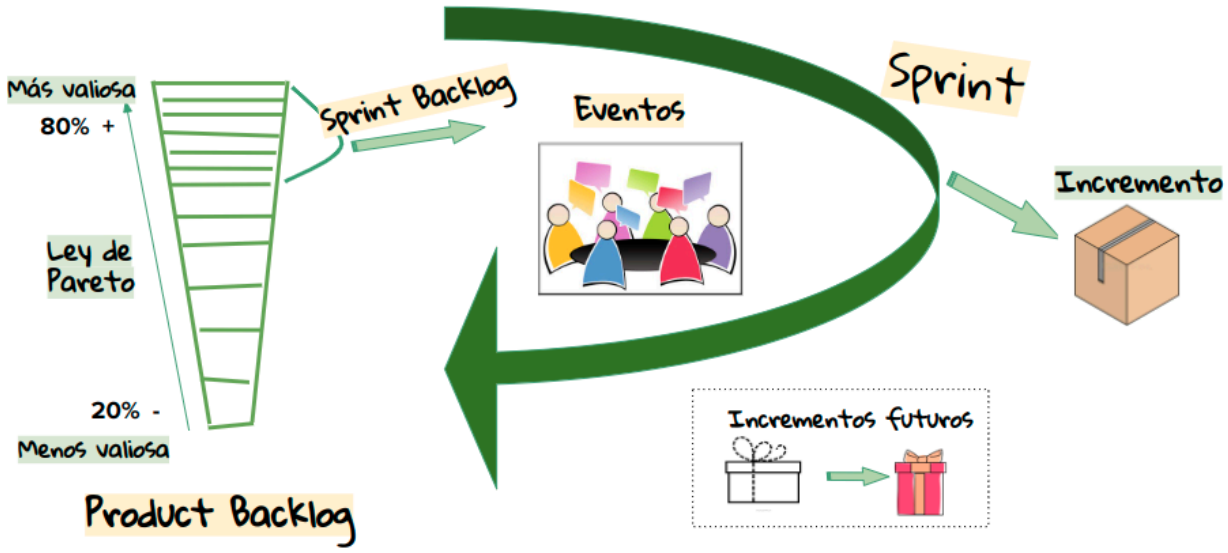
- Artefactos
  - Product Backlog
  - Sprint Backlog
  - Incremento
- Responsabilidades
  - Scrum Master
  - Product Owner
  - Developers (todas las personas que participan del desarrollo, es decir personas que programan, diseñan, testean, etc)
- Eventos
  - Sprint Planning (+ el refinamiento)
  - Daily Scrum
  - Sprint Review
  - Sprint Retrospective

Así entonces, el Sprint pasa a ser un **contenedor** para todos los eventos. Cada evento en Scrum es una oportunidad formal para inspeccionar y adaptar los artefactos de Scrum.

Artefactos: ¿Qué son los artefactos de Scrum?

Los artefactos son las herramientas que permiten al equipo organizar y visibilizar el trabajo **(transparencia)**. **Veamos cada uno de ellos:**
Recordemos que el objetivo es obtener una versión del **producto de software funcionando con calidad** a partir de un conjunto de requerimientos (funcionalidades) a desarrollar en dicha iteración, a esta versión se lo denomina Incremento y al conjunto de requerimientos a partir del cual se desarrolló, se lo llama Sprint Backlog**.** Por lo tanto el Sprint backlog contiene una parte del listado general de todas las funcionalidades del producto (incidencias). A este listado general se lo denomina Product Backlog**.** En conclusión, el Sprint Backlog es el subconjunto de incidencias del Product Backlog a desarrollar para obtener un incremento, el cual a su vez responde al **compromiso** definido en la definición de hecho.

A continuación veremos un gráfico muy simple que muestra la interacción de los artefactos:



Analicemos

Como **artefacto** inicial contamos con el Product Backlog. Como ya dijimos este debe contener **todas** las incidencias del producto hasta el momento, las cuales deben estar **ordenadas** por **prioridad** en base al **valor** que aportan, aplicando la **Ley de Pareto**. A partir de éste se desprende otro artefacto: el Sprint Backlog, el cual se crea al iniciar el sprint.

Luego, durante el sprint, el equipo cumple con todos los EVENTOS para gestionar las tareas que se necesitan en cada caso y desarrollar las incidencias. Al finalizar el sprint se entrega un nuevo INCREMENTO.

Una característica importante del agilismo, que proviene de Lean, implica "NO PRODUCIR CON DEMASIADA ANTELACIÓN". De esta manera, las incidencias en el Product Backlog no deberían estar definidas con mucho nivel de detalle hasta que no se acerque su desarrollo.

Es decir, se pueden tener incidencias con una escasa definición o muy ambiguas (Épicas), sabiendo que necesitarán un trabajo previo de refinamiento, pero, como asumimos que habrá cambios, no conviene anticipar su preparación hasta cuando sea inminente que se vayan a desarrollar, y por ende, entrar en un sprint. Éstas se ubican al final del Producto Backlog.

RESPONSABILIDADES: ¿QUÉ SON LAS "RESPONSABILIDADES" EN SCRUM?

Para que el trabajo realmente sea en equipo y de manera colaborativa, será necesario definir las RESPONSABILIDADES de cada integrante durante el Sprint.

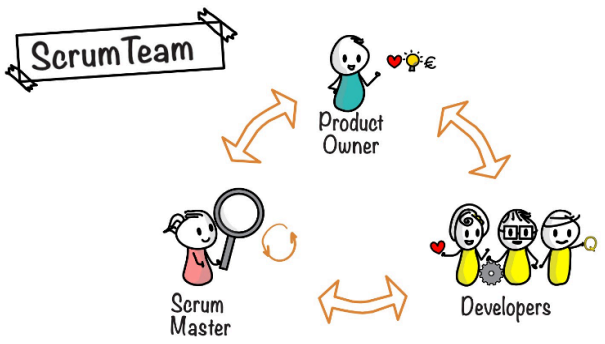
Scrum propone responsabilidades en lugar de roles para evitar acoplar ciertas tareas en una única persona, y evitar por un lado, generar un cuello de botella, y por otro caer en los roles tradicionales.

¿QUÉ RESPONSABILIDADES LE COMPETEN AL EQUIPO SCRUM?

Para llevar a cabo el proceso mencionado en el esquema anterior será necesario conocer qué responsabilidades propone Scrum. Siendo que se trata de un método ágil puntualmente para el desarrollo de software, necesitamos contar con personas que respondan a responsabilidades relacionadas al ciclo de vida del software, pero con un enfoque diferente.

Al equipo general se lo denomina EQUIPO SCRUM, el cual se compone de:

- PRODUCT OWNER (P.O)
- DESARROLLADORES/AS O EQUIPO DE DESARROLLO
  - TESTERS
  - DISEÑADORES/AS
  - DEVOPS
- SCRUM MASTER (S.M)
- STAKEHOLDERS



Product Owner

El/la Propietaria del Producto (Product Owner - P.O) es responsable de maximizar el valor del producto resultante del trabajo del equipo Scrum, definiendo las funcionalidades necesarias en base al objetivo del producto. Es la persona “dueña del producto”. Si bien, se entiende que el/la cliente es efectivamente la dueña del producto, en muchas ocasiones sucede que esta persona no tiene tiempo para realizar las tareas minuciosas de análisis y gestión del producto, por lo cual se apoya en alguien que oficie como tal.

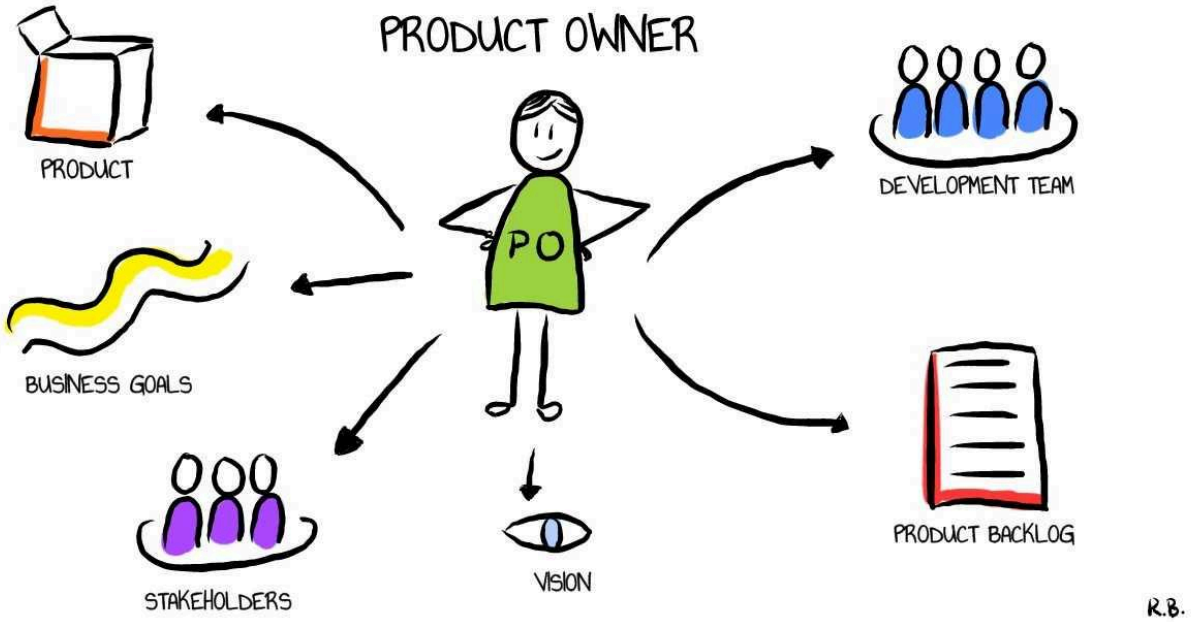
El trabajo de/la PO se refleja, principalmente, en la **gestión del Product Backlog**, lo cual incluye:

- Relevar, analizar y comunicar explícitamente el objetivo del producto
- Definir de manera clara en el Product Backlog, las incidencias que representan las funcionalidades
- Mantener actualizado el Product Backlog por prioridad según el valor de las funcionalidades de cara al objetivo del producto (para cada incremento)
- Asegurar la visibilidad, transparencia y comprensión del Product Backlog por parte de todas las personas involucradas
- Asegurar la viabilidad técnica de las funcionalidades del Product Backlog comunicándose con el equipo de desarrollo

El/la PO puede hacer el trabajo anterior o puede delegar la responsabilidad a otras personas. En cualquier caso, sigue siendo la persona responsable, es decir que **una persona, no un comité**, aunque en ocasiones puede representar las necesidades de muchas partes interesadas. Aquellas personas que deseen cambiar funcionalidades pueden hacerlo pero tratando de negociar con criterio con el/la P.O. Las **actividades** más habituales de un/a Product Owner además de gestionar el Product Backlog son:

- Gestionar las expectativas de los stakeholders
- Relevar y/o determinar las características funcionales de alto y de bajo nivel
- Generar y mantener el plan de entregas (release plan): fechas de entrega y contenidos de cada una
- Maximizar la rentabilidad del producto
- Redefinir las prioridades de las funcionalidades según avanza el proyecto, acompañando así los cambios en el negocio y la tecnología

Su homónimo en la metodología tradicional es el/la Analista Funcional.



Equipo de desarrollo

Se entiende como equipo de desarrollo a las personas cuyas responsabilidades llevan a cabo la construcción del producto durante el ciclo de vida del software. Por lo tanto, estas responsabilidades involucran tanto a desarrolladores/as, diseñadores/as, testers, devops y toda aquella persona involucrada en la construcción del producto. Por eso se dice que es un equipo multifuncional.
Son las personas del equipo Scrum que se comprometen a crear cualquier aspecto de un incremento útil (funcional) en cada Sprint. Tener en cuenta que, puntualmente, la etapa de análisis será llevada a cabo por el/la PO, aunque para completar el ciclo de vida en un desarrollo ágil, la interacción y comunicación del equipo de desarrollo con el/la PO es fundamental. Se busca que el equipo de desarrollo sea autogestionado en cuanto a la manera de organizar su trabajo. Nadie, ni siquiera el Scrum Master, tiene autoridad para decirle al Equipo de desarrollo la forma en la que debe hacer su trabajo.

Tener en cuenta que para el desarrollo de un proyecto ágil con Scrum , el equipo de desarrollo **siempre es responsables de:**

- Crear un plan para el Sprint: el Sprint Backlog
- Inculcar la calidad adhiriéndose a una definición de hecho
- Adaptar su plan cada día hacia el Objetivo del Sprint

- Entregar el incremento al finalizar cada sprint
- Responsabilizarse mutuamente como profesionales
- Proponer mejoras para evitar cometer los mismo errores en cada sprint

En el Equipo de Desarrollo no existen títulos ni rangos jerárquicos en cuanto a las tareas, es decir que todas las personas son de desarrollo sin importar el tipo de trabajo que realice. Tampoco hay sub-equipos, al margen de la cantidad de dominios de actividades que sea necesario realizar (ejemplo: prueba, análisis, arquitectura), como es el caso de cascada. Aunque los miembros del Equipo de Desarrollo tengan habilidades enfocadas en diferentes aspectos de la construcción del producto, la responsabilidad corresponde al equipo como un todo, cada uno con sus tareas puntuales.

Scrum Master

El/la Scrum Master es responsable de establecer Scrum tal como se define en la Guía de Scrum. Lo consigue facilitando (ayudando y acompañando) al equipo de trabajo en su día a día para que todes logren comprender la teoría y la práctica de Scrum, tanto dentro del Equipo como en toda la organización. Es la persona responsable de la efectividad del equipo Scrum.

El/la Scrum Master sirve al Equipo de desarrollo de varias maneras:

- Capacitar en autogestión y ayudar en la multifuncionalidad
- Ayudar a mantener el foco en la creación de incrementos de valor que cumplan con la definición de hecho
- Ayudar al entendimiento de las funcionalidades del Product Backlog
- Promover la eliminación de los impedimentos

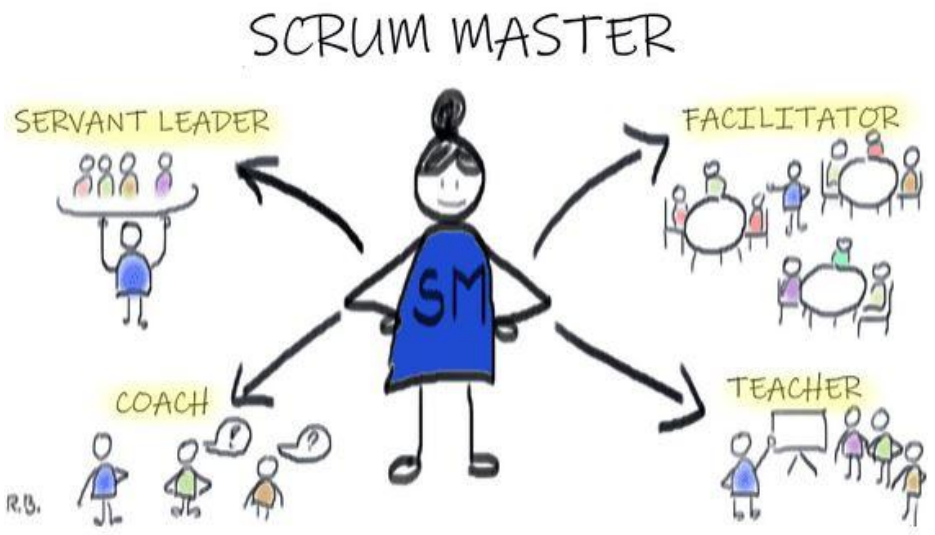
- Asegurar que todos los eventos de Scrum se lleven a cabo, que sean positivos, productivos y que se respete el tiempo establecido (time-box) para cada uno de ellos
- Asegurar la cooperación y comunicación dentro del equipo
- Detectar problemas y conflictos personales para ayudar a resolverlos
- Asegurarse que las acciones de mejora se lleven a cabo
- Ayudar al/la PO a cumplir con sus tareas

El/la Scrum Master sirve al/la Product Owner de varias maneras:

- Ayudar a encontrar técnicas para una definición eficaz del objetivo del producto y la gestión de los retrasos en el mismo
- Ayudar a definir de manera clara y concisa las incidencias del Product backlog según el objetivo del producto
- Facilitar la colaboración con las partes interesadas (stakeholders)

El/la Scrum Master sirve al/la organización de varias maneras:

- Liderar, capacitar y mentorear a la organización en su adopción de Scrum
- Ayudar a las partes interesadas a comprender y promulgar el enfoque empírico
- Eliminar las barreras entre las partes interesadas y el Equipo Scrum



El/la Scrum Master puede ser visto como una Facilitador o Coach, incluso muchas veces se lo referencia así en lugar de Scrum Master, dado que esta persona se encarga de facilitar “todo lo se necesite” para que la metodología ágil se cumpla mediante el método Scrum. **Pero ¿qué significa facilitar?**
Significa **hacer fácil o posible**. Por lo tanto, el/la SM deberá estar al servicio del equipo y liderar proporcionando espacios, herramientas, charlas, capacitaciones, mentorías, coaching, actividades, contactos, información, etc, para colaborar con el éxito del proyecto.

¡Una aclaración importante!

El/la SM **no** debe **ejecutar las acciones**, sino **facilitar** todo lo necesario para que **quien corresponda las ejecute**. Mucho menos involucrarse en las definiciones técnicas.

Stakeholders

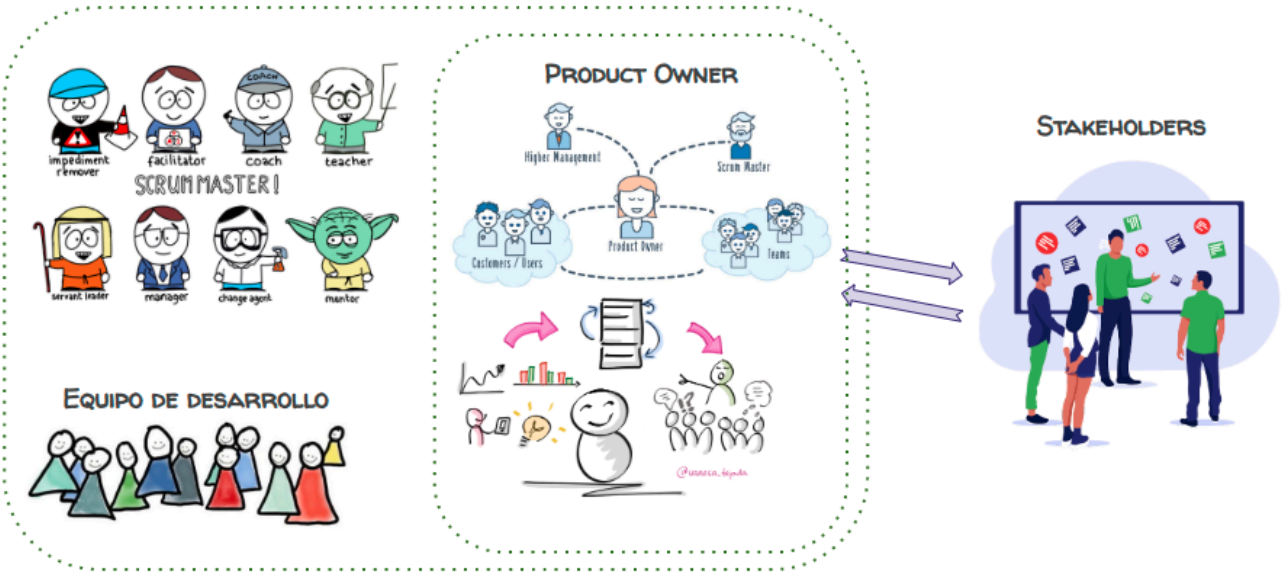
Son personas interesadas en el producto (partes interesadas), pero que no participan en el día a día de cada sprint, en general son personas de otras áreas de la empresa que tiene un interés particular en el producto. Por ejemplo, personas de marketing, de métricas, de finanzas, etc. A los/las stakeholders se les invita por ejemplo a una reunión inicial, de kick-off del proyecto (evento Inception) o en alguna review donde se hayan incluído las incidencias solicitadas por ellos/as.

En pocas palabras, y a modo de resumen de las responsabilidades, Scrum requiere un/a Scrum Master para fomentar un entorno donde:

1. El/la Product Owner ordene el trabajo de un problema medianamente complejo en un Product Backlog
2. El equipo Scrum convierta una selección del trabajo en un incremento de valor durante un Sprint

3. El equipo de Scrum y sus partes interesadas (stakeholders) inspeccionen los resultados y realicen los ajustes necesarios para el próximo sprint
4. Repetir

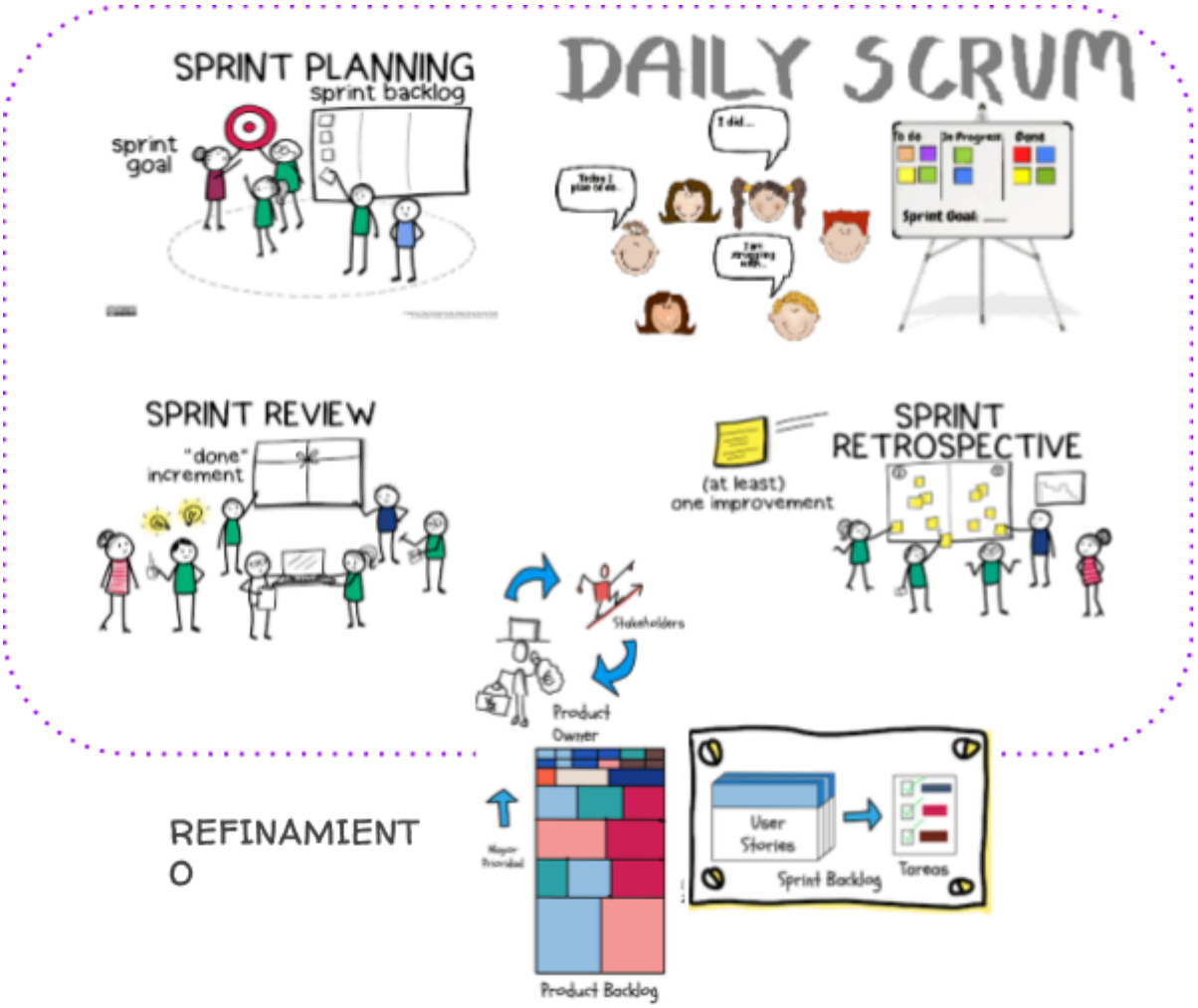
Gráfico resumen de todas las responsabilidades de Scrum:



EVENTOS ¿QUÉ EVENTOS SE REALIZAN EN UN SPRINT?

Durante el sprint se realizan reuniones específicas que Scrum las identifica como EVENTOS**.** Lo que caracteriza a un evento, por sobre una reunión tradicional, es que tiene un objetivo claro y acciones bien definidas que le permiten al equipo aprovechar bien el tiempo invertido **aportando valor**. En cada sprint se realizan todos los eventos en un orden específico.

Veamos un gráfico con todos los eventos que serán explicados a continuación en más detalle:



Veamos qué eventos propone Scrum y cuál es el objetivo de cada uno:

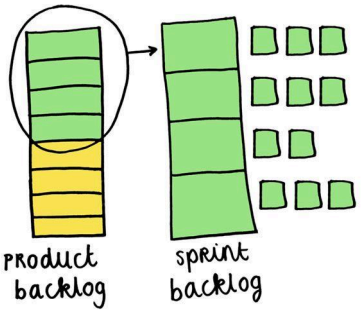
Sprint Planning

Es el primer evento que da inicio al sprint, cuyo **objetivo es planificar el trabajo** a realizar durante dicho sprint para lograr, de a poco, el objetivo del producto. Cada Sprint debe acercar cada incremento al objetivo del producto.
La planificación del sprint aborda los siguientes pasos:

1. **Definir el objetivo del sprint (sprint goal)**, respondiendo a **¿Qué valor se va a aportar en este sprint?**
   (Compromiso)
   El/la PO cuenta cómo el producto podría aumentar su valor y utilidad en el Sprint actual, a partir del cual el equipo colabora para definir un objetivo de Sprint.

2. **Seleccionar las incidencias del sprint**, respondiendo a **¿Qué vamos a realizar para cumplir con el objetivo?**

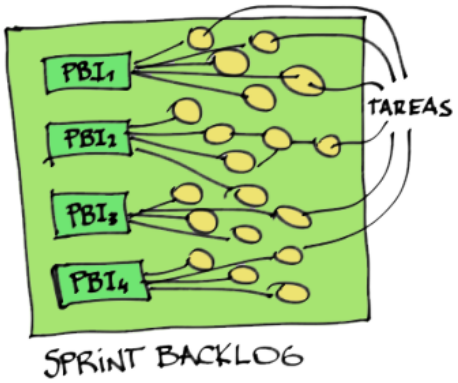
En conjunto con el/la PO se acuerdan las incidencias del product backlog más prioritarias (tope del mismo) que el equipo se compromete a entregar al finalizar el sprint.
En caso de no contar con las **incidencias refinadas**, se deberán refinar en este momento previo al inicio del sprint. **Pero se recomienda realizar esta tarea en el refinamiento del sprint anterior.**
Es importante entender que cada planificación **es un compromiso para con el cliente.**
Ahora bien ¿cómo sabe el equipo la cantidad de incidencias que podrá seleccionar?
Simple, sabiendo que cada incidencia está estimada, el equipo deberá realizar el "corte" correspondiente en base a su capacidad de trabajo; para ello es necesario conocer / calcular la velocidad.



3. **Crear las sub-tareas técnicas**, respondiendo a **¿Cómo se van a desarrollar las incidencias?**
   El equipo descompone las incidencias en tareas técnicas (sub-tasks) que se puedan desarrollar en horas (como mucho una jornada laboral) y que cumplan con la **Definición de Hecho**.
   Esta tarea también puede ser realizada en el refinamiento del sprint anterior. En este evento participa todo el equipo Scrum y el/la PO.

Gestión durante el sprint

Recordemos que Scrum no hace mención de cómo gestionar el sprint una vez iniciado el mismo, por lo cual queda a criterio del equipo. Teóricamente, el sprint se compone de Incidencias funcionales (PBI - Product Backlog Items) y sus sub-tareas técnicas de la siguiente manera:

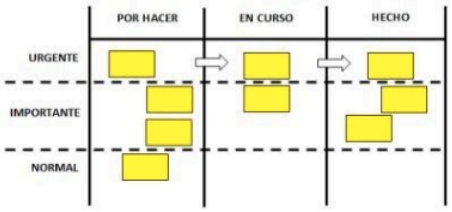


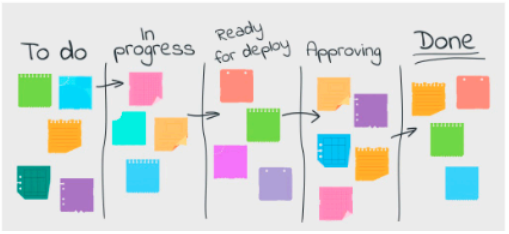
Como vemos la información no se encuentra organizada, por lo que tal como se mencionó previamente, una opción de organizar el trabajo y dar visibilidad del mismo, las incidencias se pueden agrupar en un **tablero Kanban** y así **el equipo tendrá siempre visible el estado de su trabajo.**

Importante a destacar: el tablero le debe ser útil y servir al equipo, **no a les managers**.
El tablero se puede dividir en tantas columnas como el equipo crea necesario. Cada integrante del equipo es responsable de actualizar el tablero, actualizando el estado de las incidencias según corresponda.
Kanban propone como básicas las siguientes columnas:

- TO DO (Para hacer)
- WIP (En Progreso)
- DONE (Hecho)

Otra columna necesaria podrá ser la que refleje el estado de la etapa de prueba, por ejemplo con el nombre IN TESTING (en prueba) o similar. **Veamos algunos ejemplos:**





Un punto importante es que el equipo **acuerde** los criterios necesarios para transicionar las incidencias entre las columnas de estado, y así evitar que cada integrante aplique su propio criterio sin un consenso general.

Acciones a tener en cuenta dentro del Sprint:

- No se hacen cambios que pongan en peligro el Objetivo del Sprint **(Foco)**
- La calidad no disminuye **(Compromiso)**
- El Product Backlog se refina según sea necesario **(Adaptación)**
- El alcance se puede clarificar y renegociar con el/la PO a medida que se va aprendiendo **(Coraje)**

Daily Scrum

Este evento se denomina “Diaria” y su principal **objetivo** es generar **comunicación valiosa** para el equipo, inspeccionando el progreso del sprint hacia su objetivo y adaptando el sprint backlog según sea necesario (el tablero). Además es el momento de dar visibilidad a los impedimentos y retrasos antes que sea demasiado tarde **(Compromiso, Coraje y franqueza).**
Adicionalmente, un equipo auto-organizado se beneficia mucho teniendo instancias de sincronización frecuentes

para evaluar el progreso y tomar decisiones de replanificación.
Dado que los términos “comunicación” y “valiosa” son conceptos abstractos, Scrum propone como evento un muy breve encuentro, de no más de 15’ en el cual cada integrante del equipo comunique a sus compañeros/as:

1. ¿Qué tarea terminó en pos de lograr el objetivo?
2. ¿Con qué va a comenzar para ayudar a lograr el objetivo?
3. De haber ¿qué impedimentos / bloqueos tiene que impiden lograr el objetivo?

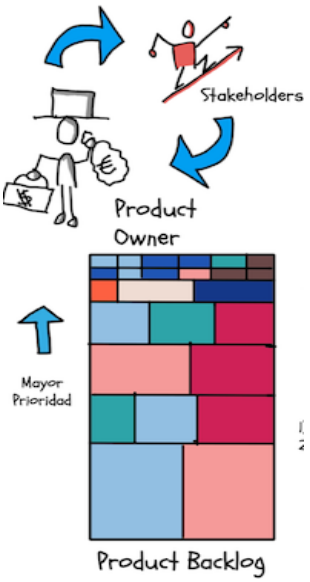
Tener en cuenta el objetivo del evento **y no su forma**. Scrum sólo propone una manera de lograrlo, pero no es la única, cada equipo deberá encontrar la manera de comunicar la información que crea necesaria. Como primer ejercicio, es recomendable seguir la propuesta del marco.
En este evento participa todo el equipo Scrum, el/la PO pueden no participar si no se cree necesario.

Refinamiento (No es un evento oficial, es una actividad muy necesaria)

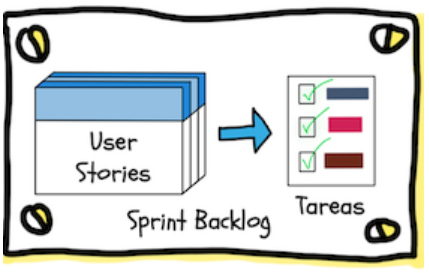
Scrum no reconoce formalmente al refinamiento como un evento, sino más bien como una actividad que permite adelantar el trabajo pendiente. El refinamiento se realiza sobre el Product Backlog, por lo que es responsabilidad del PO y tiene como objetivo **adelantar trabajo de cara al próximo sprint**. El refinamiento tiene 2 enfoques: funcional y técnico. Es el Equipo Scrum (ambas partes) quien decide cuándo y cómo se realiza esta actividad.
Se busca profundizar en el entendimiento de las incidencias que se encuentran más allá del Sprint actual y dejarlas preparadas para el próximo sprint, según el D.O.R.
Como sabemos que Scrum es un método que se basa en proyectos cambiantes, gran parte de sus tareas implica mantener el product backlog actualizado según los nuevos requerimientos.

Esta actividad suele tener 2 enfoques diferentes:

1. Funcional: el/la PO debe trabajar constantemente sobre las incidencias del Product Backlog a nivel funcional en pos de cumplir con el objetivo del producto. No debe intervenir en el Sprint Backlog. Muchas veces será necesario que consulte a les stakeholders o al/la cliente para constatar o averiguar información. Esta tarea consiste en crear incidencias, eliminarlas, editarlas, completarlas, o dividirlas en más pequeñas (slicing de épicas), además de mantener el PB priorizado. Idealmente se revisan y detallan aquellas incidencias que potencialmente se encuentren involucradas, son candidatas, para los próximos dos Sprints.

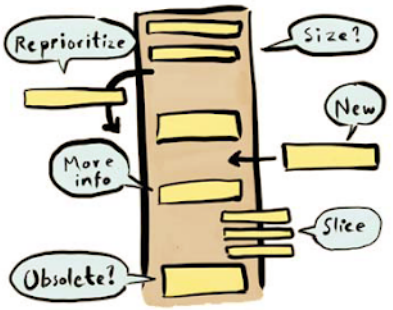


2. Técnico: el/la PO se reúne con el equipo de desarrollo para comunicar mejoras y revisar la viabilidad de su desarrollo. Luego, una vez que las incidencias se encuentren refinadas funcionalmente hablando, el equipo de desarrollo deberá crear las tareas técnicas (sub-tasks) asociadas a cada incidencia, para finalmente poder estimarlas.
   En caso de no realizar el refinamiento técnico, las tareas involucradas se deberán realizar en el evento Planning, el cual probablemente llevará más tiempo. Por todo esto, es que el equipo es quien decidirá qué conviene realizar en cada momento. La ventaja de refinar técnicamente un sprint antes, es que le damos tiempo al/la PO para que releve información, y así llegar a la planning con las incidencias completas, ya sea para poder estimar y finalmente poder incluirlas en el sprint.



¿Qué tareas se pueden realizar en un refinamiento (funcional/técnico)?:

1. Crear nuevas incidencias
2. Dividir épicas en historias de usuario
3. Modificar incidencias, ya sea actualizando o agregando información nueva
4. Completar las historias de usuario con criterios de aceptación
5. Eliminar incidencias deprecadas, duplicadas, etc
6. Re-Priorizar el orden de las incidencias
7. Crear sub-tareas técnicas
8. Estimar incidencias

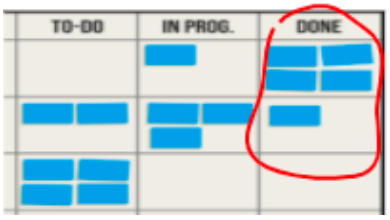


Nota: claramente no siempre se deben realizar todas estas acciones juntas, sino las necesarias. Por otro lado, es importante destacar que, si bien el/la PO es quien realiza la mayoría de las tareas, el refinamiento sirve como espacio de comunicación entre todas las partes, para que el equipo **pueda evacuar todas las dudas y adelantarse a posibles complicaciones técnicas o de definición**. Así, al realizarla con anticipación se podrá contar con tiempo para relevar información necesaria o investigar cuestiones técnicas.

Sprint Review

Al finalizar el sprint se realiza el evento Sprint Review que se divide en 2 instancias:

- Review del sprint: el propósito de la revisión del Sprint es inspeccionar el resultado del Sprint en términos de gestión y determinar futuras adaptaciones. El equipo observa el Sprint Backlog para evaluar el incremento funcional potencialmente entregable (el "qué").



En esta instancia se deberá cerrar el sprint, para poder comenzar con uno nuevo en la próxima planning. Para ello será necesario primero revisar el tablero del sprint backlog y en caso que hayan incidencias cuyo estado se encuentra desactualizado, es el último momento para realizarlo. Una vez cerrado el sprint, se deberá revisar el gráfico de Burndown Chart**.** Otra tarea a realizar en este evento es la de **actualizar** la velocidad del equipo en base a los puntos quemados.
Participa todo el equipo scrum pero sin el cliente y Stakeholders. Las PO's muy involucradas suelen participar también.

- Review del producto: el equipo Scrum presenta a las partes interesadas los resultados de su trabajo y se discute el progreso hacia el **Objetivo de Producto**. Así, el equipo de desarrollo provee al cliente/PO, el **incremento** según el objetivo planteado en la planning. Las partes interesadas **revisan** lo que se logró en el Sprint y lo que ha cambiado en su entorno. En base a esta información, todas las personas colaboran en qué hacer a continuación y se **obtiene feedback**, ya sea aceptando o rechazando las funcionalidades construidas. Este feedback puede darse sobre cambios en las funcionalidades o sobre nuevas funcionalidades que surjan al ver el producto en acción. Las cuales deberán crearse en el product backlog con su prioridad correspondiente.

Importante: las partes interesadas son quienes utilizan el producto, mientras que el equipo se dedica a anotar el feedback obtenido. Esta es una diferencia con respecto a la "Demo" tradicional, donde el equipo hace una demostración en lugar que el cliente "juegue" con su propio producto. El Sprint Review es una sesión de trabajo y el equipo debe evitar limitarla a que se convierta en una simple presentación.
En esta instancia del evento participa todo el Equipo Scrum, más todas las partes interesadas (clientes y stakeholders).

Retrospectiva

Como último evento, posterior al Sprint Review, se debe realizar la Retrospectiva. El objetivo de este evento es que todo el equipo tenga un espacio de introspección sobre su proceso de aprendizaje para tomar acciones de mejora de cara al próximo sprint, y así, evitar seguir repitiendo errores. La retrospectiva es el corazón de la mejora continua. El equipo Scrum inspecciona cómo fue el último Sprint con respecto a individuos, interacciones, procesos, herramientas y su definición de Hecho.

No se indaga sobre cuestiones técnicas.
Para esto, el/la SM debe preparar actividades que permitan al equipo reflexionar sobre todo lo sucedido durante el sprint e indagar sobre los problemas e inconvenientes atravesados.
Pero no todo debe ser negativo, también es importante saber identificar los logros y los hechos buenos que sucedieron; tomarse el tiempo para celebrar y entender qué acciones son importantes continuar haciendo, dado que aportan valor.

Valiéndose de técnicas de facilitación y análisis de causas raíces, se buscan tanto fortalezas como oportunidades de mejora. Luego, el Equipo Scrum decide por consenso **cuáles serán las acciones** de mejora a llevar a cabo en el siguiente Sprint. Estas acciones y sus impactos se revisarán en la próxima retrospectiva.

El/la SM deberá planificar la retro contemplando las siguientes fases/actividades:

1. Actividad de conexión / precalentamiento
2. Revisión de las acciones de la retro anterior para saber si fueron efectivas
3. Identificación de problemas e indagación sobre sus posibles causas
4. Planteamiento de acciones de mejora y votación para priorizarlas

Para profundizar sobre las actividades, veamos cada una en más detalle:

1. Armar el escenario
Es una forma de entrar en calor y conectar con las actividades siguientes

2. Recolectar datos
Brainstorming de cómo fue el sprint (cosas buenas y malas)

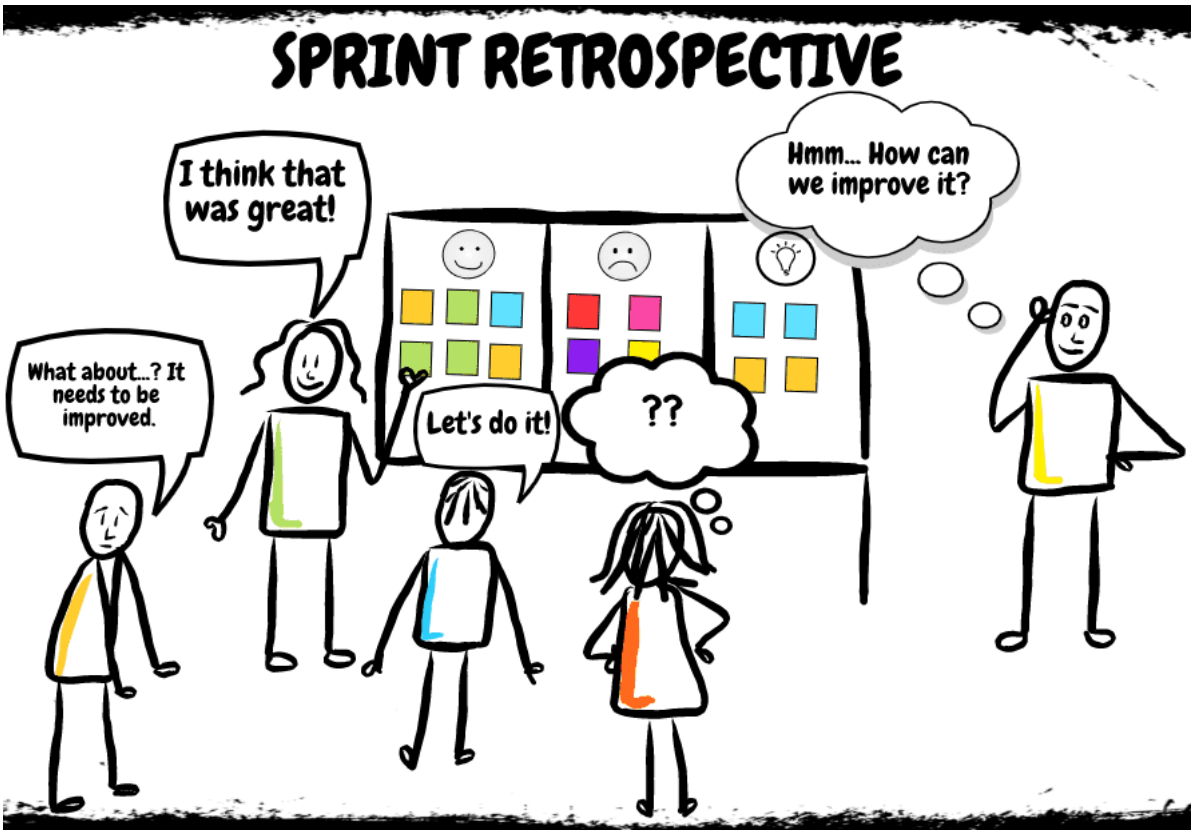
3. Indagar
Se investiga las causas / raíces de las problemáticas planteadas en la etapa de recolección

4. Decidir qué hacer
Momento de toma de decisiones. Definir y seleccionar acciones a realizar durante el sprint

5. Cerrar la retro
Espacio para que el equipo se vaya motivado

A continuación les comparto el sitio Retromat con ideas para dichas actividades.
Es importante que el/la SM propicie un espacio de confianza, donde cada integrante del equipo se pueda abrir a sus compañeros/as, exponiendo sus sentimientos y opiniones; por lo tanto, para que realmente funcione, será sumamente importante que cada integrante participe desde un lugar respetuoso, abierto a la escucha y con honestidad sobre cada actividad propuesta, siendo que es muy probable que se toquen temas sensibles para ciertas personas. En este evento participa todo el equipo y es posible que también se invite a el/la PO, dependiendo la confianza que se tenga y cuán involucrada se encuentre esta persona dentro del equipo.

Una aclaración importante, en esta oportunidad **el foco está puesto en el equipo y su proceso de trabajo**, **no en el producto**; durante todo el sprint se le dedicó tiempo al producto y los temas técnicos. Si bien, ésto no implica que no se puedan detectar cuestiones técnicas, en dicho caso, se debe identificar de manera clara cómo afecta dicho problema al proceso de trabajo, el punto técnico debe ser la causa de la problemática; en caso contrario, no es un punto que se deba revisar en este espacio. Con esto me refiero a que **no se deberán discutir o debatir resoluciones técnicas** en esta instancia.



SMART: Características de las acciones

Para que las acciones sean efectivas, su definición debe cumplir con el acrónimo SMART, el cual representa que debe ser:

Specific (Específica): no puede ser ambigua, ni genérica, se debe poder concretar

Mensurable (Medible): se debe poder medir para saber si fue efectiva

Achieve (Alcanzable): debe ser factible, nada muy ambicioso ni utópico

Relevant (Relevante): debe ser importante y aportar valor, sino no tiene sentido la acción. ¡No perdamos tiempo!

Timely (Temporal): se le debe asignar una fecha de caducidad (due-date) para realizarla antes de la misma.

Veamos algunos ejemplos:

Specific:

- Aprender idiomas ✗ -> Estudiar inglés en una academia online ✓
- Mejorar la comunicación ✗ -> Reunirnos para definir un proceso de comunicación ✓

Mensurable:

- Mejorar el backlog ✗ -> Contar con al menos 10 US antes de cada refinamiento ✓

Achieve (Alcanzable):

- Refactorizar todo el proyecto en 2 semanas ✗ -> Refactorizar las US "a", "b" y "c" en 2 semanas ✓

Timely (Temporal):

- Actualizar el tablero Kanban ✗ -> Actualizar el tablero Kanban todos los días antes de la daily ✓
- Leer más ✗ -> Leer 30' antes de dormir ✓

¡Tip importante!
Evitar las palabras genéricas como "Mejorar", "Evitar", "Tratar", etc

ACLARACIÓN: no hay ejemplo de **Relevante**, dado que es una característica muy subjetiva que depende de la visión del negocio.

UNIDAD 4: DOCUMENTACIÓN

CASCADA: ESPECIFICACIÓN FUNCIONAL

El desarrollo de software mediante la metodología tradicional inicia con la etapa de análisis en la cual se genera un documento que detalla todos los requerimientos/requisitos del cliente sobre el producto a desarrollar. Este documento se llama "ESPECIFICACIÓN FUNCIONAL" y funciona como una especie de "contrato" entre el cliente y los roles involucrados en el ciclo de vida del software. Este documento suele ser bastante extenso y con información detallada en texto y acompañada con diagramas o gráficos que ayudan al entendimiento de cada funcionalidad. Para esto se utilizan los "CASOS DE USO", los cuales organizan la información de una manera más amigable para su entendimiento.
Es una técnica que muestra la interacción entre los usuarios y cada funcionalidad de la aplicación. Para enriquecer la comunicación mediante diagramas se utiliza la herramienta "UML".

CASOS DE USO (CU)

¿QUÉ ES UN CASO DE USO?

- Es una técnica para capturar requerimientos funcionales de un sistema (Fowler)
- Un contrato entre los interesados (stakeholders) sobre el comportamiento esperado de un sistema (Cockburn)
- Generalmente es información narrada mediante texto orientada al entendimiento funcional del sistema.

¿PARA QUÉ SE UTILIZA?

Para describir las acciones o actividades. Un caso de uso expresa las acciones que deberá realizar alguien o algo para llevar a cabo algún proceso dentro del sistema. Para sistemas medianamente grandes, se implementa un modelo de casos de uso, el cual representa el **conjunto** de interacciones que se desarrollan dentro del sistema como respuesta a eventos que inicia un actor principal. Son útiles para especificar la comunicación y el comportamiento de un sistema según la interacción con todos sus usuarios. **Este modelo permite:**

- Estimar el tamaño del sistema
- Definir el alcance del sistema
- Armar los casos de prueba
- Diseñar y desarrollar

Elementos de un caso de uso:

El detalle de la información a brindar en un caso de uso dependerá de los procesos, estándares y templates que cuente cada organización. A continuación veremos la información comúnmente detallada en un caso de uso.

- ID y Nombre
- Actores
- Funcionalidades
- Escenario
  - Flujo principal de éxito
  - Pasos
  - Flujos Alternativos
  - Flujo de Excepción
- Pre y Post Condiciones
- Detalles de Diseño o Implementación

Veamos cada uno:

1. **Actores**: cualquier elemento que necesite interactuar con el sistema.
   Puede ser:
   - Una persona (usuario)
   - Un módulo del sistema
   - Otro sistema

   Básicamente un actor se refiere a un **rol específico** de interacción con el sistema, por lo tanto pueden haber tantos actores con individuos a interactuar con las funcionalidades.
   Hay 2 tipo de actores:
   - Primario: aquel que inicia el caso de uso
   - Secundario: aquel que interviene internamente dentro del caso de uso.

2. **Funcionalidades**: son las acciones funcionales que podrá realizar un actor dentro del sistema. Y como tal se describen mediante verbos. Ejemplos:
   - Aprobar Pedido
   - Completar Orden de Compra
   - Generar Facturación
   - Realizar Extracción

   En general, la funcionalidad inicial que dispara el actor primario dentro del sistema es la que le da el nombre al caso de uso. El ID suele ser un incremental ascendente.

3. **Escenario**: describe la secuencia de pasos que debe realizar un actor para realizar una funcionalidad. No contienen condicionales (Si pasa A entonces B sino... C).

Ejemplo:
"Un cliente llega a un cajero, ingresa la clave, el cajero presenta las opciones y el cliente selecciona extracción y luego indica que quiere retirar 300 pesos. A continuación, el cajero extrae los 300 pesos e imprime el ticket correspondiente."

a. Un Paso es cada una de las interacciones de un actor o del sistema. Es la mínima unidad de escritura en un escenario. Típicamente una oración simple.
b. El flujo principal de éxito es aquel que describe los pasos de la funcionalidad completa desde su inicio hasta lograr el objetivo de manera exitosa.
c. Los flujos alternativos describen los pasos de las funcionalidades particulares y/o flujos alternativos de la funcionalidad.
d. Los flujos de excepción describen los pasos a ejecutar como parte de una excepción, tal es el caso de los errores.

4. Pre y Post-condiciones:
Las pre-condiciones son las condiciones previas para poder ejecutar el caso de uso, y las Poscondiciones definen el estado posterior a la ejecución de un caso de uso en condiciones exitosas.

5. Detalles de Diseño o Implementación: sección donde se detallan los campos, formatos, validaciones y demás información para su implementación.

Relación entre CU
Un caso de uso se puede **relacionar** con otro caso de uso, mediante links dentro del documento.
Ejemplos:
1. El cliente Realiza una Búsqueda del Producto que desea comprar
2. El cliente Realiza una Búsqueda del Producto que desea comprar (REF:CU23 – Buscar Producto)

Ejemplo de un caso de uso:
Nombre: Conversar vía telefónica

Actores: Usuario

Escenario principal de éxito: el usuario del teléfono levanta el auricular y marca el número de destino. El sistema conecta o indica error de conexión. Una vez conectado, el usuario conversa hasta que cuelga, lo que da fin a la conexión.

Flujo Principal:

1. **Usuario**: Levanta el auricular.
2. **Sistema**: Da el tono de marcado.
3. **Usuario**: Indica el número de teléfono.
4. **Sistema**: Realiza la conexión. Da tono de aviso en tanto se levanta el teléfono del lado contrario de la conexión. Permite la conversación al hacerse efectiva la conexión.
5. **Usuario**: Conversa y al finalizar cuelga el teléfono
6. **Sistema**: Termina la conexión

Flujo alternativo:

1. **3a - Usuario:** Número incorrecto
2. **4a_1 - Sistema**: Presenta tono de error y la comunicación termina.

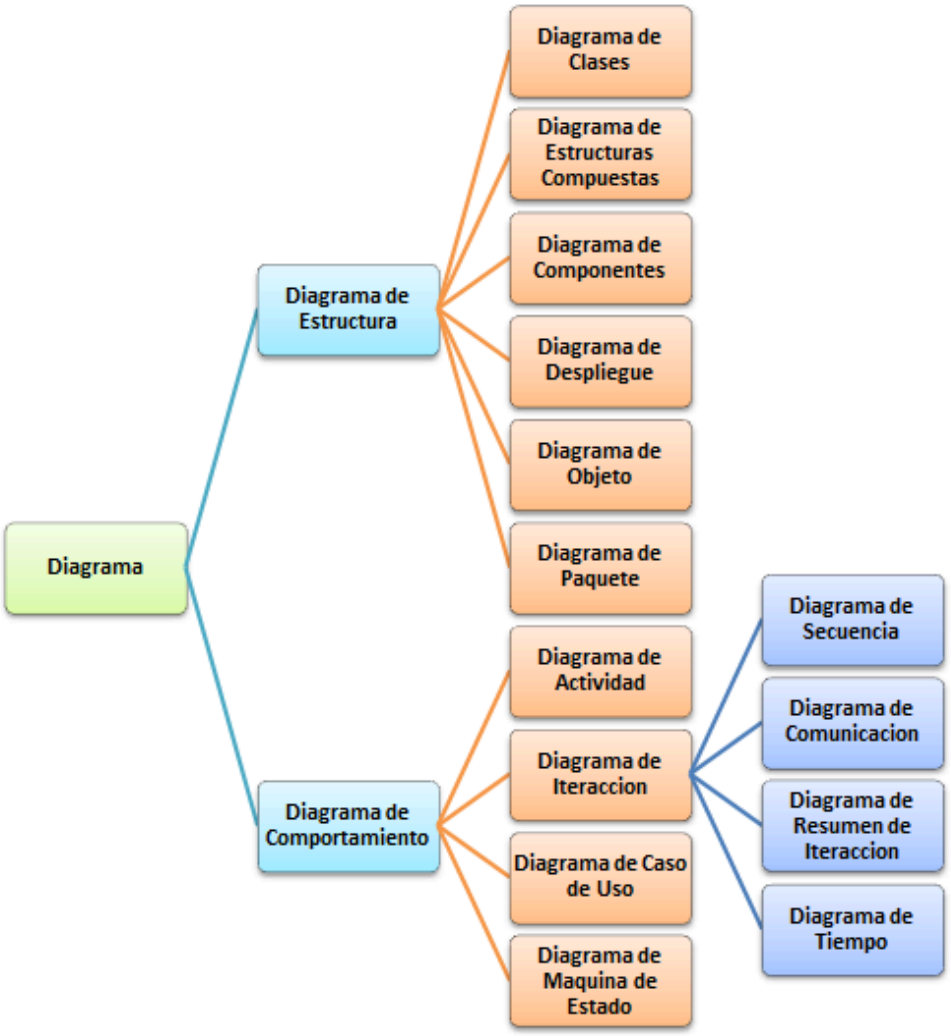
Precondición: El teléfono está colgado y tiene tono.

Poscondición: Ninguna.

UML (Lenguaje de modelado unificado)

Si bien los casos de uso se utilizan para expresar los requerimientos de manera más organizada, no dejan de ser información textual, por tanto, como una manera de enriquecer aún más la especificación funcional, es que se comenzaron a describir de manera gráfica utilizando diagramas mediante el lenguaje UML.

La versión 1.0 de este lenguaje fue liberada por el grupo “Object Management Group” en 1997. Como todo lenguaje posee su propia sintaxis y semántica que permite modelar los requerimientos para la interacción entre los artefactos involucrados (actores, CU, objetos, etc).
UML es un lenguaje para crear modelos y es independiente de los métodos de análisis y diseño. Existen diferencias importantes entre un método y un lenguaje de modelado. Un método es una manera explícita de estructurar el pensamiento y las acciones de cada individuo. Además, el método le dice al usuario qué hacer, cómo hacerlo, cuándo hacerlo y por qué hacerlo; mientras que el lenguaje de modelado carece de estas instrucciones. Los modelos (descriptos en algún lenguaje), simplemente son utilizados para describir algo y comunicar los resultados de su uso aplicados dentro de un método.
Este lenguaje permite modelar la información de distintos enfoques y para lo cual propone diversos diagramas. Los diagramas posibles a desarrollar son:



Observar que cada diagrama acompaña y enriquece la documentación correspondiente a las distintas etapas del ciclo de vida del software. Puntualmente, dentro de la etapa de análisis, como parte de la información relevada de un/a Analista Funcional, la especificación utilizará los diagramas de Casos de Uso.

Estos diagramas cuentan con los siguientes elementos de modelado:

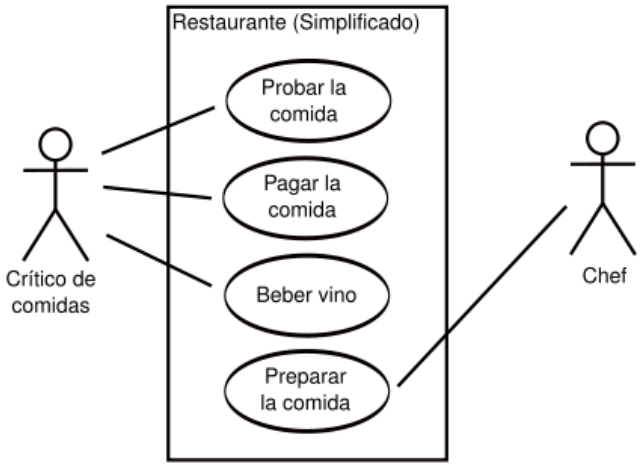
- Actores
- Relaciones
  - Generalización
  - Especialización
- Casos de usos

El lenguaje respeta las definiciones propuestas por la técnica de casos de uso agregando solamente el elemento “Relación”.
Una relación en UML es la conexión existente entre los distintos artefactos del diagrama.

Simbología

- **Actor**: un dibujo de ser humano en formato de palitos
- **Relación**: flecha o línea simple
- **Caso de Uso**: óvalo con el nombre del CU en el centro del mismo
- **Alcance**: el rectángulo delimita el alcance del sistema, es decir su límite.

Ejemplo:



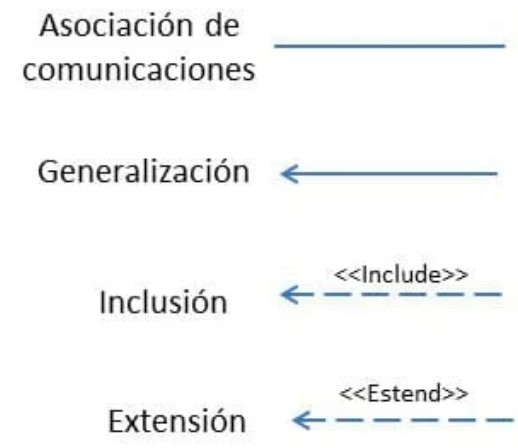
Tipos de relaciones

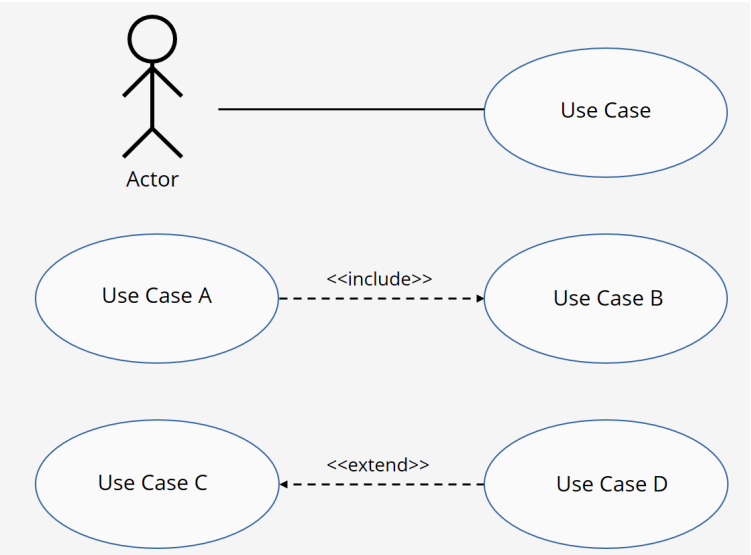
Una relación además de asociar, puede ser una generalización, o una especialización, la cual a su vez se divide en extensión o inclusión.

- ➢ **Asociación de comunicaciones:** un actor se **relaciona** con un caso de uso
- ➢ **Generalización:** un caso de uso **hereda** el comportamiento de otro caso de uso. Es el padre de la relación.
- ➢ **Especialización:** es el hijo de la relación
  - **Extensión:** un caso de uso base incorpora **implícitamente** el comportamiento de otro caso de uso extendiendo su funcionalidad para flujos alternativos. La reutilización que requerimos agrega funcionalidad pero no altera al caso base.
  - **Inclusión**: un caso de uso base incorpora **explícitamente** el comportamiento de otro caso de uso en algún lugar de su secuencia. La relación de inclusión sirve para enriquecer un caso de uso con otro y compartir una funcionalidad común.

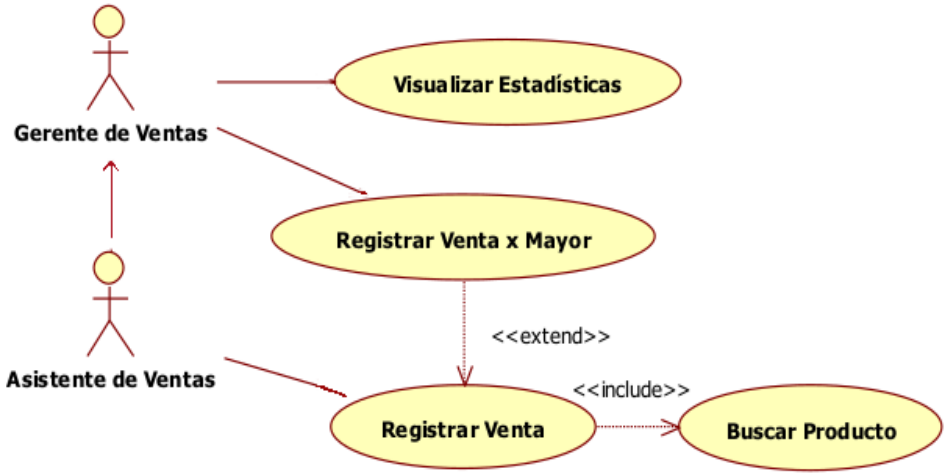
El caso de uso incluido existe únicamente con ese propósito, ya que no responde a un objetivo de un actor. Es decir, que éste es parte esencial del caso base. Sin el segundo, el primero no podría funcionar bien.

Veamos su simbología:

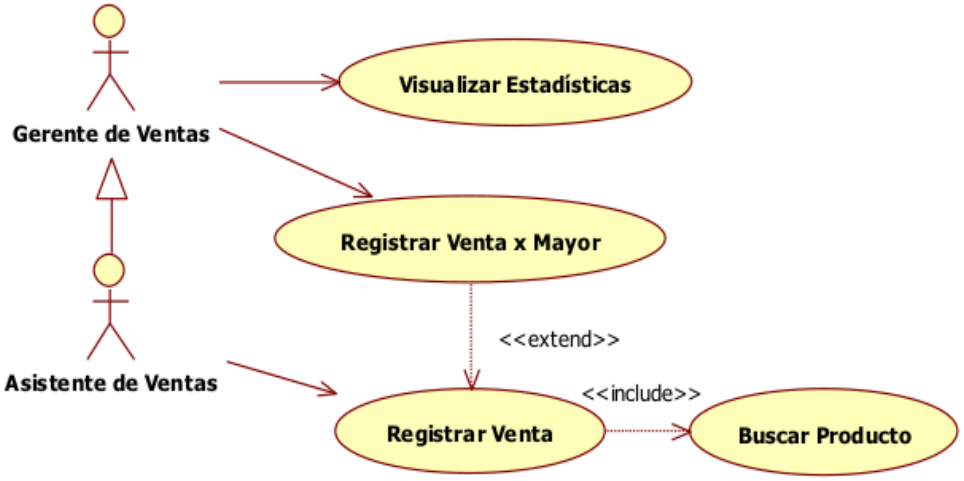




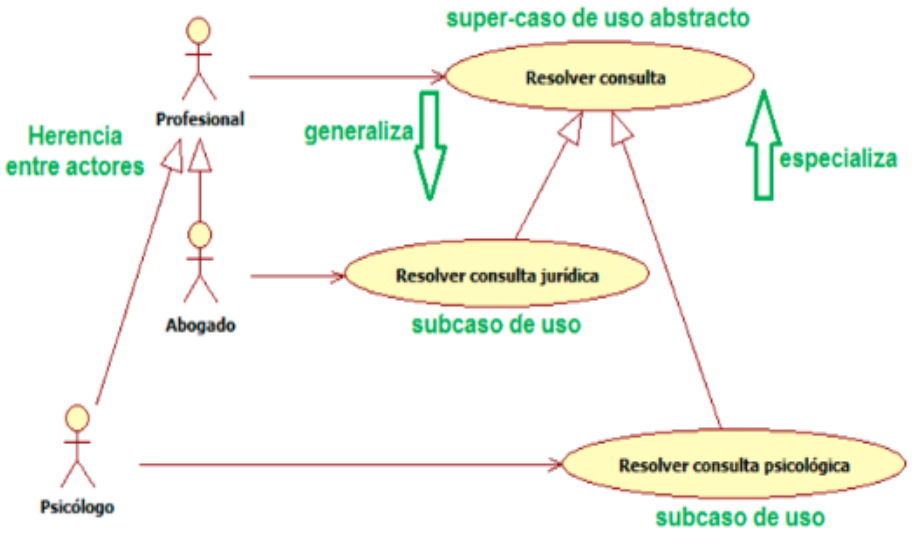
Aclaración: en algunas herramientas la asociación se identifica con una flecha y la generalización con una flecha cuya punta es más ancha.



Mismo ejemplo con la otra simbología:



Veamos cómo entender la diferencia entre generalización y especificación

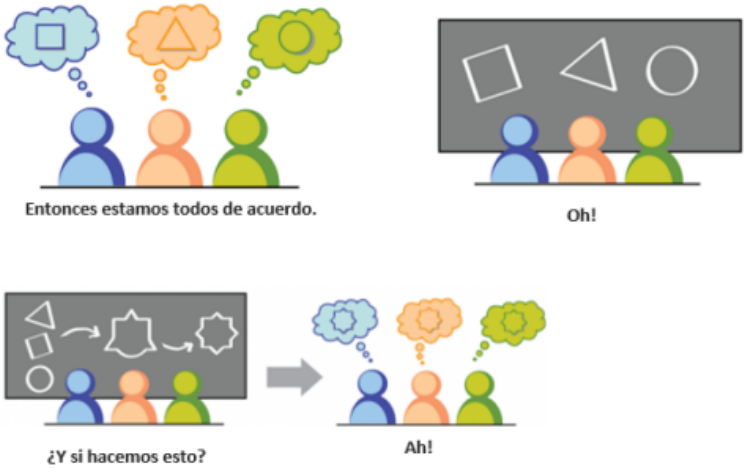


Ágil: Product Backlog

Antes de comenzar me gustaría derribar un mito: *"En la metodología ágil no se documenta"*. Esto no es verdad, puesto que sí se documenta, sólo que de una manera diferente, asociada a la dinámica de un proceso de desarrollo iterativo e incremental. Por lo cual, la documentación extensa **inicial** tal cual la conocemos en la metodología tradicional no existe. El método ágil va creando y acumulando documentación con cada incremento del producto durante cada iteración.
Amén de esto, ya sabemos que el desarrollo de un producto es parte de un proyecto, y cada proyecto tiene una etapa o fase inicial, en la cual se deberá entender y visibilizar las expectativas del/la cliente, el alcance del producto y los roles necesarios para su desarrollo. Acuerdos que deberán quedar asentados en documentos que darán inicio al proyecto.

Inception: Documentación inicial

El método Scrum propone un *evento específico* para plasmar los acuerdos del inicio de un proyecto. Este evento se llama: *Inception.*
La inception no es más ni menos que una reunión inicial en la que participan todas las personas involucradas en el proyecto (partes interesadas), desde el cliente y responsables de gestión, hasta los responsables técnicos, con el objetivo de fomentar la comunicación y acuerdos entre todas las personas para que construyan, en conjunto, una visión compartida del producto.



Durante la misma se realizan varias actividades que permiten a todas las personas entender la necesidad y el objetivo principal del proyecto, definir el alcance (que cosas sí serán parte del producto y cuáles no), definir prioridades (tiempo, costo, calidad, etc), analizar la viabilidad técnica y de calidad, y finalmente detallar la información funcional del producto. Si bien se necesita documentar toda la información (puede ser un repositorio de fotos), es fundamental que quede documentado este último punto para comenzar con el desarrollo del producto. Para ello es que se realiza una técnica que se la conoce como *User Story Mapping*.
Para proyectos medianos y grandes, suele durar una jornada de trabajo.

Objetivo del producto (Product Goal) (compromiso)

Técnica Elevator's pitch

Como primera actividad se recomienda realizar el *Elevator pitch*, la cual tiene como foco principal definir la **visión del producto u objetivo del producto (Compromiso)**. Es importante iniciar la inception con esta definición para que todas las personas tengan el mismo norte y sus aportes confluyan hacia el mismo objetivo. Siendo que la visión de un producto, es parte de la identidad del mismo, no suele ser una tarea fácil de definir y acordar entre todas las partes; es por ello que se suele utilizar el siguiente template como guía:

- Para [cliente objetivo]
- Quienes [Necesidad y/o Oportunidad]
- El [Nombre del Proyecto]
- Es un [Categoría del producto]
- Que [Beneficio clave ,razón para comprarlo].
- Diferente a [Alternativa Competitiva]
- Nuestro Proyecto [declaración de la diferencia].

De esta manera la visión / objetivo del producto quedará definida en una frase simple y concreta que contará lo necesario para generar impacto.

La actividad se llama **Elevator's pitch** como analogía de vender la idea en lo que dura el recorrido de un ascensor (de un edificio de al menos 10 pisos); donde el concepto de "pitch" proviene del béisbol, en el cual se "lanza" la pelota; en este caso el objetivo es lanzar la idea del producto a un/a posible inversora para lograr financiamiento sobre su desarrollo. Es sumamente importante tener bien en claro la idea, qué valor aporta y cuál es su diferencial con respecto a la competencia, para poder venderla en tan sólo pocos minutos y que genere impacto y la atención en la otra persona.

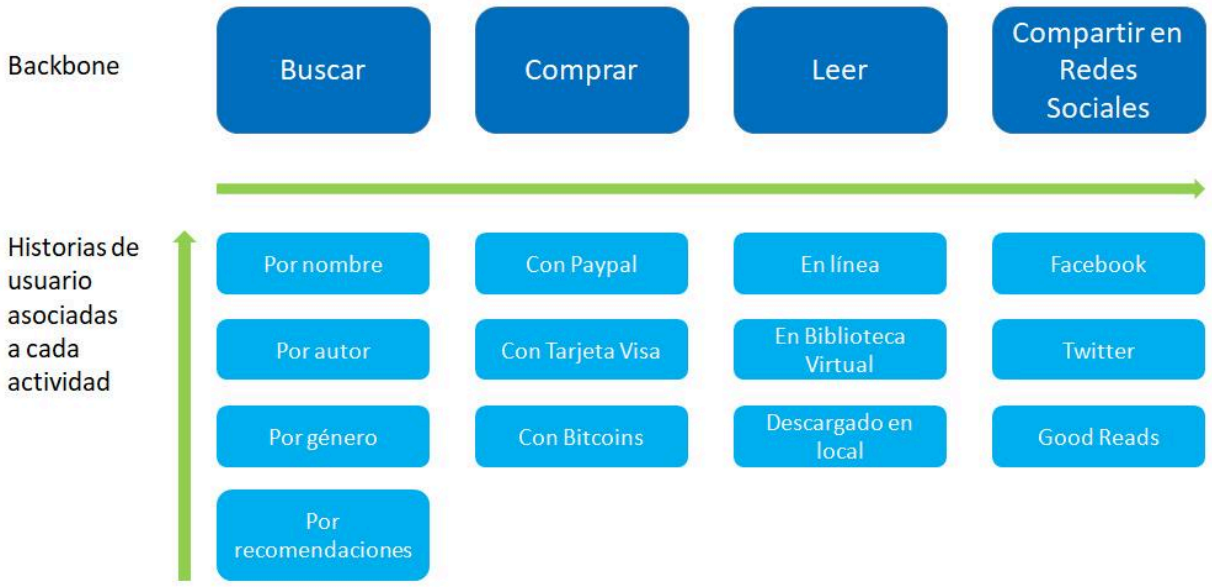
Veamos un ejemplo:

Para ***el viajero frecuente***
quien ***necesita balancear sus compromisos personales y profesionales***
el ***iTeAviso***
es una ***app. Móvil***
que ***combina tus calendarios personales y profesionales para que no te pierdas ningún compromiso y estés siempre ahí***
A diferencia de ***las aplicaciones para organizarte***
nuestro producto ***te dará descuentos en los principales restaurantes, hoteles, aerolíneas y servicios de transportación. Te hará la dueña de tu tiempo y tus compromisos.***

USM (User Story Mapping)

Es una técnica creada por Jeff Patton que permite ordenar visualmente las funcionalidades de un producto en formato de mapa bidimensional. Esto es, primero se detallan las funcionalidades generales (actividades) como columnas, denominadas Backbone (columna vertical) formando así el customer journey map (camino feliz) del producto. Es sumamente importante no perder de vista el objetivo del producto definido en el elevator's pitch.

Luego, debajo de cada funcionalidad se detallan las epicas o Historias de usuario que le van a permitir al usuario llevar a cabo dicha funcionalidad del backbone. Éstas deberán estar priorizadas según el valor que representen al producto / cliente, de manera decreciente, quedando en el tope aquellas que aportan más valor.

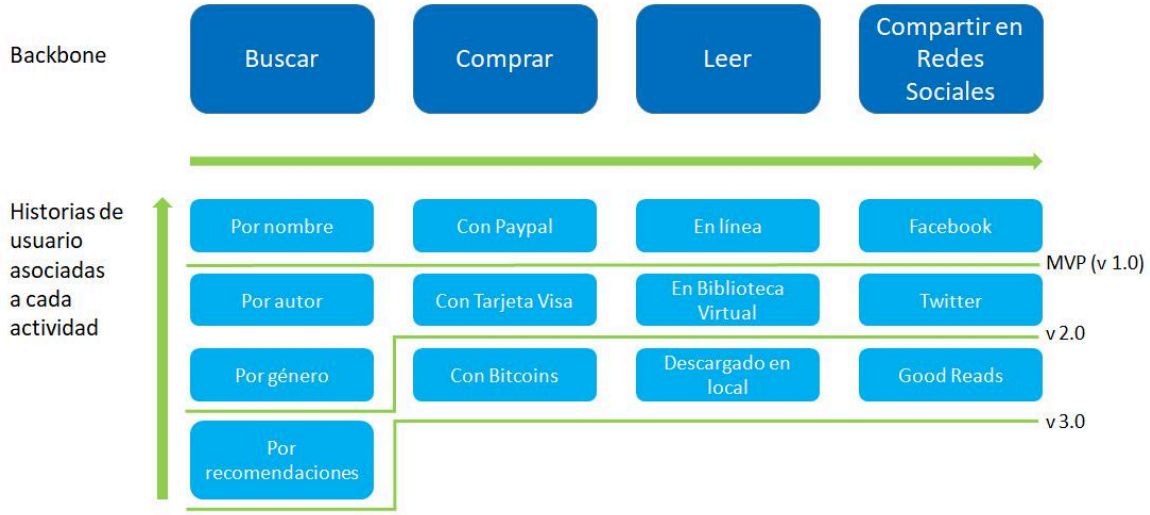


La idea del USM es plasmar cual brainstorming todas las funcionalidades posibles del producto, pero una vez finalizado, será momento de organizar toda la información para así obtener las diferentes versiones del producto. A la primera versión, que será la mínima necesaria para salir a producción se la identifica como *MVP*.

MVP (Minimum Viable Product)

El MVP representa la versión 1.0 del producto, dado que define el **Producto Mínimo Viable**. Esto es, se trata de un **producto** (no es una idea), es **mínimo** porque tiene las funcionalidades elementales pero necesarias, y es **viable** porque cumple con el objetivo inicial definido en el resto de las actividades de la inception.

Esto se logra, simplemente identificando dentro del USM cuáles son las funcionalidades que integran esta versión.



Esta actividad es la última que se realiza en el evento inception. Con esta documentación, el PO estará en condiciones de armar el Product Backlog para comenzar con el desarrollo ágil.
En caso de necesitar, con el backlog creado y la organización provista por el USM, se podrá realizar una estimación, a muy alto nivel en caso de contar con un equipo de cero, o más precisa en caso de contar con un equipo estable y que conoce su velocidad.

Documentación de iteraciones

Para poder crear este *Product Backlog (P.B)*, será necesario conocer los tipos de incidencias que puede contener. A cada elemento del PB, se lo denomina *PBI* (Product Backlog Item) o simplemente incidencia (issue).

Items Product Backlog (tipo de incidencias)

- Épica (epic)
- Historia de usuario (user story)
  - Sub-Tarea (sub-task)
- Tarea técnica (task)
- Error (bug)

Epic (Épica)

Una épica representa una funcionalidad que no se puede desarrollar en un sprint. Ya sea por su tamaño, es demasiado grande para ser entregada tal y como se ha definido en de una sola iteración, o por su ambigüedad e incompletitud en su descripción. En el primer caso, es lo suficientemente grande como para ser partida en pequeñas historias de usuario. Las épicas sólo describen un objetivo o necesidad funcional, no describe tareas técnicas, ni de gestión, ni de ningúna otra índole.
En el segundo caso, se trata de un mero título y/o una descripción con una información tan vaga que no permite conocer lo que se solicita, y por ende, no se puede desarrollar. Una épica nace de una necesidad grande y/o compleja y posiblemente bastante abstracta y genérica.
Para resolver el primer caso, la épica se deberá dividir en historias de usuario (US). Es decir que las US "nacerán" de esta épica; no es que se agrupan y se "asocian" a una épica (esto se llama Theme). Es la épica la que, al dividirse, genera nuevas historias de usuario; y como todo lo que se divide, desaparece como entidad. Cuando dividimos una épica la misma se reemplaza por el conjunto de US surgidas a partir de ella. Si bien este es un concepto teórico, dependiendo de la herramienta, la épica en ocasiones no se elimina, sino que queda linkeada a las US generadas para mantener una trazabilidad en la información.
Las épicas suelen describirse simplemente con un título que representa la "gran funcionalidad", ejemplo: "ABM de productos", "Venta de pasajes", "Facturación de ventas", "Módulo de liquidación", etc. En algunas ocasiones también pueden llevar una breve descripción muy genérica y ambigua. **No tienen la estructura de una historia de usuario.**

User Story (Historia de usuario):

Una User Story o historia de usuario es básicamente una funcionalidad lo suficientemente pequeña que aporta valor al producto y que se puede desarrollar dentro de una iteración.

Cada historia de usuario consta de 3 **etapas** conocidas como "Las 3 C":

Card
Conversation
Confirmation

Card (Ficha/tarjeta)

Básicamente es la descripción que representa a la "ficha" / tarjeta con la descripción de la funcionalidad. Se define QUÉ se necesita. Esta tarjeta debe contar con el siguiente formato:

Cómo <ROL>

Quiero/necesito <NECESIDAD>

Para <BENEFICIO>

En el campo **<rol>** se deberá detallar el rol que tiene la necesidad, por ejemplo *"Gerente de marketing"*, *"usuario logueado", "usuario sin loguear", "admin"*, "cliente", *"comprador/a", "vendedor/a"*, etc. Es importante, en caso que la app cuente con distintos perfiles de usuario, identificar cuál es el que tiene dicha necesidad. Para proyectos grandes con diversos tipos de usuarios, o usuarios poco convencionales, es interesante utilizar la técnica PERSONAS utilizada en UX y creada por Alan Cooper.

En el campo **<necesidad>** se detallará el requisito en cuestión de una manera **netamente funcional**. El/la PO no se focaliza (y en general desconoce) los conceptos técnicos. Por lo general no son personas desarrolladoras. Incluso de serlo, en este caso, si se trata de un producto funcional, las US deben contener la descripción de la funcionalidad.

Por último, en el campo **<para>** se deberá detallar **el motivo por el cual se está solicitando dicha funcionalidad**, es decir **¿para qué se quiere tal cosa?**. Este campo es **sumamente importante** y muchas veces omitido descartando así su valor, debido a la complejidad para completarlo. Dicha complejidad justamente obliga a analizar si la US que se está creando realmente es necesaria o es meramente una tarea "caprichosa".
Una manera de identificar este campo podría ser preguntarse **¿qué valor le va a aportar al rol que lo necesita?** o por el contrario, pensar **¿qué pasa si el producto no cuenta con dicha funcionalidad?**

Conversation (conversación)

Siendo que la Card suele tener una descripción bastante escueta para su desarrollo (programación y testing) será necesario contar con una etapa en la cual el equipo pueda completar la información faltante. Esto deriva de una charla con el/la PO, generalmente en el refinamiento y/o planning.

En esta instancia, el equipo básicamente se saca todas las dudas para terminar de comprender el requerimiento desde un lugar tanto funcional como técnico. Es importante completar la US con la información relevada, enriqueciendo así la documentación generada en el Product Backlog.

En ocasiones a esta etapa también se la considera como un momento de **negociación** entre las partes, dado que no siempre será viable el desarrollo (o testing) de una funcionalidad tal y como se requiere debido a cuestiones técnicas o de infraestructura. En dicho caso, el equipo podrá analizar y realizar una propuesta diferente que convenza a la PO. Eso sí, será necesario que la propuesta ofrecida siga aportando valor y cubriendo la necesidad de negocio.
Un punto importante: en la descripción de la Card se menciona sólo lo **qué** se necesita y no cómo se deberá realizarlo.

Confirmation (confirmación)

Como última instancia, a la Card se le agrega información sobre la manera en que se debe visualizar la funcionalidad. En esta etapa es donde sí se detalla el CÓMO. Esto implica detallar los criterios de aceptación (CA) a través de los cuales el/la PO confirma dicho requerimiento. Son criterios que harán que se confirme si la US efectivamente se desarrolló según lo esperado. Los mismos se detallan en un formato de lista (items /puntos / viñetas) y el rol de PO es quien conoce y debe completar esta información. En caso que el detalle de los CA sea muy complejo de describir, es muy útil agregar un mockup (boceto visual) de cómo se debe visualizar la funcionalidad en el producto.

Como parte de esta información se detalla:

- Información de estilo visual (texto o mockup)
- Validaciones con sus respectivos mensajes, ya sea de éxito o de error
- Relación con el resto de la app (link desde donde se accede)
- En caso de tratarse de una operación, se debe mencionar el mensaje **explícito** a mostrar luego de su ejecución
- Si hay archivos, indicar su formato
- Si hay respuesta de una búsqueda, especificar los campos a mostrar

Campos de una historia de usuario

Como hemos mencionado previamente, una US es un tipo de incidencias, y como tal se describe con los siguiente campos mínimos:

- ID
- Título: representa un breve resumen de la descripción, por lo cual debe ser un verbo que describa la funcionalidad
- Descripción: información de la card

- **Criterios de aceptación**: obtenidos en la etapa de la confirmación
- **Adjuntos**: para subir lo mockups
- **Estimación** (en puntos de historia): para especificar la complejidad de desarrollar (programación y testing) la US

INVEST: Características de una US

Para crear una US eficiente y asegurar la calidad de su escritura, contamos con un método creado por Bill Wake en 2003 que sirve para comprobar la calidad en base a una serie de características. Básicamente una US bien escrita debe cumplir con el acrónimo INVEST:

Independiente
Negociable
Valiosa
Estimable
Small (pequeña)
Testeable

- **Independiente**: es ventajoso que cada historia de usuario pueda ser planificada y desarrollada en cualquier orden. Para ello las historias deberían de ser totalmente independientes (lo cual facilita el trabajo posterior del equipo). Resaltar que las dependencias entre historias de usuario pueden reducirse combinándolas en una o dividiéndolas de manera diferente.

- **Negociable**: las historias deben ser negociables ya que sus detalles serán acordados entre el/la PO con el equipo durante la fase de conversación antes de su planificación. Por tanto, se debe evitar detallar información no relevada o confirmada tanto del lado funcional (revisión con cliente) como del lado técnico (revisión con equipo).

- **Valiosa**: como parte de la metodología, el valor es un componente esencial para un proyecto, por lo cual si la US no aporta valor al negocio, deberá descartarse.

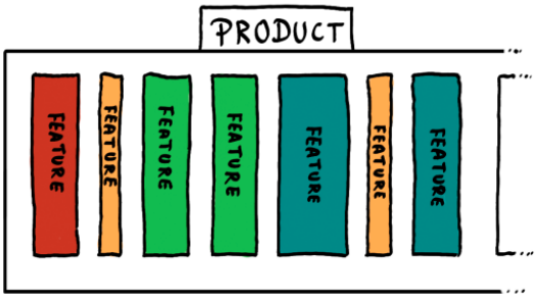
- **Estimable**: contar con una US estimada hace que el equipo pueda determinar si la misma se puede desarrollar en una iteración. La estimación además permite identificar si en realidad se trata de una épica en lugar de una US. En cuyo caso habrá que dividirla. Así como también permite conocer la velocidad del equipo.

- **Small**: la US si no es pequeña, no cumple con su definición. Básicamente se debe evaluar su tamaño al redactarla. Es muy probable que una US sea muy grande si:
  - Tiene mucho texto
  - Tiene muchos criterios de aceptación (CA)
  - El mockup tiene muchas funcionalidades
  - El título hace referencia a un módulo completo

- **Testeable**: si la US no detalla la información mínima para testear el caso feliz, no es una US que vaya a ayudar al equipo a aportar calidad, es una US incompleta. Esto es, será necesario cumplir con su definición agregando los CA correspondientes para su posterior validación.

Slicing de US ("División" de historias de usuario)

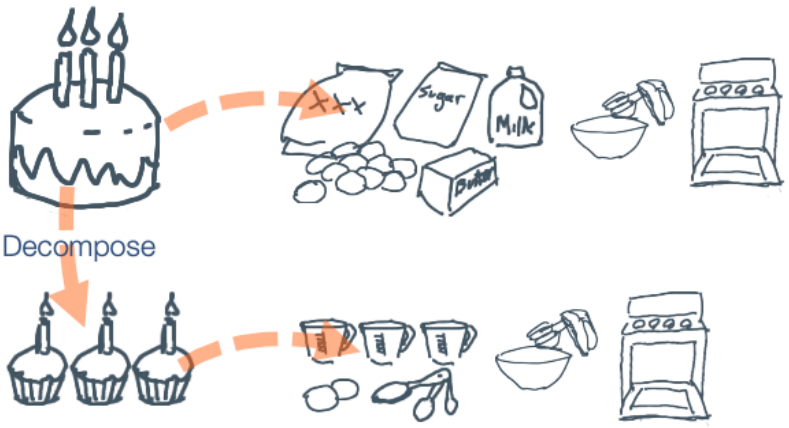
Si entendemos un producto de software como un conjunto de funcionalidades que creamos, evolucionamos o descartamos, podemos visualizar nuestro producto de la siguiente manera:



En esta metodología debemos entender que no existe final, salvo que retiremos el producto del mercado o decidamos no evolucionarlo más. Esta diferencia es vital, no estamos ante proyectos con fecha de inicio y fin, sino productos que se van incrementando con el tiempo en función de las necesidades que queramos cubrir o el problema que estemos resolviendo.
Habiendo aclarado / recordado este punto, será cuestión de pensar cómo vamos a trabajar con estas funcionalidades para su posterior desarrollo.
Como ya hemos mencionado, las épicas son quienes se van a "dividir" en US, pero antes de ahondar en la técnica adecuada, les dejo una breve aclaración.

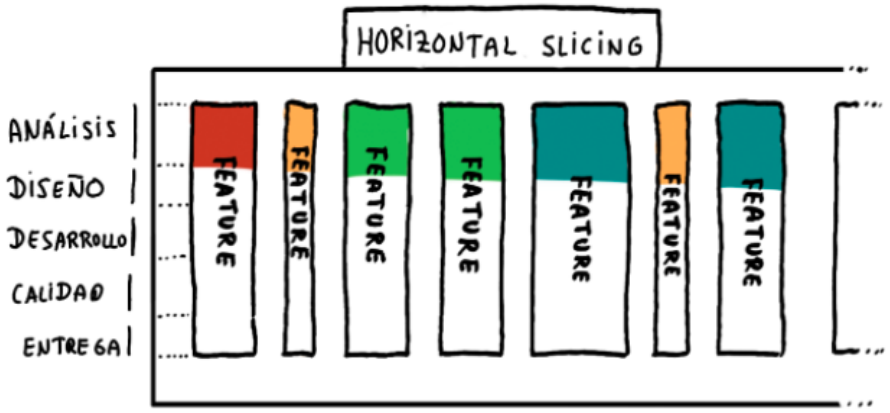
Notar que el término división se encuentra entre comillas, esto se debe a que si bien la traducción literal del término conlleva a dicha palabra, conceptualmente hay una pequeña aclaración:

"Slicing no se trata de dividir, sino más bien de encontrar los incrementos que vamos a ir añadiendo poco a poco para hacer crecer una solución básica. No es el concepto de dividir que solemos tener, el de hacer de algo grande en trozos más pequeños. Se parece, pero no es lo mismo"
Por este motivo, conceptualmente, vamos a entender a la división como una ***descomposición****.*



Descomposición por capas (Horizontal Slicing)

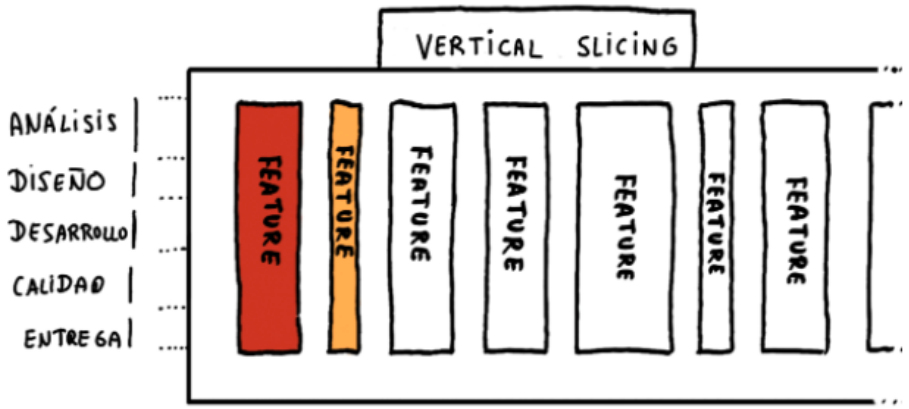
Cuando se plantea un cambio de paradigma de Cascada a Agile, muchos equipos siguen descomponiendo el trabajo de manera horizontal (Horizontal Slicing), planteando las tareas según las etapas de ciclo de vida del software.



Esta visión horizontal nos lleva al mismo punto que el de Cascada, ya que es fácil caer en la tentación de agrupar todas las tareas de análisis en un “sprint de análisis” y tener “Cascada en Sprints”. A pesar de aplicar agilismo (scrum por ejemplo), esta manera de organizar el trabajo nos sigue impidiendo focalizar en el negocio al no obtener un feedback temprano (tenemos que esperar mucho a tener algo terminado a mostrar).

Descomposición incremental (Vertical Slicing)

Uno de los objetivos del agilismo es mejorar la capacidad de adaptación de un equipo por lo que, para adaptarnos, tenemos que inspeccionar el resultado que nuestro producto produce a través del feedback temprano y, para eso, necesitamos tener incrementos de funcionalidades terminadas en cada iteración. Para conseguir esto, utilizamos la división de funcionalidades de manera vertical. Con la visión vertical, tratamos de hacer funcionalidades pequeñas que aporten valor y que podamos entregar a nuestros clientes cuanto antes.



Con la visión vertical el equipo se centra en el impacto del producto y no en el "x" por ciento de avance del proyecto para llegar en fecha.

Patrones de slicing de historias de usuario

Ahora bien, pero ¿cómo logramos esto?, es un tanto complejo dado que no estamos acostumbrado/as, culturalmente hablando, a encarar las tareas descompiéndolas en partes pequeñas. Para esto es que contamos con algunos **patrones** de slicing:

- **Pasos de un workflow:** identificar los pasos del workflow de la funcionalidad y crear una US por cada uno de ellos. **Ejemplo:**
  Épica: Publicar un artículo / Como usuario necesito publicar un artículo para divulgar mi trabajo
  Nota: por más que se escriba con formato US no deja de ser una funcionalidad muy grande (épica disfrazada de historia de usuario).
  Users Stories:
  1. **Como** autor **necesito** editar un artículo **para** obtener feedback de parte de mi editor
  2. **Como** editor **necesito** recibir una notificación cuando un artículo ha sido publicado **para** brindar feedback sobre el mismo
  3. **Como** editor **necesito** mejorar el artículo **para** publicar una versión fiel a las intenciones del autor
  4. **Como** autor **necesito** recibir el feedback de parte del editor **para** pulir mi artículo
  5. **Como** autor **necesito** enviar la versión mejorada del artículo **para** divulgar mi trabajo

  Nota: este conjunto de US implican la publicación de un artículo en base al workflow de la funcionalidad

- **Camino Feliz/Alternativos:** crear US para cumplir con el camino feliz y otras que agreguen validaciones y soporte a las excepciones. **Ejemplo:**
  Épica: autenticarse en una aplicación

  Users Stories:

1. **Como** usuario **necesito** autenticarme en la aplicación **para** acceder a los beneficios de la misma
2. **Como** usuario **necesito** recuperar password **para** tener las credenciales completas
3. **Como** usuario **necesito** confirmar la password nueva **para** autenticarme a la aplicación

➢ **Reglas de negocio**: si la funcionalidad describe una serie de reglas de negocio seguramente podemos descomponerla en historias que cumplan cada una de esas reglas con un nivel de especificidad mayor teniendo en cuenta cada caso particular. **Ejemplo:**
Épica: emitir el DNI

Users Stories:

1. **Como** ciudadana/o **necesito** emitir un DNI nativo **para** transitar por el país legalmente
2. **Como** ciudadana/o **necesito** emitir un DNI de naturalización **para** cambiar mi nacionalidad
3. **Como** ciudadana/o **necesito** emitir un DNI de extranjero **para** legalizar mi situación en el país

Notar que: el "para" no es el mismo en todos los casos, dado que cada persona cuenta con una situación en particular.

➢ **Mayor esfuerzo**: muchas veces podemos descomponer una funcionalidad, en la que un esfuerzo dedicado a una de las historias impactará en todo el resto, asumiendo una complejidad más alta en la primera de ellas.
Un ejemplo típico es la inclusión de pagos con tarjetas de crédito, donde primero se debe preparar el software para poder gestionar el pago con una tarjeta (la más utilizada por ejemplo), y posteriormente añadir nuevas tarjetas que requieran menor esfuerzo. **Ejemplo:**
Épica: Pagar con tarjeta de crédito
Users Stories:

1. **Como** cliente **necesito** pagar con tarjeta visa **para** completar la compra
2. **Como** cliente **necesito** pagar con tarjeta Mastercard **para** completar la compra
3. **Como** cliente **necesito** pagar con tarjeta American Express **para** completar la compra

Notar que: en este caso el "para" es el mismo en todos los casos porque la funcionalidad es la misma, sólo cambia el tipo de tarjeta.

➢ **De simple a complejo**: hay historias que ocultan cierta complejidad en su funcionalidad o que cuentan con muchas acciones pequeñas camufladas mediante conjunciones. En este último caso los indicadores más comunes se

manifiestan a través de la cantidad de Criterios de Aceptación, mientras que en el primer caso son más difíciles de detectar. Una buena manera es preguntarse ***¿se puede resolver de una manera más simple?***
Si la respuesta es sí, entonces se sugiere crear una US con la versión más simple y luego iterar con historia que complejizan el proceso. **Ejemplos:**
Ej 1. Épica (camuflada en US compleja): Como solicitante de un préstamo **necesito** calcular los pagos de mi hipoteca **para** determinar qué tipo de préstamo me conviene.
Nota: es una funcionalidad difícil de desarrollar sin experiencia previa en el tema, y además puede resultar de un desarrollo muy grande como inicio.

Users Stories:

1. **Como** solicitante de un préstamo **necesito** calcular los pagos de mi hipoteca manualmente **para** determinar qué tipo de préstamo me conviene
2. **Como** solicitante de un préstamo **necesito** calcular los pagos de mi hipoteca mediante una planilla de excel **para** determinar qué tipo de préstamo me conviene
3. **Como** solicitante de un préstamo **necesito** calcular los pagos de mi hipoteca mediante una calculadora online **para** determinar qué tipo de préstamo me conviene

Notar que: en este caso el "para" es el mismo en todos los casos porque la funcionalidad es la misma, sólo cambia la complejidad del proceso.

Ej2. Épica (camuflada en US con conjunciones):
Como usuario logueado necesito conocer los vuelos disponibles entre un origen y un destino, pudiendo indicar además el número de escalas y buscar vuelos por un rango de fechas para reservar un vuelo
Nota: es una funcionalidad con muchas "cosas" por desarrollar (las conjunciones).
Users Stories:

1. **Como** usuario logueado **necesito** conocer los vuelos disponibles entre un origen y un destino **para** saber si puedo reservar el vuelo
2. **Como** usuario logueado **necesito** indicar el número de escalas **para** reservar un vuelo que se adapte a mis preferencias
3. **Como** usuario logueado **necesito** buscar vuelos por un rango de fechas **para** ver si hay vuelos disponibles para cuando quiero viajar

➢ **Por tipo de datos**: cuando los datos de la funcionalidad tienen un significado llamativo, será necesario analizar e identificar el dato o tipo de dato con el mayor valor comercial para descomponer la historia en función de ello.

Ejemplos:

Ej 1 – Épica: consolidar reporte online de ciudades

Users Stories (slicing por dato):

1. **Como** gerente de ventas **necesito** obtener el reporte de ventas online de la ciudad con mayores ventas locales **para** conocer la rentabilidad de dicha ciudad

Notar que: en este caso se trabaja en primera instancia sobre una sóla ciudad (un dato particular), aquella que tiene mayor cantidad de ventas, dejando que el resto, por ahora, se siga realizando de manera manual / offline.

Ej 2 – Épica: comprar productos (en una librería que tiene varios tipo de productos)

Users Stories (slicing por tipo de dato):

1. **Como** adolescente **necesito** comprar un libro del nivel secundario **para** realizar los trabajos de la escuela
2. **Como** adulto **necesito** comprar un libro literario **para** leer un nuevo libro
3. **Como** niño/a **necesito** comprar un libro infantil **para** aprender a leer

Notar que: el/la PO analizó los tipo de productos que más se necesitan en base al público del negocio. En este caso, la librería recibe más clientes del tipo adolescente, por lo tanto se focaliza en los productos que éstos consumen, luego se irán agregando el resto aplicando el mismo criterio.

➢ **Por operaciones (ABMC)**: cuando la funcionalidad involucra las operaciones de alta, baja, modificación y consulta, será necesario descomponer dichas operaciones una en cada US, comenzando con lo estrictamente necesario, dejando para más adelante las operaciones menos utilizadas, las cuales quedarán temporariamente soportadas por procesos manuales.

Ejemplo:

Épica / US camuflada: Administrar tienda online / **Como** encargada **necesito** administrar los productos que se venden en mi tienda online **para** vender lo que la gente quiere comprar.

Users Stories:

1. **Como** encargada **necesito** agregar productos a mi tienda online **para** vender lo que la gente quiere comprar.
2. **Como** encargada **necesito** ver los productos de mi tienda online **para** verificar el stock de cada producto
3. **Como** encargada **necesito** actualizar la información de los productos de mi tienda online **para** que el producto refleje la información precisa actual
4. **Como** encargada **necesito** eliminar productos de mi tienda online **para** que la gente no acceda a productos que ya no están disponibles

Nota: prestar atención a los verbos genéricos como "gestionar" o "administrar", por ejemplo, "como estudiante necesito administrar mi cuenta". La palabra "administrar"

tiende a ocultar acciones más específicas, como "cancelar una cuenta", editar una cuenta, etc.

- **Foco en el aprendizaje (spike)**: a menudo las funcionalidades no son necesariamente demasiado complejas sino que tienen muchas incógnitas del plano técnico (como elegir qué biblioteca de software usar). En dicho caso, será necesario que el equipo realice una investigación sobre dicha tecnología mediante un spike a modo de fase exploratoria. Ahora bien, este recurso será utilizado sólo en caso de real desconocimiento.
  Como parte de la investigación será necesario establecer un conjunto de preguntas cuyas respuestas serán resultado de la investigación. Las cuales deben ser detalladas en la descripción del spike, así como también cuándo se considera que el spike se cumplió (definición de done). Es importante respetar estas consignas, dado que es muy común caer en la tentación de investigar "eternamente", por lo cual hay que saber cuándo frenar y dejar el resultado de la investigación bien especificado en algún documento.
  Nota: el uso del spike debemos considerarlo como último recurso. Probablemente el equipo sepa lo suficiente como para construir algo inicial, y luego cuando ya haya agotado toda posibilidad, si quedan cuestiones por aprender se aplica el spike. Por lo tanto, debemos hacer todo lo posible para utilizar uno de los patrones anteriores antes de recurrir a este patrón.

- **Con servicios externos (mockear)**: cuando se consumen servicios externos y aún no contamos con ellos, una buena práctica para avanzar en nuestro producto es simular la dependencia (mocking) para aislarla simulando así su comportamiento o comenzar con una utilización básica del servicio externo que luego se seguirá mejorando de manera incremental.

- **Método de la hamburguesa**: ver la explicación en el siguiente sitio
  Bonus: les comparto estos sitios para profundizar un poco más sobre el relevamiento / análisis de requerimientos y la manera de organizarlos de manera ágil.
  Explicación conceptual - Ejemplo

Cada uno de estos patrones se pueden combinar según se necesite, es decir que cada épica (o US camuflada) se puede refinar tantas veces como sea necesario hasta lograr contar con un backlog de historias INVEST.
Bonus de US: para quienes quieran profundizar en el tema les comparto este sitio con el resumen de las distintas características que puede contener una US.

D.O.R y D.O.D: relación con el Sprint Backlog

Siendo que el sprint Backlog se crea a partir del Product Backlog, será necesario definir los criterios que le van a permitir al equipo de desarrollo poder trabajar con las historias de usuario dentro de un sprint.

Para esto se cuentan con 2 conceptos:

D.O.R (Definition Of Ready – Definición de listo)

Son los criterios mínimos (conjunto de características) a partir de los cuales el equipo define que una historia de usuario está lista para desarrollarse.

Ejemplos:

- La US debe tener los criterios de aceptación
- La US debe ser INVEST
- La US debe tener los mockups de las pantallas (en caso de necesitar)

US que no cumple con el D.O.R no se puede desarrollar, y por tanto no puede entrar en un sprint.

D.O.D (Definition Of done – Definición de hecho / terminado) (Compromiso)

Son los criterios mínimos (conjunto de características) a partir de los cuales el equipo define que la incidencia está terminada para ser parte del incremento. Estos criterios suelen estar asociados al aporte de calidad del producto.

Ejemplos:

- La US cumple con todos los criterios de aceptación
- Las US pasaron todos los tests automáticos
- Les testers dieron el visto bueno de calidad
- La US fue revisada por 2 integrantes del equipo (code review)
- El código fuente está en el branch “xx” y el build se ejecutó correctamente

US que no cumple con el D.O.D no se entrega como parte del incremento a revisar con el cliente.

Estos criterios se definen en la Sprint Planning. Es muy probable que el equipo los defina en el primer sprint, y en cada planning chequee si amerita contar con una nueva definición para el sprint en cuestión, en caso contrario se tienen en cuenta los criterios ya definidos. Ambas definiciones deben estar consensuadas y visibles para todo el equipo Scrum.

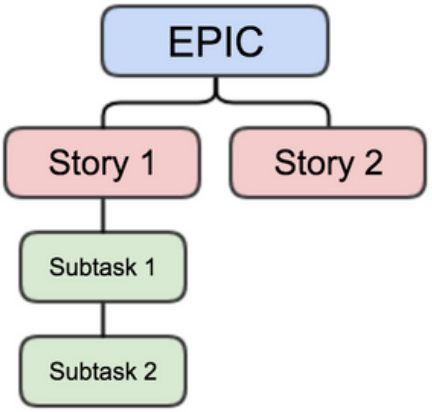
Sub-tasks (Sub-tareas)

Como venimos viendo las historias de usuario representan funcionalidades, pero también es necesario registrar las tareas técnicas que implica cada una. Para ello es que se crean **sub-tareas** relacionadas a cada US. Estas incidencias son creadas netamente por el equipo de desarrollo en el refinamiento, o como última instancia, en la planning. Se deberá crear una sub-task por cada tarea de programación y de testing que implique el desarrollo completo de la historia de usuario.

Las tareas deben ser pequeñas, es decir que no deberían llevar más de una jornada laboral y cuya sumatoria completa la US.

Los campos de una sub-task son: ID, TÍTULO y DESCRIPCIÓN. Esta última se deberá **redactar en lenguaje netamente técnico**. En este nivel de documentación no interviene el/la PO, quien sólo se encarga de las tareas funcionales del producto. Las sub-task no se suelen estimar, aunque de ser necesario se estiman en horas, dado que deberían ser lo suficientemente chicas como para desarrollarlas en un día.

La jerarquía completa quedaría:



Veamos un ejemplo completo:

ID: US-12
Título: Pagar con tarjeta de crédito visa
Descripción:
Como cliente **necesito** pagar con tarjeta visa **para** completar la compra

Criterios de Aceptación:

1. Se debe poder ingresar el DNI del cliente
2. Se debe poder seleccionar el tipo de tarjeta "Visa"
3. Se debe poder ingresar los siguientes datos de la tarjeta de crédito:
   a. Nro de tarjeta
   b. Fecha de vencimiento
   c. Código de seguridad
4. Se muestra un botón "Pagar". Al clickear sobre el botón se debe:
   a. Validar que los datos de la tarjeta se correspondan con el cliente
   b. En caso afirmativo se deberá registrar el pago asociado al cliente y mostrar el siguiente mensaje: "EL PAGO SE CONCRETÓ SATISFACTORIAMENTE".
      En caso de ocurrir un error se deberá mostrar: "EL PAGO NO SE PUDO CONCRETAR"
      En caso negativo se deberá informar "LOS DATOS DE LA TARJETA NO SON VÁLIDOS PARA DICHO CLIENTE"

Puntos de historia: **8**

Tasks (tarea técnica)

La tarea técnica y la sub-task son muy similares, ambas representan trabajo técnico del equipo de desarrollo, la diferencia radica en que se utiliza la **sub-task para tareas técnicas** exclusivas de **desarrollo de una funcionalidad** (por eso están relacionadas a una US), mientras que la **Task es una tarea independiente de las funcionalidades** del producto. Puede ser una tarea de configuración, de desarrollo general, de base de datos, de servicios, etc.

Bugs / Defecto (Error)

Para este tipo de incidencia debemos recordar el ciclo de vida del software.
Sabemos que durante la etapa de desarrollo, los/las testers diseñan los casos de prueba, para que en la etapa de prueba finalmente se ejecuten.
Como resultado de dicha ejecución se pueden dar 2 opciones:

1. La prueba pasó (Success)
2. La prueba falló (Fail)

Para el caso desfavorable necesitamos gestionar el reporte del error. Para ello es que se utiliza el tipo de incidencia Bug, también conocido como defecto o simplemente “error”.
La manera de reportar un bug se verá en la siguiente unidad sobre **Calidad**.

Theme (tema)

El theme o tema **no es un tipo de incidencia**, sino una manera de **agrupar** las incidencias. Para proyectos grandes es necesario ordenar, bajo un título o etiqueta, aquellas incidencias que están relacionadas a un mismo tema. Dependiendo la herramienta, el theme será un label de color con un título que permite identificar la temática de todas las funcionalidades que agrupa. Es muy común que este concepto se confunda con una épica. A tener en cuenta que el theme sólo agrupa funcionalidades, mientras que la épica es una funcionalidad (grande).

Spikes

Esta incidencia se define cuando se necesita realizar una investigación técnica. La estructura consta de un ID, un título y una descripción. La cual debe expresar la tarea de investigación junto con la definición de done correspondiente (output esperado como finalización de la investigación). Como por ejemplo, un documento con un resumen de lo investigado o una P.O.C (prueba de concepto).

UX y Mockups gráficos

Recordemos que como parte de una historia de usuario contamos con la instancia de confirmación, en la cual se detallan los criterios de aceptación; y tal como hemos mencionado en la misma se detallan los criterios que conforman la funcionalidad. Los cuales se pueden detallar por escrito, en formato de puntos / viñetas, o formato gráfico. Para ello, es que se realizan los mock-ups.

Estos documentos aportan información visual sobre la pantalla y la manera de visualizar la funcionalidad. Si bien los mockups suelen ser creados por la gente de diseño, en caso de no contar con dicho rol, el/la PO puede crearlos aunque sea de manera manual. La visualización gráfica de la pantalla esclarece mucho el entendimiento de la necesidad.

Diferencia entre requisito, CU y US

Los tres intentan representar las características que tiene que tener un sistema, la diferencia está en el enfoque dado.

- Los requisitos del sistema (en la especificación funcional) están escritos desde la perspectiva del sistema y no en la interacción del usuario, representan las funcionalidades generales en estado puro.

- Los Casos de Uso (CU) están escritos como una serie de interacciones entre el usuario y el sistema. Hacen hincapié en el contexto orientado al usuario, es decir a las funcionalidades que utiliza cada usuario identificado en el sistema. Son la forma de capturar los requisitos del sistema desde el punto de vista del usuario. Se incluyen en la especificación funcional en el desarrollo con cascada.
- Las Historias de Usuario (US) sirven para describir lo que el usuario desea ser capaz de hacer. Además, las historias de los usuarios se centran en el valor que genera utilizar el producto en lugar de una especificación detallada de lo que el mismo debe hacer. Están concebidos como un medio para fomentar la colaboración. Se incluyen en el product backlog de un desarrollo con Scrum.

Bonus: les comparto la fuente de estas diferencias para ahondar aún más:
http://www.angellozano.com/requisitos-del-sistema-vs-casos-uso-vs-historias-usuario/

UNIDAD 5: CALIDAD

BREVE INTRODUCCIÓN

Como se podrá observar, el concepto de calidad lo vamos a encontrar en distintos aspectos del desarrollo de software. Por un lado, el ciclo de vida del software cuenta con una etapa exclusiva para "probar" el producto. A su vez, uno de los principios más importantes de la metodología ágil se basa en la entrega de software con calidad.

¿Pero es posible asegurar un producto 100% efectivo y correcto?

La respuesta es no… pero sí podemos asegurar un producto con "la mayor calidad posible" o mejor dicho, con una calidad "aceptable". Es verdad que esta expresión es bastante ambigua, motivo por el cual, en la presente unidad conoceremos algunas técnicas y prácticas que nos ayudarán a acercarnos un poco más a este objetivo tan ambicioso.

Antes de avanzar, me gustaría aclarar "una pequeña-gran" diferencia:

Se considera que un producto de software con calidad debe SER CORRECTO Y FUNCIONAR CORRECTAMENTE (CORRECTITUD). Parece que estamos hablando de lo mismo ¿verdad? Pues hay una sutil diferencia, bastante implícita, en la semántica de cada expresión.

¿Cómo sabemos que el software es correcto? (corrección)

Básicamente cuando satisface al cliente, es decir cuando cumple con sus expectativas, y por lo cual con los requerimientos documentados (ya se en una especificación funcional o mediante un product backlog).

Por el contrario, no será correcto si escuchamos de parte del cliente la famosa frase *"No es lo que pedí".* En este caso, es muy probable que los requerimientos no hayan sido documentados según las necesidades del cliente. Esto va a depender mucho de la metodología de desarrollo empleada.

Entonces ¿cuándo un producto funciona correctamente? (correctitud)

Ni más ni menos que cuando el producto respeta y coincide con los requerimientos documentados con la mayor calidad posible, es decir, "sin errores". Pero como sabemos que esto no es factible, el nivel de calidad deberá ser acordado entre todas las partes.

Esto es, tanto el cliente como el equipo deberán negociar y definir la tasa o grado de calidad que se considera aceptable antes de la entrega del producto.

Por el contrario, cuando no se llega a cierto grado de calidad, encontrando muchos defectos en el producto, se dice que éste no funciona correctamente.

Habiendo aclarado esto, será necesario que al momento de construir un producto de software debemos asegurar la **corrección** y **correctitud** del producto a entregar. Debemos respetar ambas características. Dicho en otras palabras, desarrollar el producto que el cliente realmente solicitó y que el mismo cumpla con la tasa de calidad acordada.

Antes de continuar me gustaría continuar con más aclaración. Es muy común utilizar como sinónimos los términos, **error, falla y defecto (bug).** Pero, nuevamente hay una sutil diferencia entre cada uno, la misma no radica en el resultado final, sino en la fuente de donde se origina.
Veamos:

- El **error** es **humano**. Es decir que las personas somos quienes cometemos errores, no los sistemas
- Lo que **falla** son los **sistemas** a causa de los errores humanos
- Los **defectos** son la manera que tenemos los humanos de reportar el error causado para corregir el sistema.

  Como una manera de gestionar los defectos, decimos que reportamos un error (bug) mediante la creación de una incidencia de tipo defecto, de ahí que llamamos a esta tarea, realizar el “reporte de bugs”.

Tipo de pruebas

Hasta ahora venimos realizando mucho hincapié en el software desde una visión funcional, por lo cual, claramente vamos a necesitar realizar pruebas con dicho enfoque. Pero… no serán las únicas pruebas que se necesitan tener en cuenta para que todo el proyecto (junto con su producto), sea exitoso.
Tal como hemos mencionado en la unidad 1, **un producto, un proyecto y un sistema son conceptos diferentes**. Hasta ahora cuando hablábamos de calidad, el foco estaba puesto enteramente en el producto, pero como **un producto es sólo una parte de un proyecto dentro de un sistema**, para que todo este sistema funcione, debemos contemplar y proveer distintos enfoques de calidad.
De aquí es que existen estos 2 tipo de pruebas:

- Pruebas Funcionales
  - Manuales
  - Automatizadas
- Pruebas no funcionales

Pruebas funcionales

Tal como su nombre lo indica, son aquellas pruebas que se basan exclusivamente en validar cada funcionalidad del producto según sus requerimientos funcionales.
Como parte de este tipo de pruebas contamos con:

- Pruebas unitarias
  - Caja Negra
  - Caja Blanca
- Pruebas de integración
- Pruebas de humo (Smoke test)

- Pruebas de regresión
- Pruebas de aceptación de usuario (U.A.T)
- Pruebas exploratorias

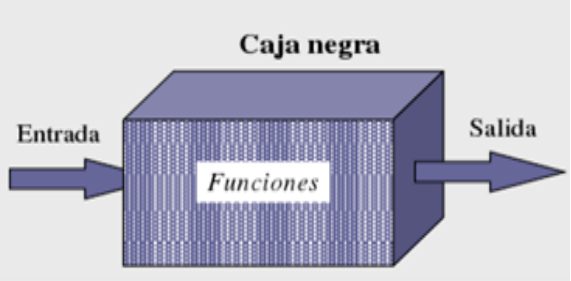
A su vez, cada una de estas pruebas puede clasificarse según el enfoque: Manual o automatizado. Por ahora veremos de qué trata cada una conceptualmente hablando y luego ahondaremos en la diferencia del testing manual o automatizado (automation tests).

Pruebas unitarias

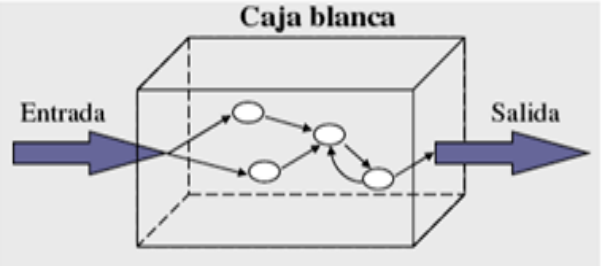
Son aquellas pruebas que validan la unidad mínima de funcionalidad, son las pruebas más atómicas del producto. En la metodología ágil, son aquellas que validan cada criterio de aceptación de las historias de usuario. Mientras que en cascada deberán validar cada funcionalidad según los casos de uso.

- de caja negra: son las pruebas que se basan exclusivamente en la interfaz y su respectiva funcionalidad. Nos debemos abstraer de la lógica interna del producto; sólo se tienen en cuenta los input (datos de entrada) y sus respectivos outputs (datos resultantes). Para ejecutar estas pruebas será necesario diseñar los casos de pruebas correspondientes.

  "El objetivo consiste en interactuar con la interfaz para probar su funcionalidad, asegurando que las entradas y salidas del sistema cumplen con los requerimientos"



- De caja blanca: por el contrario, estas son pruebas que sí validan la unidad funcional desde adentro, es decir, en base a la lógica de la funcionalidad. Nos interesa saber cómo se resuelve el problema, en base a las entradas para así validar su resultado de salida. Para ello, es que en este caso, las validaciones se deben programar, a diferencia de las pruebas de caja negra.
  Es decir, que quienes se enfocan en este tipo de pruebas son personas técnicas del equipo de desarrollo. Cada una de estas pruebas validan la unidad funcional del código que hace al producto.



Los tipos de pruebas son parte de la ingeniería del software y no de una metodología de trabajo. Pero la forma de implementarlas sí dependerá de la metodología utilizada.
En el caso de cascada se codifica la funcionalidad del producto, y luego en su etapa correspondiente se codifican las pruebas. Mientras que en la metodología ágil, al concretar todas las etapas en cada iteración, los test se van desarrollando casi junto con el código fuente del producto. Si bien es muy común que se codifique cada US y llegando al final del sprint se realicen los test unitarios, no es lo recomendable según la filosofía ágil; dado que de esta manera se está disfrazando cascada dentro del sprint. Lo más agilista es utilizar la técnica TDD, donde el equipo desarrolla directamente mediante los test.

Pruebas de integración

Estas pruebas apuntan a probar la unión de las partes. Es decir, hasta ahora probamos cada funcionalidad por separado, pero ¿qué pasa al integrarlas?
En un producto chico estas pruebas pasan casi desapercibidas, dado que se ejecutan muy rápidamente; pero para productos grandes realmente es necesario tomarse el tiempo para validar el producto como un todo.
Nuevamente, la forma de implementarlas dependerá de la metodología de trabajo. Si de agilismo hablamos, estas pruebas se van ejecutando con cada incremento, una vez que se hayan ejecutado las pruebas unitarias y antes de finalizar el sprint.
Estas pruebas suelen estar más orientadas a la caja negra, en la cual el equipo de testing se dedica a probar el producto como tal, focalizando en la interacción de sus partes; se validan las reglas de negocio del producto y la consistencia de los datos a lo largo del recorrido. Como prueba mínima se debe asegurar la correctitud y corrección del camino feliz.

Pruebas de humo (Smoke test)

Dijimos que en las pruebas de integración debemos validar el producto completo, nuevamente para productos chicos no hay inconvenientes, por lo cual la mayoría de estas pruebas prácticamente no son necesarias en dicho caso; por lo tanto de ahora en adelante nos vamos a enfocar sólo en los productos grandes.

Entonces, **¿cómo probamos todo el producto de manera integral? ¿por dónde comenzar? ¿cuánto tiempo tenemos?¿vamos a llegar?**

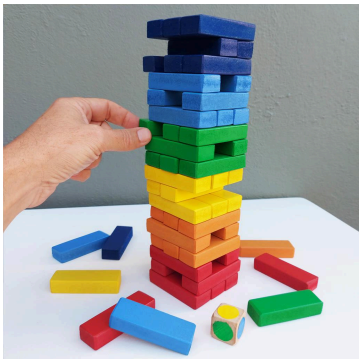
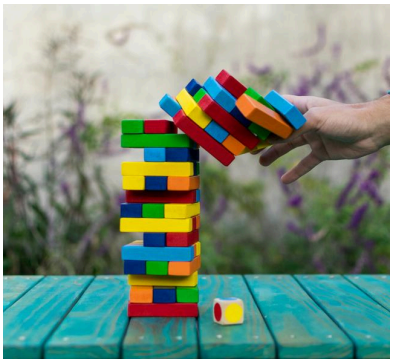
La variable fundamental que generó la necesidad de contar con este tipo de pruebas es, el tan preciado **"Tiempo"**. En muchos casos, lamentablemente, no se dispone del tiempo necesario para realizar las pruebas de integración en profundidad, por lo cual, una manera de asegurar cierto grado de calidad, es mediante las pruebas de humo o más conocidas como Smoke test.
Con estas pruebas no se busca validar todo el producto de forma exhaustiva, sino realizar una serie de validaciones a las **funcionalidades más críticas que por lo general son parte del camino feliz**. La idea es hacer un **repaso rápido** de lo más **crítico** que se va a entregar y asegurar que funcione correctamente. No deben encontrarse defectos críticos o bloqueantes, en cuyo casó se deberá frenar la entrega hasta su resolución. Por eso es importante contar con el tiempo necesario, para llegado el caso, estar a tiempo de corregir los errores antes de la entrega. De esta manera, este tipo de pruebas son las últimas que se realizan, son el último recurso para asegurarnos la calidad del producto antes de su entrega.

Nota: el nombre proviene como analogía de las pruebas rudimentarias en ingeniería electrónica, en las que se comprueba que el encendido de un circuito no causa humo ni chispas.

Pruebas de regresión

Como parte de las pruebas de integración notamos que es muy común que al agregar funcionalidad nueva o modificar la existente, se suelen cometer errores (humanos), por lo cual, la tarea del equipo de testing será **sostener y asegurar** la calidad del producto en cada incremento. Para ello es que se necesitan realizan (diseñan y ejecutan) las pruebas de regresión.

Cual jenga, básicamente queremos evitar el efecto "onda" o "dominó" ante los cambios en el producto:

Es por esto que las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el producto, tanto para corregir un error (correctitud) como para realizar una mejora (corrección). No alcanza con probar sólo las funcionalidades modificadas o agregadas, sino que también será necesario validar que estos cambios no produzcan efectos negativos como consecuencia sobre otras funcionalidades ya desarrolladas (y entregadas, en el caso del agilismo).
Normalmente, este tipo de pruebas implica la **repetición** de las pruebas que ya se

han realizado previamente, con el fin de asegurar que no se hayan introducido más, o nuevos errores que comprometan el funcionamiento general del producto.
Para esto, la manera de documentar las pruebas será **creando un plan de pruebas de regresión** a partir del diseño de los casos de prueba ya diseñados. Pero siendo que no podemos ejecutar todos los casos de prueba que disponemos, por tratarse de un universo tan amplio y sabiendo que el tiempo siempre apremia, será necesario identificar y elegir qué pruebas son las que deberán integrar el plan de regresión.
Las pruebas de regresión pueden incluir:

- La ejecución de las pruebas del plan de regresión
- La revisión de los procedimientos preparados antes del cambio, para asegurar que permanecen correctamente y descartar que la causa de los errores provengan del proceso en lugar de la vesión del código fuente
- Revisión de los datos (BBDD o repositorio de información correspondiente) para asegurar la integridad de la información

Pruebas de aceptación de usuario (U.A.T)

Las pruebas de aceptación de usuario o su homólogo en inglés User Acceptance Test, tal como su nombre lo indica, son pruebas que suelen realizar los usuarios o incluso el cliente, como última instancia antes de publicar la versión a producción. Son las pruebas más cercanas a la realidad posible, dado que no están sesgadas por el equipo de desarrollo. El cliente suele tener una mirada netamente funcional de su negocio, por lo cual validará las reglas esenciales del producto. Cuando el cliente confirma la aceptación del producto luego de estas pruebas es cuando finalmente se publica. Para esto, las pruebas se realizan en un ambiente específico llamado U.A.T (Pre-producción, beta, homologación, etc) lo más similar a producción posible, simulando el entorno productivo.

Pruebas de usabilidad (UX)

De estas pruebas se encarga generalmente el equipo de diseño, dado que se focalizan en probar la experiencia de usuario (**U**ser e**X**perience). Es decir, buscan indagar cuán fácil e intuitivo es utilizar el producto para los/as usuarios/as.

Las características evaluadas en la usabilidad incluyen:

- **Sesiones:** qué tan fácil es para los/las usuarios/as realizar funciones básicas la primera vez que utilizan el producto
- **Eficiencia:** que tan rápido los/as usuarios/as experimentados/as pueden realizar sus tareas
- **Memorización:** que tan fácil de memorizar es el uso del producto, esto es, cuando un usuario pasa mucho tiempo sin usarlo ¿puede recordar lo suficiente para usarlo con efectividad la próxima vez, o tiene que empezar a aprender de nuevo?

- **Errores:** cuántos errores atribuibles al diseño comete el/la usuario/a, qué tan severos son y qué tan fácil es recuperarse de los mismos.
- **Satisfacción:** qué tanto le gusta (o desagrada) al/la usuario/a utilizar el producto.

Pruebas exploratorias

Estas pruebas se suelen realizar para complementar nuestro set de pruebas ya diseñadas (caja negra). Es decir, cuando se desea ampliar la cobertura de calidad, es factible buscar bugs más complejos o nuevos flujos con este enfoque.
En este enfoque, a diferencia del resto, no se separan las etapas de diseño de casos de casos de prueba y de ejecución, sino que se realizan ambas tareas en simultáneo, una alimenta a la otra.
Este enfoque de testing es muy aprovechado en la metodología ágil, dado que permite explorar (de ahí el nombre) y aprender con cada iteración. Pero nuevamente, como suele suceder, hay un mito a derribar; y es la creencia que en este tipo de pruebas es muy informal y no se documenta, pues no es así.
Cuando se hace testing exploratorio, se diseña y ejecutan pruebas con un objetivo en concreto. Luego con las conclusiones obtenidas, se va aprendiendo sobre el producto, y se utiliza dicha información para iterar, diseñando y ejecutando nuevas pruebas.
Cuando hacemos testing exploratorio no probamos por probar, sino que antes de empezar tenemos que definir varias cosas:

1. **Sesiones:** se establecen sesiones de trabajo en un tiempo limitado, en las cuales se define un **objetivo general** sin una guía como sucede en caja negra. Por ejemplo establecer flujos que podrían seguir los usuarios y probarlos, ver cómo se integra la aplicación con un servicio externo, vulnerabilidades de seguridad en el login etc. Quien testea es responsable del camino que seguirá para conseguir ese objetivo, el cual puede ir cambiando a medida que se va aprendiendo sobre el producto durante el tiempo establecido. Los resultados obtenidos permitirán además definir las siguientes sesiones a realizar, es decir, una mejora continua.

2. **Tiempo:** será necesario establecer el tiempo que durará la sesión para “no irnos por las ramas”.

3. **Documentación:** a medida que se va probando se va tomando nota y documentando las pruebas (símil a diseñar los casos de prueba).
   No hay ninguna restricción sobre cómo documentar en testing exploratorio. La misma puede ser en papel, un documento, post-its, alguna herramienta de tracking de incidencias, etc. También es muy útil sumar una mirada gráfica como ser screenshots, mapas conceptuales o mockups.

Lo que sí es imprescindible (como en cualquier actividad de testing) es ser capaces de reproducir los bugs que se encuentran.

4. **Reporte de bugs:** al finalizar con las sesiones, se reportarán los bugs encontrados.

Este tipo de pruebas permiten llegar a donde la automatización no puede llegar: pensar como un ser humano (los/las usuarios/as)
Particularmente para este tipo de testing se necesitan las habilidades blandas más características de un tester:

El testing exploratorio bien planteado puede dar muy buenos resultados, pero todo depende de las habilidades del tester, de que sea capaz de analizar en qué puntos podría fallar el producto (de acuerdo al objetivo establecido), qué riesgos puede tener eso etc.; todo ello basándose en el conocimiento e intuición que tiene del producto. Al final, motivamos a los/las testers a que utilicen su intuición, ideas nuevas, su habilidad crítica para detectar bugs más complejos. Y así entrenar y mejorar como testers.

Pruebas manuales vs automatizadas

Hasta ahora venimos mencionando distintos tipos de pruebas funcionales, cada una con su objetivo, pero no hemos mencionado la manera de llevarlas a cabo. Las pruebas de caja negra (y las que derivan de ella) se pueden realizar de 2 maneras: **manual o automatizada**.

Las pruebas manuales, básicamente requieren de la interacción del/la tester con el producto, si bien se cuenta con herramientas que permitan agilizar dicha interacción, como ser Selenium, la mirada de un ser humano para pensar como otro ser humano es sumamente útil. Pero esta mirada es muy beneficiosa cuando necesitamos encontrar bugs muy ocultos o flujos complejos, pero en cambio para las pruebas de rutina, como las del camino feliz, y las funcionalidades mínimas y críticas, suele ser una tarea bastante engorrosa y aburrida de llevar; por lo tanto, para este tipo de pruebas, se suele recurrir a la automatización de las pruebas ya diseñadas. Ésto implica codificar las pruebas mediante un lenguaje de programación provisto para ello. Como parte del set de pruebas a automatizar es muy común, y pertinente, incluir las pruebas de regresión, las cuales incluyen las de smoke test.

Pruebas no funcionales

En este caso el foco estará dado sobre el proyecto y/o sistema en general más allá del producto. El producto ya se testeó durante las pruebas funcionales, pero como ya sabemos un sistema es más que un producto y la calidad deberá ser transversal al sistema. De esta manera es que contamos con una serie de pruebas que aseguran la calidad del sistema según los requerimientos no funcionales establecidos. Estos se suelen documentar dentro de la especificación funcional (en cascada) o como parte de la Inception en el caso del agilismo.

Como parte de este tipo de pruebas contamos con las siguientes pruebas más relevantes:

- Pruebas de rendimiento / performance
  - Pruebas de volumen
  - Pruebas de carga
  - Pruebas de concurrencia
  - Pruebas de stress
- Pruebas de seguridad

Pruebas de rendimiento o performance

Estas pruebas abarcan varios tipos de pruebas enfocadas en el rendimiento y la capacidad de respuesta de un sistema o componente bajo diferentes volúmenes de carga. Dicho de otra forma, las pruebas de rendimiento determinan cómo se comporta un sistema en términos de respuesta y estabilidad sobre una carga de trabajo concreta.

1. Pruebas de volumen

Son pruebas que se realizan cuando el producto cuenta o se espera que cuente con una amplia cantidad de **datos**. Es decir, se prueba que la aplicación “se banque” procesar un número elevado de información. Lo que se busca con estas pruebas, es encontrar el límite que pueda causar el fallo en el sistema, tanto por la capacidad de soportar un volúmen alto de información, como el tiempo de respuesta al ser consultada.

2. Pruebas de carga

Este tipo de prueba tiene como objetivo medir la capacidad de carga del sistema ante cierta **cantidad de usuarios**. Es decir, calcular la respuesta de la aplicación con diferentes medidas de usuario o peticiones. Ejemplo: conocer cuál es la respuesta al procesar el ingreso de 10, 100 y 1000 usuarios de forma parametrizada. Este resultado se compara con el resultado esperado.

3. Pruebas de concurrencia

Este tipo de prueba se basa en medir la capacidad de respuesta que tiene el sistema ante ciertas peticiones pero de manera **simultánea**.

Por ejemplo, un elevado número de software de terceros realizando la misma llamada sobre una API, o muchos usuarios enviando el mismo formulario de contacto. Un claro ejemplo de la importancia de este tipo de pruebas se da en muchos casos para ocasiones particulares, como ser el “Black Friday” para productos del estilo e-commerce.
El comportamiento del sistema, incluido el tiempo de respuesta del software que perciben los usuarios, dependerá de la concurrencia que exista en un momento determinado sobre el mismo. Pero para esto debemos identificar cuáles son los niveles de concurrencia que encontramos en las pruebas de rendimiento:

- **Concurrencia a nivel de software**: nos indica el número de usuarios que se encuentran en la aplicación en un instante dado.
- **Concurrencia a nivel de transacción**: indica el número de transacciones que se realizan de manera simultánea

4. Pruebas de stress

Tal como su nombre lo indica, estas pruebas buscan estresar al sistema, exigiéndolo a su capacidad máxima. Si bien son muy similares a las anteriores, la diferencia radica en que debemos superar los límites esperados en el ambiente de producción o los límites que fueron determinados en las pruebas.
Básicamente esta prueba se utiliza para romper la aplicación. Se va doblando la cantidad de usuarios que se agregan al sistema y se ejecuta una prueba de carga hasta que se rompe. Este tipo de prueba se realiza para determinar la solidez del sistema en los momentos de carga extrema y ayuda a determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.
Se puede utilizar para medir la capacidad de dicho sistema o componente en caso que no disponga de suficientes recursos (como por ejemplo ancho de banda, procesador, memoria, etc…).

Pruebas de seguridad

Estas pruebas están relacionadas con el hacking ético, se utilizan para detectar vulnerabilidades y auditoría de buenas prácticas, comprueban los atributos o características de seguridad del sistema, si puede ser vulnerado, si existe control de acceso por medio de cuentas de usuario y si pueden ser vulnerados estos accesos. Entre las características de seguridad de un sistema, están la confidencialidad, integridad, autenticación, autorización y la disponibilidad. Es vital asegurar la seguridad, no sólo de la aplicación en cuanto a su operatoria funcional, sino a la privacidad de la información. Es por eso que en productos muy grandes es una tarea muy compleja testear el comportamiento del software teniendo en cuenta la consistencia de los datos pero sin transgredir estas medidas de seguridad.

Pruebas de caja negra

Adicionalmente a lo ya mencionado, vamos a profundizar puntualmente sobre este tipo de prueba. Antes de continuar, cabe aclarar que este tipo de prueba es la más utilizada y se realiza con igual importancia tanto en **metodología tradicional como en el agilismo.**

Como parte de los posibles casos de prueba, los mismos se van a clasificar en:

- Casos Positivos: aquellos que cumplen con los requerimientos (especificación / US)
- Casos Negativos: aquellos que no cumplen con lo solicitado. Son los casos que, como testers, queremos ejecutar para "romper" la aplicación.

Una aclaración importante

Ambos son casos de prueba, es decir que al ejecutarlos, ambas pruebas pueden salir exitosas o fallar. Donde para este último caso se deberá reportar el bug correspondiente. **Pero no confundir un caso de prueba negativo con un bug.**

Ahora veamos cómo se diseña un caso de prueba.

Diseño de casos de prueba

Un caso de prueba refleja básicamente la prueba que se desea ejecutar, y como tal debemos documentarlo. La documentación estándar y básica de un caso de prueba contempla la siguiente información y estructura:

- ID
- Título de la prueba
- Pasos + datos
- Resultado esperado
- Precondiciones (opcional)
- Bug-ID (opcional)

Consideraciones importantes:

- El título debe describir la funcionalidad a probar, por lo que debe comenzar con un verbo en infinitivo.
- En los pasos se deberá detallar cada acción a realizar junto con sus respectivos datos, para ejecutar la prueba.
- El resultado esperado es el campo quizá más complicado de comprender aunque a simple vista no pareciera. En este campo se deberá detallar lo que **realmente** debemos ver en la pantalla. Es decir, la descripción debe ser exacta, concreta y específica. **No es la explicación** de lo que se espera, sino lo que se espera ver explícitamente. Hay una sutil diferencia.

Por ejemplo para la prueba del alta de un producto:

- **Incorrecto**: *"El producto se deberá guardar en la BBDD"* ⊗
- **Correcto**: "Se muestra el mensaje *"El producto se registró correctamente" y se visualiza el nuevo producto en el listado de productos"* ✓

En el primer ejemplo estamos explicando lo que debe hacer internamente la aplicación, lo cual dijimos que no es correcto; mientras que en el segundo, estamos detallando el resultado que esperamos ver en pantalla, en el producto (no en la base de datos, recordemos que es un test de caja negra). Aunque en muchos casos, dependiendo de los conocimientos del tester, se puede hacer "caja gris", indicando información relevante de la base de datos.

- Finalmente como parte de las PRECONDICIONES sólo debemos indicar, en caso de haber, las necesidades (condiciones) que necesitamos para poder ejecutar la prueba con éxito. Ejemplo: contar con las credenciales del admin.

- EL BUG_ID se completará con el Id del bug reportado si en la etapa de ejecución, la prueba falló.

Como ya vimos las pruebas cuentan con 2 posibles estados: éxito o falla, pero **¿cómo declaramos que una prueba falló?**
Pues bien, para ello es que utilizaremos el campo **"Resultado Esperado"** del caso de prueba. La idea es comparar dicho campo con lo que vemos en la pantalla. Si ambos coinciden, la prueba fue exitosa; en caso contrario, si vemos una diferencia (por más mínima que sea) quiere decir que la prueba falló. Por este motivo es tan importante ser muy específicos/as al redactar el resultado esperado.
Veamos algunos ejemplos:

EJEMPLO 1: UN EJEMPLO CLÁSICO

ID	TÍTULO	PASOS + DATOS	RESULTADO ESPERADO	PRECONDICIONES
2	Registrar un cliente válido	1. Ingresar a la url: "https://miCarritoDeCompras.com" 2. Presionar botón "Registrarme" 3. Completar el formulario con: a. Nombre y Apellido: Antonella Zanetti b. Fecha de Nacimiento: 22/07/2002 c. Nacionalidad: Argentina d. DNI: 39123987 e. Usuario: anto.zane f. Password: 22AntoZ4ne 4. Clickear en botón "Aceptar"	Se debe visualizar la home con el nombre de Antonella Zanetti en el lateral superior derecho	

Ahora veamos qué pasa si creamos un caso de prueba que se encuentra más avanzado en el workflow de funcionalidades.

Observar en el siguiente ejemplo la variación en el campo de Pasos + Datos:

ID	Título	Pasos + Datos	Resultado Esperado	Precondiciones
15	Buscar producto existente	1. Loguearse 2. Ingresar el nombre “Cafe” 3. Click en botón “Buscar”	En el listado de productos se visualiza: Café Arlistan Cafe Nescafe Cafe Cabrales Cafe Dolca	Se debe contar con productos de de marcas de café

Analicemos:

En este caso, en lugar de escribir detalladamente todos los pasos, sólo mencionamos los módulos hasta llegar a la funcionalidad que realmente queremos testear.

De esta manera podemos completar el campo de Pasos + datos, con 3 posibles opciones:

1. Indicando todos los pasos para llegar a la funcionalidad, como se observa en el Ejemplo 1 (sobre todo en las primeras interacciones con la app)
2. Mencionando los “módulos” (funcionalidades generales), dando por hecho que el/la tester sabe llegar hasta allí, como se observa en el resto de los ejemplos
3. Indicando los casos de prueba (CP) anteriores que hacen al camino para llegar a la funcionalidad que finalmente queremos probar.

Ahora veamos otros ejemplos:

Ejemplo 2

Veamos un ejemplo más, con un caso positivo y otro negativo:

ID	Título	Pasos + Datos	Resultado Esperado	Precondiciones
23	Pagar con una tarjeta de crédito Visa válida	1. Loguearse 2. Buscar producto 3. Cargar carrito 4. Ingresar a Pagos 5. Ingresar el DNI 20234981 6. Seleccionar la tarjeta Visa 7. Ingresar el nro de tarjeta: 4751238765124123 8. Ingresar la fecha de vto 04/23 9. Ingresar el cód. de seguridad: 216 10. Clickear en botón "Pagar"	- Se debe visualizar el mensaje "El pago se concretó satisfactoriamente". **Opciones**: 1. Se debe encontrar en la tabla Clientes_Pagos un registro con la tarjeta terminada en 4123 asociada al cliente 341 2. Al pegarle al servicio "pago/visa" debe traer la tarjeta terminada en 4123 asociada al cliente 341	Debe existir el cliente con DNI 20234981
24	Pagar con una tarjeta de crédito Visa inválida	1. Loguearse 2. Buscar producto 3. Cargar carrito 4. Ingresar a Pagos 5. Ingresar el DNI 12345678 6. Seleccionar la tarjeta Visa 7. Ingresar el nro de tarjeta: 111122233334444 8. Ingresar la fecha de vto 01/23 9. Ingresar el cód. de seguridad: 216 10. Clickear en botón "Pagar"	Se debe visualizar el mensaje "Los datos de la tarjeta no son válidos para dicho cliente" **Opciones**: 1. No se debe encontrar en la tabla Clientes_Pagos un registro con tarjeta terminada en 4444 2. Al pegarle al servicio "pago/visa" no debe traer la tarjeta terminada en 4444	

Analicemos:

Observación 1: notar que para indicar la clasificación de cada caso de prueba, marcamos en verde los casos positivos y en rojo los negativos. Esto es sólo una sugerencia en caso de utilizar una planilla excel como herramienta. En caso contrario se puede utilizar alguna herramienta que contemple la gestión de casos de prueba. En dicho caso, la manera de identificar cada uno quedará a criterio de su configuración en base a la decisión del equipo.

Observación 2: en el campo "Resultado esperado", en esta oportunidad al tratarse de un registro nuevo será necesario, además de la validación en la aplicación, revisar si los datos quedaron consistentes. Para ello, hay varias opciones dependiendo del repositorio de datos con el que se cuenta (BBDD o servicio) o incluso del acceso que tenga el/la tester para consultar la información. Si bien esta es una prueba de caja negra, es necesario que el equipo de testing tenga un mínimo de alcance para la revisión de los datos; de esta manera será necesario solicitar a quien corresponda dicho acceso. Ya sea directamente a la BBDD, o mediante una vista, o a través de un servicio, o a través de una tabla de "histórico / Backup", respetando lo más posible no intervenir en la lógica del producto, preservando así el concepto de caja negra.

Ejemplo post ejecución:

Ahora imaginemos que ejecutamos estos casos, y tuvimos que reportar un bug debido a que el caso 23 falló. He aquí el ejemplo:

ID	Título	Pasos + Datos	Resultado Esperado	Precondiciones	Bug-ID
23	Pagar con una tarjeta de crédito Visa válida	1. Loguearse 2. Buscar producto 3. Cargar carrito 4. Ingresar a Pagos 5. Ingresar el DNI 20234981 6. Seleccionar la tarjeta Visa 7. Ingresar el nro de tarjeta: 4751238765124123 8. Ingresar la fecha de vto 04/23 9. Ingresar el cód. de seguridad: 216 10. Clickear en botón "Pagar"	- Se debe visualizar el mensaje "El pago se concretó satisfactoriamente". **Opciones**: 1. Se debe encontrar en la tabla Clientes_Pagos un registro con la tarjeta terminada en 4123 asociada al cliente 341 2. Al pegarle al servicio "pago/visa" debe traer la tarjeta terminada en 4123 asociada al cliente 341	Debe existir el cliente con DNI 20234981	#18

Ejemplo 3: Completitud de pantalla

Hay una prueba que es un común denominador de todas las aplicaciones: que las pantallas tengan todos los campos e íconos (botones) necesarios para poder operar. Para estos casos se realiza un caso de prueba específico, que se suele denominar "Completitud de pantalla", en cuyo Resultado Esperado se detallan todos los campos y/o íconos que debe tener la pantalla en cuestión. Es uno de los pocos casos donde no se opera con la pantalla mediante una acción (evento), y por lo cual el campo "Pasos + datos" sólo contendrá el link a la página.

Técnicas de diseño de pruebas

Para comprender este concepto empecemos por pensar en las respuestas a las siguientes preguntas ¿cuántas pruebas podemos realizar para aportar calidad?¿vamos a terminar algún día o las pruebas son infinitas? ¿cómo elegir cuáles sí y cuáles no?
Pues bien, básicamente la respuesta a la 2da pregunta es Sí! Las pruebas pueden llegar a ser infinitas, no en un sentido matemático, sino metafóricamente hablando. Es decir, se nos pueden ocurrir muchísimos casos si tenemos en cuenta que la pregunta característica que ronda el cerebro de un/a tester es "¿qué pasa si?", esto se da porque todo el tiempo queremos llegar a los límites en la aplicación.
Pero lo que sí es finito (en todo sentido), es la variable que ya venimos pensando: **El tiempo**. Es por este motivo que no será posible escribir todas las pruebas que se nos ocurra, y en muchos casos tampoco tendría sentido. Pero afortunadamente, contamos con algunas técnicas que nos van a permitir identificar y discernir qué datos son claves para cada prueba y así poder seleccionar las pruebas que nos permitan aportar calidad al producto en tiempo y forma.

Técnica de Valores Límites

Esta técnica, tal como su nombre lo indica, nos permite probar con los datos que rondan los límites del caso. ¿Por qué es esto? simplemente porque son las situaciones comunes de error al desarrollar. Si bien sabemos que no es lo mismo un ">" (estricto) que un ">=" (mayor igual), es justamente donde se suele cometer errores. Es por ello que esta técnica nos indica que los datos apropiados para realizar la prueba son 3: aquellos que rondan el límite del dato.
Así es que debemos crear 3 casos de prueba: uno para el valor "mayor a", otro para el valor "igual a" y el último para el valor "menor a".

Veamos algunos ejemplos:

- Si estamos probando el campo edad para saber si la persona es mayor o no, deberíamos crear estos 3 casos:
  1. Menor de edad (< 18) → vamos a probar con el dato 17
  2. Cumplio (= 18) → vamos a probar con el dato 18
  3. Mayor de edad (> 18) → vamos a probar con el dato 19

- Si tenemos un campo con una hora (contemplando sólo los minutos). Por ejemplo el de inicio de la clase:
  1. Llegó temprano (< 19:00 hs) →18:59
  2. Llegó justo (= 19:00 hs) → 19:00
  3. Llegó tarde (> 19:00 hs) → 19:01

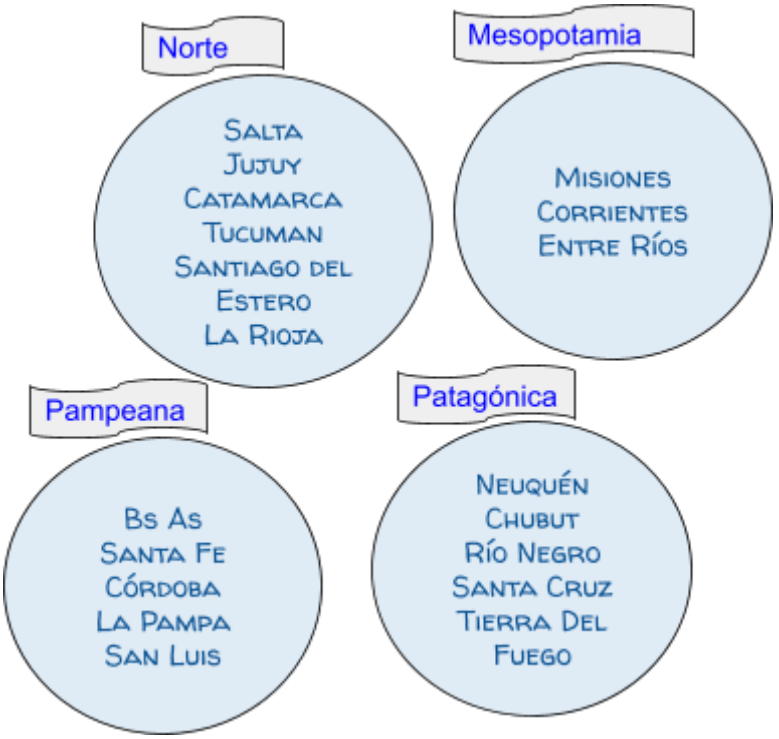
- Queremos extraer dinero ($) del cajero de una caja de ahorro cuyo límite es $5000 y sabiendo que sólo cuenta con billetes múltiplos de 100:
  1. Extraemos menos que el límite (<5000) → 4900
  2. Extraemos justo el límite (= 5000) → 5000
  3. Superamos el límite (> 5000) → 5100

Técnica de Partición de Equivalencias

Supongamos que tenemos que testear un buscador de provincias Argentinas, en el cual se puede filtrar por nombre de provincia y por Región, donde la Región es un combo con las siguientes opciones: Norte, Pampeana, Patagónica, Mesopotamia, Ninguna y Todas). Al buscar, la aplicación muestra la información básica de la provincia.

¿Cuántas pruebas tendríamos que realizar según la combinación de las 23 provincias y sus 6 opciones de región?
¡Demasiadas! Además, como ya vimos, no tiene sentido crear un caso de prueba para cada una. Pero entonces ¿cómo decidir qué datos tener en cuenta para realizar las pruebas?

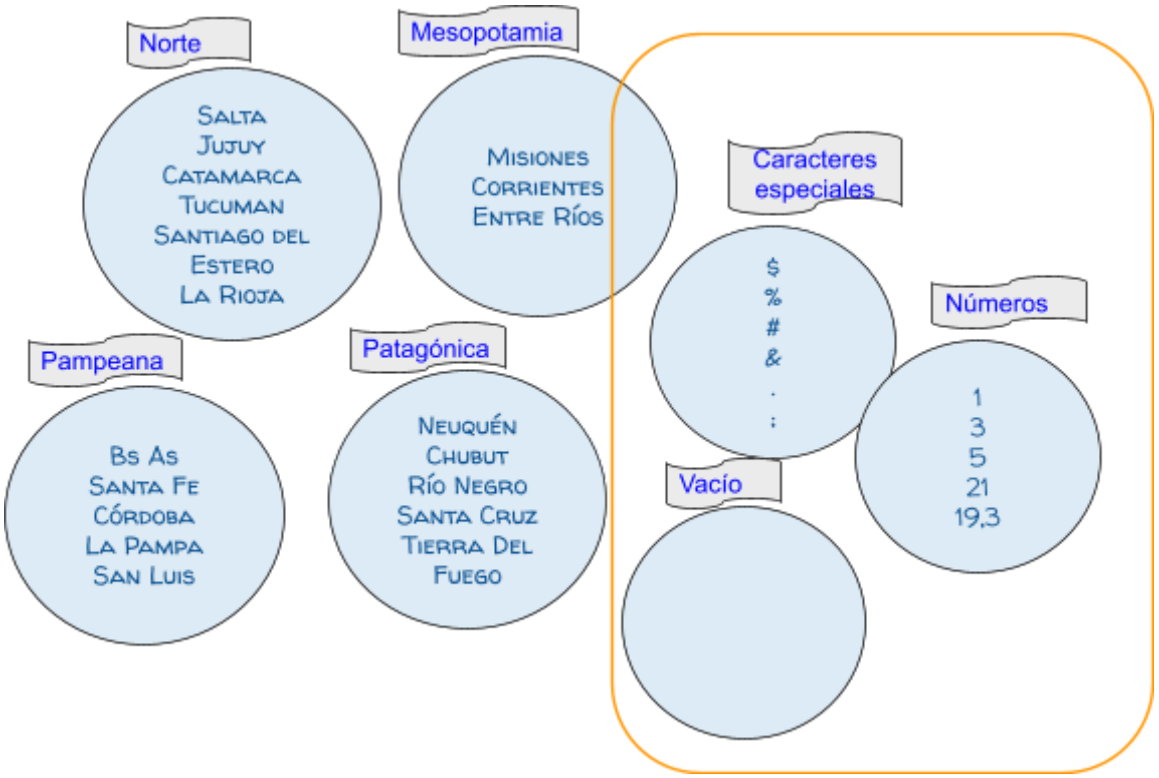
Pensemos entonces que podríamos organizar la información por regiones en forma de conjuntos. De esta manera, quedaría un conjunto por cada región con sus respectivas provincias.



A cada uno de estos conjuntos los vamos a llamar Particiones. Pero ¿a qué nos referimos con "equivalencias"?. Dentro de este contexto, los datos de un conjunto se consideran **equivalentes** cuando tienen el mismo comportamiento para la funcionalidad que estamos probando. Esto es, si probamos buscar la información de la provincia de Salta y la de Jujuy para la región Norte, ¿tienen el mismo resultado esperado? es decir, ¿vamos a encontrar a ambas provincias para dicha región? Obviamente la respuesta es sí!, por lo cual todas las provincias de cada región son equivalentes entre sí. De esta forma, vamos a crear un caso de prueba para una provincia (una cualquiera) de cada región. Todos estos casos son positivos.

Pero faltaría crear las particiones para abarcar los casos negativos. Para ello agregamos los siguientes conjuntos:

1. El vacío: un caracter blanco (con la barra espaciadora) o sin ingresar dato
2. Caracteres especiales / símbolos: colocamos un par, sólo para identificarlos
3. Números: colocamos un par, sólo para identificarlos. Podemos poner tanto enteros, como naturales o reales; dado que en este caso, todos esos datos son parte de la misma equivalencia. Distinto sería si quisiéramos testear la mayoría de edad del ejemplo anterior.



Así, por cada partición que agregamos tendremos un nuevo caso de prueba.
De esta manera nos aseguramos de cubrir todas las casuísticas, pero eliminando casos repetidos o ambiguos.

Técnica de tablas de decisión

Esta técnica en general se utiliza para productos con muchas funcionalidades, y reglas de negocio complejas.
Básicamente consiste en armar una tabla de verdad con las combinaciones de todas las condiciones de las reglas de negocio (entradas) y las acciones/operaciones a realizar en base a ellas (salidas). De esta forma, si tenemos N condiciones se tendrán 2^N combinaciones. Las filas de la tabla se dividirán en 2 partes, una parte para las condiciones y otra para las operaciones, y las columnas serán las respectivas combinaciones.
Primero vamos a completar metódicamente cada combinación con sus respectivos valores booleanos para cubrir todas las posibilidades. Luego se van descartando las redundancias, las contradicciones o ambigüedades, quedando sólo aquellos casos viables y necesarios a testear. Una vez que tengamos las combinaciones de todas las reglas de negocio **válidas**, completamos la sección de las acciones. Se deberá marcar con una cruz cada acción que aplique a la combinación correspondiente.
Por cada combinación se deberá crear un caso de prueba, teniendo en cuenta sus valores de entrada (datos) y la salida correspondiente (resultado esperado).

Aquí un simple ejemplo:

CONDICIONES	1	2	3	4
¿Paga contado?	S	S	N	N
¿Compra > $ 50000?	S	N	S	N
ACCIONES				
Calcular descuento 5% s/importe compra	X	X		
Calcular bonificación 7% s/importe compra	X		X	
Calcular importe neto de la factura.	X	X	X	X

Reporte de bugs

Luego de ejecutar las pruebas, en caso que alguna falle deberemos reportar su respectivo bug. Para ello contamos con una estructura básica de interacción con el equipo de desarrollo.

Estructura de un bug

La estructura e información correspondiente al tipo de incidencia Bug es:

- ID
- Título
- Descripción
  - Resultado esperado vs el Resultado obtenido
- Pasos para reproducirlo
- Versión
- Severidad / Criticidad

Consideraciones importantes:

- El Título es un breve resumen **informativo** de la descripción, esto quiere decir que debe expresar un **error breve pero claro**; describir la funcionalidad donde ocurrió el error **no es un error**!. Ejemplos:
  - **Incorrecto**: “Error en el login” → Esto no informa el error
  - **Correcto**: “El login no valida las credenciales” → Esto sí informa

- La Descripción debe explicar cuál es el error encontrado reflejando la diferencia entre el resultado que se esperaba ver y el que efectivamente se obtuvo. Para enriquecer esta descripción se suele adjuntar la evidencia mediante un screenshot.

Hay herramientas que dividen la descripción en 2 campos, pero en un caso u otro, es condición necesaria detallar ambos.
Veamos un ejemplo de cada uno, utilizando el caso del login que venimos viendo:

Con campos separados:

Resultado esperado: Al loguearse se debe acceder a la home con el nombre del usuario que inició sesión.

Resultado obtenido: No se accede nunca a la home, más allá de las credenciales ingresadas.

Con un sólo campo

Al loguearse se debe acceder a la home con el nombre del usuario que inició sesión, en lugar de eso, no se accede nunca a la home, más allá de las credenciales ingresadas.

Nota: algunos/as testers a pesar de tener un único campo, lo detallan por separado como si fueran 2, utilizando una viñeta para cada caso.

- Los **Pasos para reproducirlo** básicamente son los mismos que los del caso de prueba. ¿Esto quiere decir que no lo debemos explicitar? No, hay que volver a escribirlos, dado que los casos de prueba son documentos para el equipo de testing, al cual no accede el equipo de desarrollo; mientras que el reporte de bug se va a asignar al equipo de desarrollo.

- La **Versión** del producto que donde se encontró el error
- La **Criticidad / severidad** mide cuán urgente se debe resolver el error reportado, y suele contener las siguientes opciones:
  - Crítico/urgente
  - Alto
  - Medio
  - Bajo

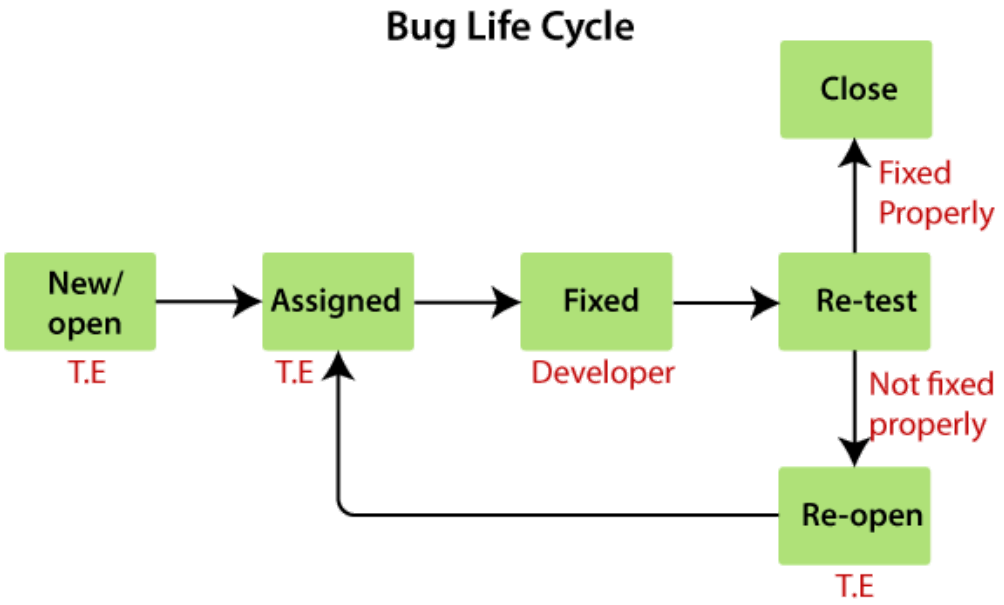
Definir qué valor le corresponde a cada bug dependerá de la cualidad/capacidad del tester que lo está reportando, el cual se basará en el conocimiento que tenga del producto con respecto al negocio. Esto no quiere decir que siempre sea prioritario. La prioridad y severidad son medidas diferentes. La severidad la va a definir el/la tester, mientras que la prioridad el/la PO / cliente / analista funcional. Un bug puede ser crítico, pero no prioritario para el negocio, porque por ejemplo, puede ser una funcionalidad que deje de existir, o se deba modificar en base al negocio.

Veamos un ejemplo utilizando el CP del ejemplo anterior:

ID	Título	Descripción	Pasos para reproducir	Versión	Severidad
18	No se informa el mensaje de pago de visa	Al pagar con una tarjeta de crédito visa se debe visualizar el mensaje "El pago se concretó satisfactoriamente", en cambio la aplicación no informa al respecto. No se observa ningún mensaje, cuando la transacción efectivamente se realizó en la BBDD.	1. Loguearse 2. Buscar producto 3. Cargar carrito 4. Ingresar a Pagos 5. Ingresar el DNI 20234981 6. Seleccionar la tarjeta Visa 7. Ingresar el nro de tarjeta: 4751238765124123 8. Ingresar la fecha de vto 04/23 9. Ingresar el cód. de seguridad: 216 10. Clickear en botón "Pagar"	v1.1	Media

Workflow del ciclo de vida de un bug:

Siendo que el reporte del bug implica una interacción con varios miembros del equipo (en caso del agilismo) o incluso diferentes equipos (en caso de cascada), será necesario tener control y realizar un seguimiento del estado de las incidencias de bugs reportadas. Es por ello que un bug pasa por diferentes estados desde que se inicia hasta que finaliza.
La cantidad y nombre de cada estado será definido por el equipo, así como también su workflow. A continuación veremos los estados mínimos y estándares que conforman este ciclo de vida:



Tener en cuenta que antes de cerrarlo, el equipo de testing deberá volver a probar (ejecutar el caso de prueba) para validar que efectivamente se resolvió.
También es importante destacar que no siempre que un bug se resuelve/cierra es porque se resolvió el error, se puede cerrar por otros motivos, como ser:

- No se pudo reproducir
- Quedó obsoleto
- Está duplicado

Técnica TDD

Esta práctica se llama Test Driven Development, que significa Desarrollo guiado por pruebas. Es importante hacer hincapié en su nombre, el cual explicita que se trata de una **técnica de desarrollo** (manera de desarrollar) y no de un tipo de testing. Al "hacer TDD" no estamos haciendo pruebas, estamos codificando mediante pruebas, que no es lo mismo. Esta técnica es muy empleada en los equipos ágiles dado que favorece la filosofía iterativa e incremental. Mediante TDD se puede programar las US del producto de manera incremental y aportando calidad al software.

Esta técnica se basa en 2 etapas:

1. Escribir las pruebas primero
2. Refactorización del código

Les comparto un resumen gráfico de la técnica:

Bonus: les dejo esta muy breve presentación a modo de resumen

Integración continua (CI)

Hasta ahora hemos visto muchas prácticas y conceptos para velar por la calidad de los productos de software, siendo éste uno de los pilares importantes de la metodología ágil.

Pero ¿cómo aseguramos la calidad en cada iteración? ¿es posible tener un control al momento de integrar cada parte mientras vamos iterando?

Pues para ello, vamos a unificar 2 tipos de pruebas: las de integración, con las de automatización. De esta manera es que vamos a aprovechar la práctica de Devops, que se ajusta muy bien a esta necesidad: La integración continua (Continuous Integration)

Cito la definición provista por Atlassian:

"La integración continua (CI) es la práctica de automatizar la integración de los cambios de código de varios contribuidores en un único proyecto de software."

Permite al equipo de desarrollo fusionar con frecuencia los cambios de código en un repositorio central donde luego se ejecutan las compilaciones y pruebas. Un sistema de control de versiones del código fuente es el punto clave del proceso de CI.

Prácticas de CI

Los ingredientes clave de la integración continua son los siguientes:

- Un sistema de control de fuentes o versiones que contenga todo el código base, incluidos los archivos de código fuente, las bibliotecas, los archivos de configuración y los scripts
- Scripts de compilación automatizados
- Tests automatizados
- Infraestructura en la que ejecutar las compilaciones y las pruebas.

Ventajas

- El equipo de desarrollo puede detectar y solucionar problemas de integración de forma continua, evitando el caos de última hora cuando se acercan las fechas de entrega.
- Disponibilidad constante de una versión para pruebas, demos o lanzamientos anticipados.
- Ejecución inmediata de las pruebas unitarias.
- Monitorización continua de las métricas de calidad del proyecto.

UNIDAD 6: MÉTRICAS

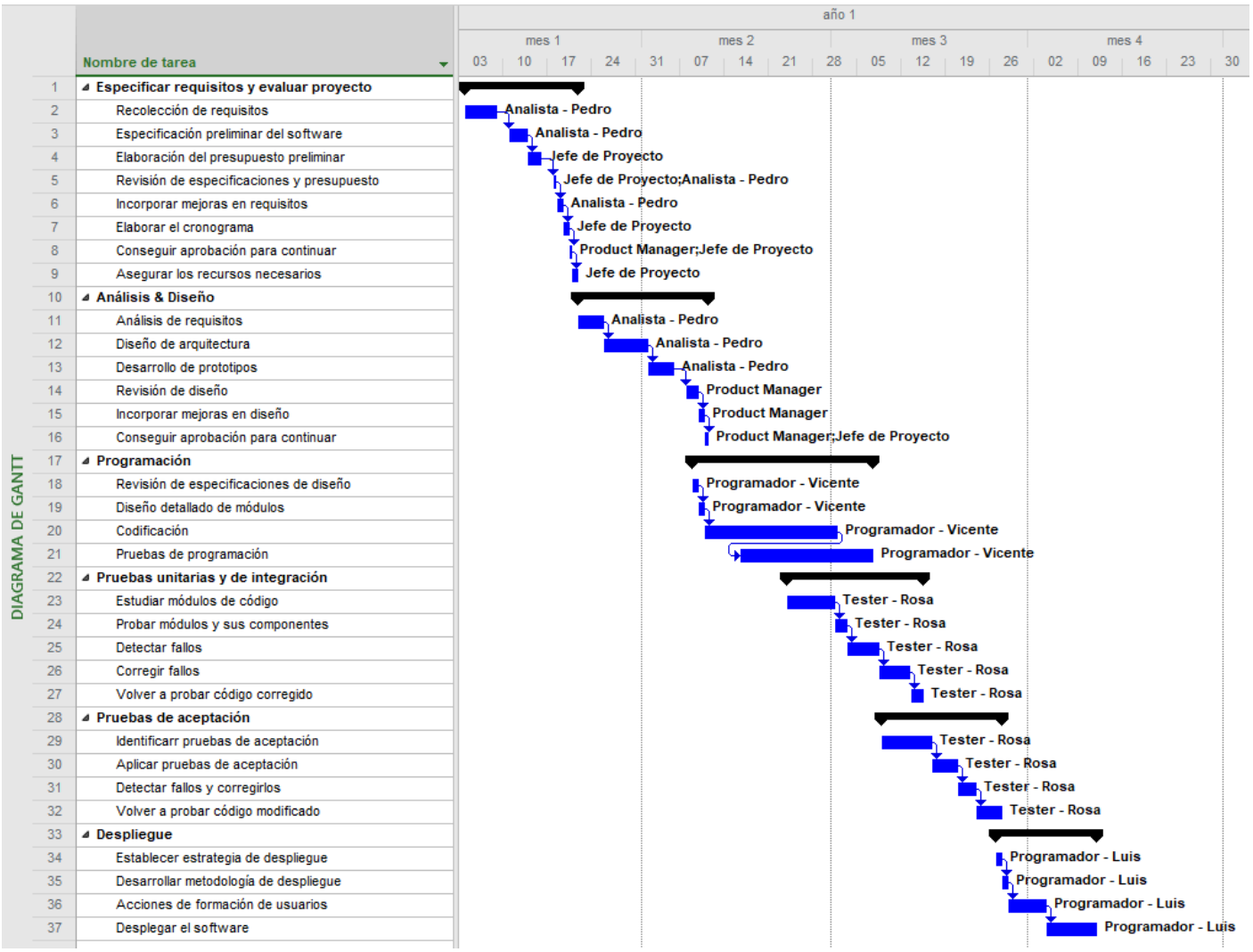
Como parte de la gestión de un proyecto en general, y de software en particular, será necesario tomar ciertas medidas para evaluar si el proyecto se encuentra encaminado según el objetivo planteado. Básicamente necesitamos contar con herramientas y procesos que nos permitan evaluar el estado del proyecto en un momento dado. Para ello es que se utilizan las métricas, las cuales dependerán de la metodología de trabajo empleada.

CASCADA

DIAGRAMA DE GANTT

Siendo que Cascada sigue el desarrollo por etapas, y por ende la planificación a largo plazo desde el inicio del proyecto, será necesario que sus métricas reflejen la evolución del mismo en cada etapa conforme avanza el tiempo, así como también la disponibilidad de las personas que trabajan en el proyecto. Para esto se utiliza una herramienta que permite detallar las tareas de manera cronológica, a modo de poder estimar, muy aproximadamente, la evolución y finalización del proyecto; indicando así una posible fecha de entrega del producto. La herramienta de gestión utilizada para ello es el Diagrama de Gantt.

Aquí un Ejemplo:



Estimación por tiempo

En Cascada al utilizar el diagrama de Gantt, la unidad de medida para la estimación del desarrollo del proyecto es el tiempo. Seteando cantidad de días a cada tarea (y horas para aquellas más pequeñas) para estimar su fecha de finalización. Si bien una estimación es una aproximación, es decir no es un dato preciso, la estimación por tiempo para tareas muy grandes incrementa el nivel de incertidumbre, bajando así la tasa de precisión para con la fecha final. Esto se debe a que el método es muy subjetivo y en general suele estar a cargo de sólo una persona o un grupo muy reducido; esta persona suele ser el/la PM (Project Manager - Gerente del proyecto).

Agile

Estimación por puntos

Para la metodología ágil en cambio, por las desventajas mencionadas previamente, no se suelen estimar las historias de usuario en tiempo, sino que se estiman en puntos de complejidad llamados: Story Points o Puntos de Historia.

Estos puntos representan cuán complejo es desarrollar (programar y testear) la funcionalidad. Por lo que, a cada historia de usuario se le asigna un puntaje que representa la estimación de dicha complejidad. Otra diferencia con cascada, es que en este caso la estimación la realiza el equipo completo. En el refinamiento o, como última instancia en la planning, el equipo deberá estimar cada US para poder planificar el sprint. Como técnica para discernir que este acto sea lo más justo y democrático posible se utiliza la técnica de Planning Poker.
El planning poker consiste en estimar utilizando cartas con una escala de puntos. El puntaje es un valor numérico (natural) extraído de la serie de Fibonacci. Es decir, no es una escala con todos los valores de la recta numérica. La práctica requiere que cada miembro del equipo tenga su propio mazo de cartas con el cual estimar las US. Antes de ahondar en el proceso, será necesario que el equipo elija una funcionalidad conocida por todos/as y que acuerden qué puntaje se le asignaría, la idea es contar con un punto de partida a modo de referencia para, al momento de estimar el resto de las historias, tener una con la cual comparar. A esta referencia se la llama Pivot. Es muy común que, por ejemplo, se utilice el **login** como pivot.

El proceso para estimar cada historia es el siguiente:

1. Análisis: se analiza la historia a estimar entre todo el equipo (se lee, se analizan y entienden las tareas técnicas, se debate, etc)

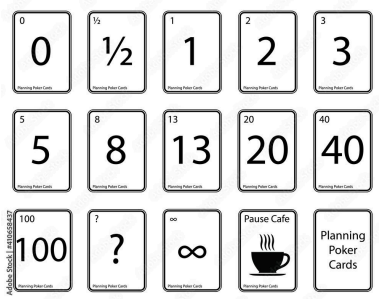
2. Elección del puntaje: cada persona elige una carta de su mazo, cuyo valor estima que representa la complejidad de dicha historia en base al análisis y debate previo, comparandola con el pivot. Esta elección es netamente personal e individual, es decir, la persona no debe sesgar al resto del equipo con comentarios o gestos que hagan alusión al valor (dificultad) seleccionado. Es muy importante que cada persona haga su proceso mental para con la elección del valor, para así poder enriquecer la planning y eventualmente al equipo. Es muy común que las personas con más seniority sesgen al resto del equipo.

3. Exposición del puntaje: cada persona “juega” la carta sobre la mesa o simplemente la muestra

4. Resolución: se observan los puntos de cada persona. Si hay unanimidad se estima con dicho puntaje, en caso contrario se debate al respecto, donde cada persona expone su punto de vista y argumenta el puntaje elegido. Luego del debate se vuelve al punto 2) hasta conseguir unanimidad.
   Si luego de varias rondas no se consigue unanimidad, se puede acordar democráticamente el valor que eligió la mayoría.

Ahora vamos a detenernos en los posibles valores que nos provee esta técnica. Para entender la ventaja de utilizar la serie de fibonacci, será necesario focalizar en el patrón que nos presenta.

Veamos las cartas del Planning Poker



Analizando este patrón, observamos que en las cartas con valores más chicos, la diferencia entre ellos (su resolución) es menor con respecto a las cartas con valores más grandes. Pero ¿por qué esto es una ventaja para estimar?

Pues porque para las historias más chicas se puede dar un debate más reñido e interesante dado que no da igual 1 que 3 puntos, hay una clara diferencia en la representación del esfuerzo. En cambio, cuando la US es muy grande, y todo el equipo coincide en esto, da igual que mida 20, 33 o 40 puntos; en cuyo caso ni siquiera amerita debate alguno. El equipo ya se dió cuenta que en realidad se trata de una épica más que una historia de usuario, y como tal, deberá aplicar algún patrón de slicing para descomponerla y así obtener las US que sí se podrán estimar.

Entre los valores además vemos algunos casos “especiales”:

- El valor 0: indica que no necesitamos estimarla
- El valor 1/2: es demasiado pequeña, casi que no incide en la velocidad
- El valor “?”: implica que se desconoce o no se termina de comprender lo que hay que hacer. Para el primer caso, si todo el equipo siente lo mismo, se deberá crear un spike, mientras que en el 2do caso, es un buen momento de volver a consultar para sacarse las dudas. Si se puede resolver en el momento, se estima, pero en caso contrario, se deja en el backlog para consultar más adelante.
- El valor “∞”: representa que es muy grande y que esa persona ya se dió cuenta que en realidad es una épica, y que no tiene sentido “jugarse” por elegir un valor particular.
- La taza de café: es el símbolo que representa “momento de break”. Se solía utilizar cuando no se contaba con la actividad de refinamiento, y por ende todas las tareas se solían realizar en la planning; de ahí que el tiempo dedicado era tan largo que ameritaba hacer breaks cada tanto.

Es importante recordar que un equipo ágil se conforma con todos los roles involucrados, por lo cual de la estimación participan tanto quienes desarrollan como quienes testean.

Nota: las cartas pueden ser físicas o se puede utilizar alguna app que permita seleccionar el valor correspondiente.

Burndown Chart

Siendo que esta unidad hace hincapié en las métricas, así como Cascada organiza las tareas cronológicamente en el calendario del Diagrama de Gantt para medir el proyecto, en la metodología ágil tenemos, a priori, 2 instancias en las cuales podemos conocer el avance del equipo o estado del producto.

1. Durante el sprint: podemos revisar el tablero Kanban
2. Al cerrar el sprint: como parte de la review, se deberá crear el cuadro de Burndown Chart.

Pero ¿de qué se trata este cuadro?

Se trata de un gráfico que refleja, como una especie de "foto", la evolución del trabajo y/o producto durante el sprint. Pero para que este cuadro sea valioso, será necesario durante el sprint, que el equipo se comprometa a cerrar cada incidencia a medida que se vaya finalizando.

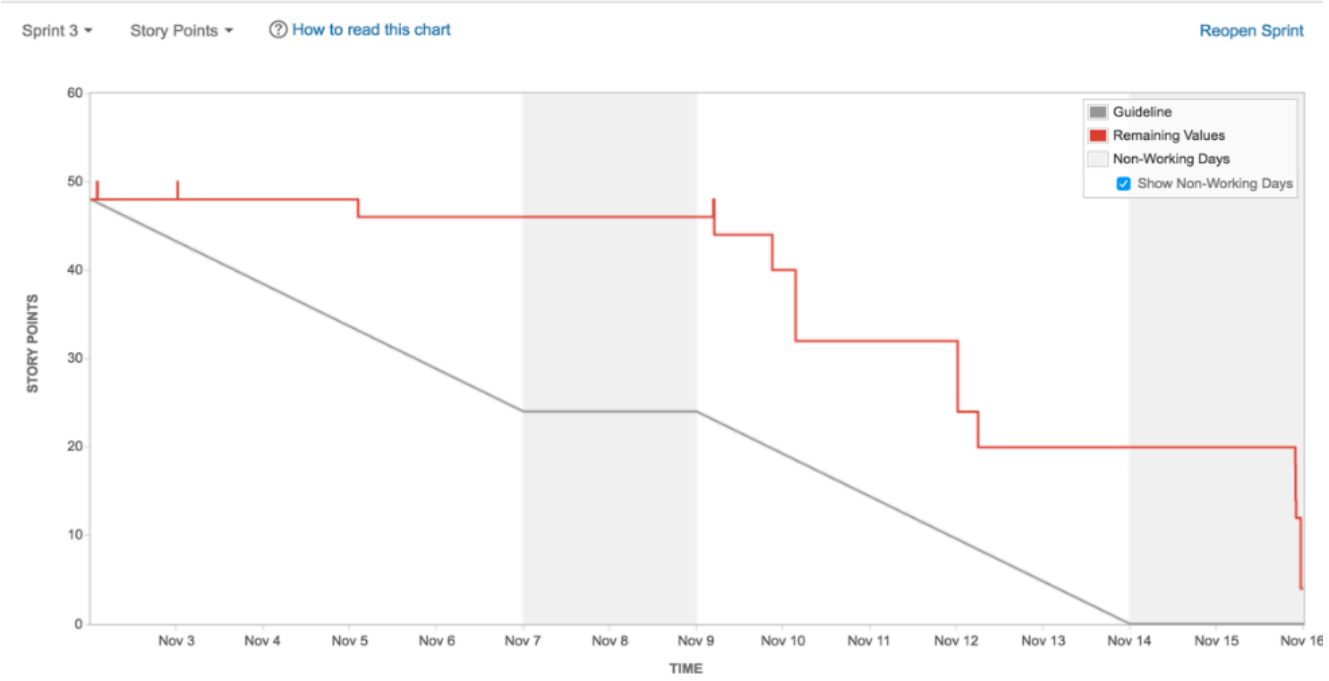
Si bien el concepto de "cerrar incidencia" depende del workflow utilizado, un clásico estándar implica "pasar" la tarea al estado "Done" (hecho) del tablero Kanban. A esta acción, el Burndown Chart la denomina "quemar puntos". Esto se debe a que cuando se inicia cada sprint, el cuadro inicia el eje vertical (y) con la cantidad total de puntos a "quemar", o mejor dicho a entregar. De esta manera, cada vez que se cierra una incidencia (entrega tarea), se queman los puntos estimados, quedando así menos puntos (tareas) a resolver como equipo. Dicha acción se observa como parte de la línea de status dentro del gráfico.

Nota importante, podemos analizar el gráfico desde 2 visiones:

- Avance del producto: se focaliza en los puntos quemados
- Avance del equipo: se focaliza en las tareas resueltas

Lo importante de estas visiones es que no existe ninguna que focalice el trabajo individual, como parte de la metodología ágil, el foco siempre está puesto en el trabajo de equipo vs el individual.

A continuación un ejemplo de un gráfico de burndown chart:



El eje x setea las fechas del sprint, y el eje y los puntos a quemar según lo planificado y comprometido al iniciar el sprint.
La línea tangente (descontando el fin de semana) representa el ideal según la velocidad del equipo, mientras que la línea roja representa el estado por fecha de la quema de puntos. De esta manera se puede medir si el equipo va adelantado o atrasado con las tareas planificadas del sprint.

Velocidad del equipo

A modo de resumen podemos decir que la herramienta para medir la productividad de un equipo o proyecto ágil es el burndown chart, el cual se alimenta de los puntos de historia planificados en cada sprint, que en su conjunto reflejan el trabajo del equipo. Pero ¿cómo sabe el equipo, al momento de planificar, cuál es su capacidad de trabajo, es decir, cuántas historias es capaz de resolver en un tiempo determinado (en un sprint)?
Si bien sabemos que el agilismo no utiliza el tiempo como unidad de medida, en términos de métricas y de contratos, necesitamos tenerlo en cuenta. Entonces la pregunta es ¿cómo sabe el equipo cuándo podrá entregar cierta funcionalidad?
Pues bien, para ello es que se cuenta con el concepto de **velocidad del equipo**. Ésta representa la unidad de medida que tiene el equipo en cuanto a su capacidad de trabajo; puntualmente significa la cantidad **media** de trabajo que el equipo puede resolver en una iteración (sprint), la cual se representa en puntos de historia.
Si bien aún no estamos hablando de tiempo, para responder a la pregunta de "cuándo", simplemente es cuestión de realizar un cálculo. Veamos, si por ejemplo la velocidad del equipo es de 20 pts, para resolver una épica cuya sumatoria total es de 38 puntos, se necesitarán al menos 2 sprints; y suponiendo que cada sprint dura 2 semanas, el equipo tardará 1 mes en resolver la épica.

Para concluir, si bien la velocidad no está expresada en tiempo, al relacionar los puntos con la duración de los sprints se obtiene un tiempo determinado.
Una consideración importante es que la velocidad del equipo va variando. Es común que en los primeros sprints aumente conforme el equipo va aprendiendo sobre el producto, hasta encontrar una etapa de estabilidad. Pero también puede ocurrir que decremente en caso que el equipo sufra modificaciones que lo desestabilicen, como ser vacaciones, licencias, enfermedades, capacitaciones, o incluso que el equipo se recorte; por más que ingresen nuevos reemplazos, lleva un tiempo volver a estabilizar la velocidad. La velocidad es un factor que se debe tener en cuenta cada vez que se realiza una planning. El equipo en base a su velocidad, determinará cuántas US se compromete a entregar en dicho sprint. No olvidar tener en cuenta las circunstancias especiales del equipo, como ser vacaciones; no es lo mismo contar con una persona menos para el sprint.

Conclusión sobre scrum

Para destacar me gustaría concluir la importancia del valor **"Compromiso"** como parte de los pilares de Scrum (notar el resaltado en cada caso, a lo largo del libro).
A modo de resumen comparto 3 compromisos bien definidos que hacen a la metodología ágil en general y a Scrum en particular:

1. El **compromiso** de definir **"El objetivo del producto"**, que permite tener una visión global del producto a construir.
2. El **compromiso** de definir **"El objetivo del sprint"** en pos de ir alcanzando el objetivo del producto en cada incremento mediante las iteraciones
3. El **compromiso** de definir **"La definición de hecho / terminado"** en pos de que cada incremento aporte la calidad suficiente para entregar un software de calidad.

Para profundizar...

Como bonus general les comparto algunos blogs por si les interesa ahondar sobre esta filosofía tan particular y occidentalizada como el agilismo:

- **Dime lo que mides y te diré lo que careces:** https://mamaqueesscrum.com/2020/08/03/dime-lo-que-mides-y-te-dire-de-lo-que-careces/
- Y algunos de mi propia autoría :)
  - **El agilismo más allá del desarrollo de software:** https://medium.com/@warmiguercio/agilismo-m%C3%A1s-all%C3%A1-del-desarrollo-de-software-ced5fa9982e2
  - **El proceso detrás del proceso:** https://medium.com/@warmiguercio/el-proceso-detr%C3%A1s-del-proceso-422a1bdaa68a
  - **No son ágiles las empresas, lo son las personas:** https://medium.com/@warmiguercio/no-son-%C3%A1giles-las-empresas-lo-son-las-personas-45c33ca66101

Bibliografía utilizada:

- Guía de Scrum - Edición 2020
- Proyectos ágiles con Scrum - 2da Edición - Kleer