ELEMENTOS DE INGENIERÍA DE SOFTWARE

(v1.2.2026)

UNIDAD 1: Ingeniería de software

1. Introducción a la Ingeniería de software
2. Conceptos elementales de la ingeniería de software
3. Metodologías de gestión

UNIDAD 2: Ciclo de vida del software

1. Instancia previa al ciclo de vida
2. Ambiente de trabajo

UNIDAD 3: Metodologías de gestión

1. Introducción
2. Enfoques metodológicos
3. Metodología Tradicional
   a. Problemáticas
4. Metodología ágil
   a. Características: solución a las problemáticas
   b. Manifiesto ágil
   c. Ley de Pareto
5. Conclusión

UNIDAD 4: Historia de los métodos de gestión

1. Métodos tradicionales
2. Métodos ágiles

UNIDAD 5: Métodos de gestión

1. Método tradicional: Cascada
2. Método ágil: Scrum
   a. Pilares de Scrum
   b. Valores de Scrum
   c. Elementos de Scrum

UNIDAD 6: Documentación

1. Cascada: Especificación Funcional
   a. Casos de uso (CU)
   b. UML
2. Scrum: Product Backlog
   a. Inception
   b. Incidencias
   c. Técnicas de Slicing de US
3. Diferencia entre requisito, CU y US

UNIDAD 7: calidad

1. Concepto de calidad
2. Clasificación de pruebas
   a. Pruebas funcionales
   b. Pruebas no funcionales

3. Pruebas de Caja Negra
    a. Diseño de casos de prueba
    b. Clasificación de casos de prueba
    c. Técnicas de diseño de casos de prueba
4. Reporte de bugs
5. Técnica TDD
6. Integración continua (IC)

UNIDAD 8: métricas y estimaciones

1. Métricas en Cascada
    a. Diagrama de Gantt
    b. Estimación por tiempo
2. Métricas en Scrum
    a. Estimación por complejidad
    b. Burndown Chart
    c. Velocidad del equipo

CONCLUSIONES GENERALES

UNIDAD 1: INGENIERÍA DE SOFTWARE

1. Introducción 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

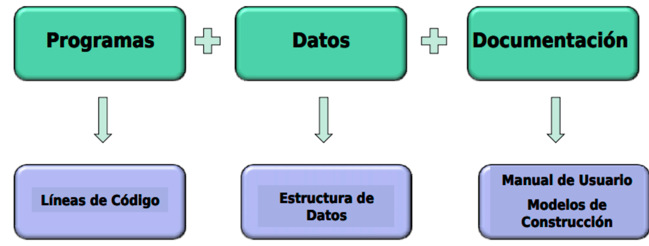
¿Qué es un sistema informático?

Es un sistema que nos permite almacenar y procesar información de manera automática, mediante una serie de partes interrelacionadas, formadas por recursos (hardware y software) y personal humano.

¿Qué es un software?

Es un un conjunto de elementos compuestos por:

1. **Programas**: código fuente
2. **Datos**: estructuras de datos
3. **Documentación**: manuales



Características

- Es un elemento lógico y no físico. Usable mediante una interfaz
- No se estropea pero se degrada. Es mantenible
- No tiene piezas de repuesto
- No se produce en serie. Se construye a medida y es reusable.

Diferencias técnicas claves

Aunque los siguientes términos a menudo se suelen utilizar como sinónimos, existen diferencias técnicas clave entre: programa, aplicación, software y producto de software.
A grandes rasgos, un **programa** consta de un código fuente que finalmente es ejecutable en un **hardware** (parte física). Mientras que el **software** (parte lógica) consta de un conjunto de programas, datos y documentación respaldatoria que permitirá su aplicación dentro de un **sistema informático**. Una **aplicación** es un software construido para un usuario final, que permite una interacción con la misma. Ahora bien, si el software o la aplicación se convierten en un objeto negociable dentro de una transacción comercial entre un cliente y un proveedor, podemos decir que éste se transforma en un **producto de software** (empaquetado) desarrollado con estándares de calidad y con el fin de ser comercializado.

¿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)

¿Es un producto o un 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?

Será necesario entonces incorporar las siguientes actividades:

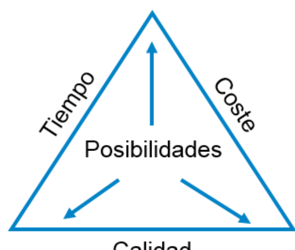
- Planificación
- Control y seguimiento: métricas
- Calidad
- Gestión

Entonces, ¿qué tipo de habilidades y capacidades deben tener quienes construyen software?

- Creatividad
- Lógica
- Estructura
- Capacidad de resolución
- Ley del mínimo esfuerzo
- Adaptación para aprovechar los recursos ya existentes sin reinventar la rueda

¿Qué es un proyecto de software?

Es un conjunto de actividades planificadas que permiten construir en un tiempo determinado un **producto de software**, el cual deberá cumplir con su **ciclo de vida** desde el relevamiento de los requisitos hasta su puesta en marcha, y cuya planificación se deberá realizar bajo el marco de una **metodología de gestión**.
De esta manera será necesario equilibrar las variables que intervienen, teniendo en cuenta que en la mayoría de los casos será necesario resignar alguna de ellas: **tiempo, costo o calidad**.



3. Metodologías de gestión

Una metodología de gestión consta de un conjunto de técnicas y prácticas que le permiten a un equipo llevar adelante un proyecto, puntualmente en nuestra industria, para el desarrollo de un software. Cada metodología cuenta con un **enfoque** específico y con **métodos** o **frameworks** (marcos de trabajo) adecuados para su aplicación.

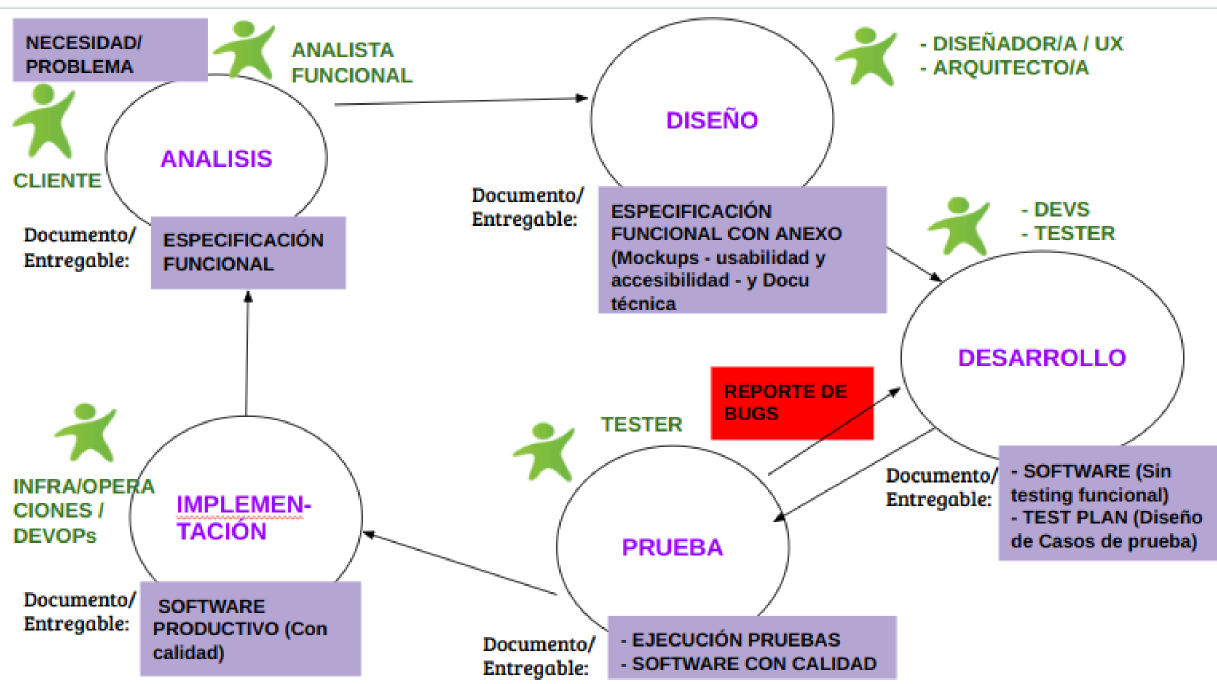
¿En qué tipo de industrias se pueden implementar proyectos de software?

Un proyecto de software se puede implementar y por ende gestionar básicamente en cualquier industria que requiera un producto de software. La industria y el mercado durante muchos años han estado en constante crecimiento, originando el surgimiento de empresas con características y objetivos variados. Por lo pronto mencionaremos 3 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 bancaria, industria automotriz, industria de la salud, etc.

UNIDAD 2: CICLO DE VIDA DEL SOFTWARE

Antes de ahondar en las metodologías de gestión, será necesario entender cómo es el ciclo de vida de un producto de software. A continuación se muestra un diagrama que contempla sus etapas, sus roles y el 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 del **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 actividades, 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, así es que se entrega la **especificación funcional con su anexo (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 entregable de esta etapa resulta en el **producto de software sin calidad** (desarrollado pero sin testing funcional) y un **plan de pruebas: test plan (output)**.

4. **PRUEBA**: en base al **producto de software (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 con calidad (output).**

5. **IMPLEMENTACIÓN**: contando con un **producto de software con calidad (input),** el mismo deberá ser implementado en el ambiente de producción, es decir ponerlo en marcha para el uso público. Para esto, se solicita al equipo, comúnmente denominado **operaciones o infraestructura**, que realice el deploy del producto. Esto 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).

Instancia previa al ciclo de vida

El ciclo de vida del software inicia luego de concretarse un acuerdo entre el cliente y el proveedor sobre el proyecto de software (no sólo el producto). Es decir 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 de 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 proyecto.

Ambiente de trabajo

Un ambiente de trabajo 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 específica para operar en dicho espacio.

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, del 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, además de los ya mencionados, coexistan el ambiente de testing (o QA), el de integración y uno de UAT (user acceptance test) o también denominado pre-productivo.

UNIDAD 3: METODOLOGÍAS DE GESTIÓN

Introducción

El desarrollo de software no es una tarea sencilla y como hemos visto consta de varias etapas. Los productos de software, a diferencia de productos de otras industrias, no se caracterizan por ser estáticos, donde se construyen bajo un concepto definido y conocido, y se entregan a un cliente final. Los productos de software, por el contrario, se encuentran en constante evolución dentro de contextos cambiantes que demandan un crecimiento de sus funcionalidades y modificación de las ya existentes. Así es como el desarrollo de un producto de software termina formando parte de un proyecto con actores que lo gestionan. Y para que la gestión resulte exitosa será necesario contar con una metodología de trabajo. A su vez, las metodologías de trabajo responden a conceptos y marcos teóricos a partir de los cuales se basarán luego los métodos prácticos que las lleven a cabo.

Las metodologías que abordaremos son:

- **La metodología tradicional**
- **La metodología ágil**

Enfoques metodológicos

Cada una de estas metodologías sigue un **enfoque** diferente en base a la manera de abordar las etapas del ciclo de vida.

La **metodología tradicional** sigue un **enfoque secuencial** o también llamado lineal dado que implementa cronológica y secuencialmente cada una de las etapas del ciclo de vida del software. Es decir que hasta que no termine una etapa no se puede avanzar a la siguiente. Así, cada etapa resuelve una parte del proceso (no del producto).

La **metodología ágil** en cambio, sigue un enfoque **iterativo e incremental** donde implementa todo el ciclo de vida del software (cada etapa) en pequeñas **iteraciones** a partir de las cuales se obtiene como resultado un **incremento** del producto por cada iteración.

Es importante destacar que estas metodologías imparten un marco teórico pero no hacen mención de su implementación en términos empíricos (prácticos) para dicho enfoque.

Metodología tradicional

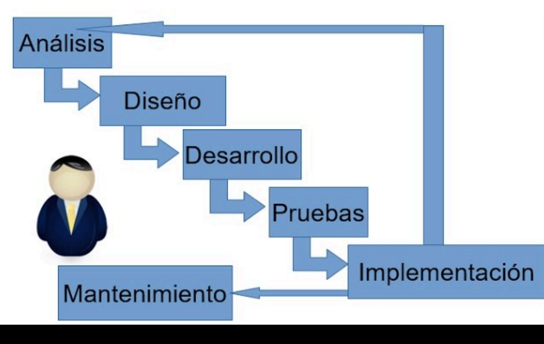
Contexto histórico

El desarrollo de los sistemas (así denominado en ese entonces) mediante un proceso 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 implemente cada una de las etapas del ciclo de vida del software tal y como se implementa en las industrias automotrices. Así es como nació, lo que denominaremos en el presente libro, la metodología tradicional junto con su método inicial Cascada.

Características

En la industria automotriz o en las manufactureras, 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 montar la carrocería de un auto si aún no se tiene el chasis"*.**
En la ingeniería de software se buscó replicar el mismo modelo, donde una de las etapas no puede iniciarse hasta que la etapa anterior se encuentre completa. Otra característica fundamental y consecuente de este enfoque es que la definición del producto se detalla de manera **completa y total** al inicio del proceso.

Veamos un gráfico representativo:



Problemáticas y desventajas

Los productos de software no comparten las mismas características que los productos de las industrias tomadas como modelo, motivo por el cual esta metodología no es la más apropiada para el desarrollo de un producto de software. Dichas industrias construyen productos tangibles, con un formato conocido y repetible, mientras que un producto de software es abstracto y no se fabrica en serie, por lo que no es reproducible. Por dichos motivos es que la implementación de esta metodología trajo aparejado los siguientes problemas:

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 a la metodología en un proceso muy burocrático y propenso al arrastre de errores entre cada etapa. Asimismo **todos los requerimientos** se deben determinar al inicio del proceso, por lo que una vez finalizada esta etapa no hay espacio para su revisión y modificación, convirtiendo todo lo relevado en una pieza fundamental para el éxito del proyecto. No hay espacio para el cambio o para el "malentendido" en término de las funcionalidades. Motivo por el cual se consigue, en la mayoría de los casos, una documentación incompleta y/o errónea.
3. **RIGIDEZ:** los desarrollos de este tipo pueden durar meses sin que el cliente vea el avance del producto. Recién en su entrega final el cliente tiene acceso al mismo, razón por la cual se dificultan las actualizaciones de los requisitos y/o la solicitud de nuevas funcionalidades. Siendo que la metodología se caracteriza por la definición total y completa de los requisitos en una etapa muy temprana se terminan generando ciclos de desarrollo inflexibles y poco adaptables a los cambios.

4. **COSTOSO**: ante los errores detectados, el costo en la corrección del producto puede ser altísimo. Esto se deba a:
    a. **El desconocimiento del origen del error:** si se desconoce el origen del error se deberá realizar ingeniería inversa de cada etapa hasta detectarlo.
    b. **La corrección del error:** de tratase, en el mejor de los casos, de un error en la etapa de desarrollo, su resolución podría ser relativamente sencilla pero si el error se produjo en la etapa de diseño, por ejemplo o aún peor, en la etapa de análisis, el costo de su corrección es realmente alto, siendo que se deberán corregir todas etapas.
5. **EXPECTATIVA VS REALIDAD:** siendo que el proceso consta de largos períodos con poca participación del cliente y un relevamiento completo al inicio, es muy probable que en la entrega del producto, éste no logre satisfacer las expectativas del cliente, quien termina por recibir un producto no deseado y que no representa lo que realmente tenía en mente.
6. **ROI TARDÍO:** el cliente desconoce la rentabilidad de su producto hasta el momento de su entrega, lo que afecta a su retorno de inversión (Return on investment - ROI) incluso en el mejor de los casos, donde la llegada al mercado puede ser demasiado tarde para ser competitivo. Esta situación aún puede empeorar si el producto no satisface las expectativas del cliente y debe ser modificado, generando así una inversión significativa con un retorno nulo o incluso negativo.

Como conclusión esta metodología no es muy propicia para proyectos comerciales en contextos muy 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 cuentan con estas características y es por tal motivo que está cada vez más en desuso y siendo reemplazada por la metodología ágil.

Metodología ágil

Contexto histórico

Con el paso del tiempo, la construcción de productos de software bajo la metodología tradicional, implementada por el método Cascada, estaba siendo cada vez más costosa. Las problemáticas de este método que no contemplaban los imprevistos y cambios, llevaron a una tasa de fracasos muy elevada dentro de una industria con un mercado cada vez más cambiante, dinámico y competitivo. Esto generalmente se relacionó con la entrega tardía del producto o un presupuesto excesivo. Dicha situación propició durante la década del 90', el surgimiento de varios movimientos identificados con el nombre de "metodologías livianas" (Lightweight Methodologies), los cuales cuestionaron el enfoque de la metodología tradicional proponiendo uno diferente en contraposición. Y visto que todas comparten el enfoque ágil es que en en el presente libro, se diferencia entre metodología, aquella con un enfoque teórico, y método, como los marcos prácticos que siguen los enfoques metodológicos. Así es que en la próxima unidad se detallarán los métodos que sigue cada metodología y su evolución a lo largo de la historia.
Por ahora continuemos especificando las características de la metodología ágil.

Características: solución a las problemáticas

Como primera medida se cambió el enfoque secuencial para pasar a un enfoque iterativo e incremental. De esta manera, el producto ya no se define (ni desarrolla) de manera completa y rígida al inicio del proyecto, propiciando la participación del cliente en una etapa más temprana, y cuyo feedback permita adaptar el producto a un mercado cada vez más cambiante.
Este enfoque conlleva al desarrollo del producto parte por parte, de manera que al inicio del proyecto se entrega un producto más pequeño, pero con la necesidad de construir incrementos que permitan evolucionar la idea original, y como consecuencia, iterar el proceso para llevarlos a cabo. Así es como el enfoque derivó al iterativo e incremental.

El Manifiesto Ágil

Como hemos anticipado al inicio de la unidad, la metodología ágil se imparte mediante diferentes métodos. Con el fin de estandarizar y normalizar dichos métodos es que en febrero de 2001 se reunieron en EEUU un grupo de profesionales reconocidos del desarrollo de software y referentes de las metodologías livianas, para crear de manera oficial un **manifiesto** firmado por quienes ya venían aportando ideas a esta metodología. El manifiesto no es más ni menos que un espacio unificado que conglomera varios fundamentos ágiles basados en la filosofía Lean.

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

Los valores ágiles son:

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 con una mayor valoración por sobre los segundos.

Aquí un breve análisis de cada uno:

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 la metodología ágil a 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.

Los 12 principios se podrán encontrar en el manifiesto. Los valores son los pilares sobre los cuales se construyen los principios. Hay principios más generales que resumen el espíritu ágil y otros son más específicos y orientados al equipo o al proceso de trabajo.

Ley de Pareto

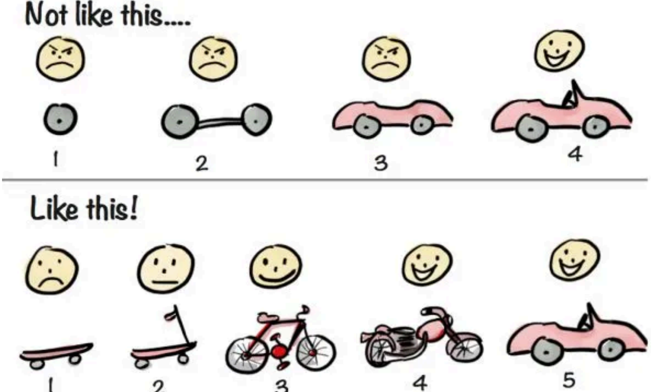
El agilismo como parte de su manifiesto focaliza constantemente en el concepto de **“valor”**, desde las funcionalidades que contendrá el producto hasta las acciones que toma el equipo. En cada decisión, con sus respectivos matices, está planteado e involucrado el aporte de valor que le será redituado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

> **Ley de Pareto:** es una proporción entre 2 variables que nos permite medir en términos de inversión-ganancia o costo-beneficio, donde los valores están dados por la relación del 80% y el 20%.

Por ejemplo, en términos de productividad se identificó que el 20% de las tareas de alto impacto producen el 80% del valor total. En términos de calidad, el 80% de los efectos son producidos por el 20% de las causas. Esta ley también se conoció como el principio de “**los pocos vitales** (el 20% principal que genera el 80% importante) **y los muchos triviales** (el 80% restante que genera el 20% remanente)”.

Conclusión

A modo de conclusión sobre ambas metodologías, cuyas bases se encuentran enteramente sobre sus enfoques. Vamos a ver a continuación la diferencia entre ambas mediante el siguiente ejemplo gráfico. Se desea construir un auto como medio de transporte.



Analicemos la diferencia de enfoques al desarrollar sus funcionalidades con cada metodología:

- La **metodología tradicional** desarrolla de manera **horizontal**: construye las ruedas, luego el chasis, luego la carrocería, el interior, etc.
- La **metodología ágil** desarrolla de manera **vertical**: construye un transporte minimalista que pueda funcionar gracias a los elementos esenciales, luego mejora cada elemento y sólo después añade los elementos secundarios (asientos blandos, pintura, gps, etc).

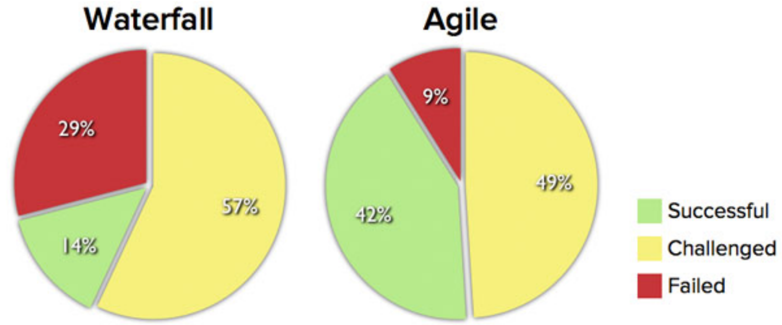
En la primera imagen se muestra un **enfoque secuencial** desarrollado con metodología tradicional, donde el cliente recién conoce y obtiene el producto en la última etapa del ciclo de vida del software, sin tener una visión de su evolución, y a la espera de calcular su retorno de inversión.
En la segunda imagen se observa un **enfoque iterativo e incremental** desarrollado con **metodología ágil**, en el cual se obtiene un producto funcional desde el inicio del proyecto. Si bien es una versión bastante mínima del producto final, el cliente ya puede comercializarlo y así recuperar un porcentaje de lo invertido para continuar con la inversión y seguir evolucionando el producto. Este enfoque permite realizar modificaciones incluso de la idea original.
En síntesis la metodología ágil se diferencia de la tradicional en cuanto al foco sobre la **adaptabilidad** a los cambios, en lugar de la **previsibilidad**, que busca planificar y tener “todo” controlado desde el inicio.
Se comparte una interesante comparación gráfica entre el desarrollo mediante el método incremental, el iterativo, y el iterativo e incremental.

CHAOS Manifiesto

En el año 1994 el Standish Group publicó un estudio conocido como el “CHAOS Report” donde se encontró la siguiente tasa de éxito en los proyectos de desarrollo de software general:

- 31.1% de los proyectos fracasaron, fueron cancelados
- 52.7 % de los proyectos se excedieron en costos y/o tiempo
- 16.2 % de los proyectos fueron exitosos

Recordemos que en 1994 la metodología utilizada era casi exclusivamente la tradicional con el método Cascada. En 2012, once años más tarde de la aparición del Manifiesto ágil, el Standish Group publicó su clásico análisis anual de gestión de proyectos de la industria del desarrollo de software, ahora llamado “CHAOS manifiesto”.
Este reporte incluye una comparación entre ambas metodologías (Cascada / waterfall como representante de la tradicional):



En ese año indicaría que:

“El proceso ágil es el remedio universal para el fracaso en los proyectos de desarrollo de software. Los productos desarrollados a través del proceso ágil tienen tres veces la tasa de éxito del método en Cascada tradicional y un porcentaje mucho menor de demoras y sobrecostos. [...] El producto de software debería ser construído en pequeños pasos iterativos, con equipos pequeños y enfocados. ”

UNIDAD 4: HISTORIA DE LOS MÉTODOS DE GESTIÓN

Hasta el momento, con las metodologías de trabajo, hemos ahondado sobre el desarrollo de software desde un enfoque teórico. Pero también hemos adelantado que el desarrollo se sustenta sobre métodos prácticos que permiten implementar las metodologías de gestión.
Así es que, a continuación, ahondaremos sobre los **métodos de gestión** asociados a cada una y que han ido evolucionando a lo largo de las décadas. Muchos de ellos fueron combinando técnicas y prácticas tradicionales con nuevas e innovadoras ideas que resolvieron las falencias de los métodos tradicionales hasta llegar a los métodos ágiles más conocidos y utilizados en el mercado actual.

En esta unidad vamos a profundizar sólo en los 2 métodos (frameworks) más conocidos de cada metodología:

- **Método cascada (waterfall)**: metodología tradicional
- **Método Scrum**: metodología ágil

Como denominación común y popular se suele abreviar directamente en "método tradicional" y "método ágil" a aquellos métodos o frameworks basados en su respectiva metodología. La denominación correcta sería: "Método de la metodología tradicional". Término que puede generar confusión por ser rebuscado. También hay quienes optan por el término en inglés "Framework de la metodología tradicional", dado que framework significa justamente marco de trabajo. Es importante destacar que si bien el método Cascada y Scrum son los más utilizados de cada metodología, no son los únicos. Lo llamativo es que en el caso de la metodología tradicional, Cascada es el primero y en cuanto al agilismo, Scrum es el último.

A continuación realizaremos, en orden cronológico, un breve **repaso histórico** de los métodos tradicionales y ágiles que han contribuido a lo largo del tiempo y han evolucionado hasta el día de hoy.

Métodos tradicionales

- **CASCADA (waterfall):** modelo presentado por **Winston Royce en un artículo en 1970** donde toma la metodología tradicional (forma secuencial de trabajar) y la convierte en método, atravesando **literalmente** cada una de las etapas del ciclo de vida del software, puesto que éste involucra algunas prácticas para la construcción de un producto de software. Sin embargo, este método se creó debido a una mala interpretación del modelo de Royce, quien argumentó posteriormente que este enfoque de fases estrictas era arriesgado y a menudo defectuoso, proponiendo en su lugar enfoques más iterativos.
- **INCREMENTAL:** creado por **Harlan Mills en 1980**, es una extensión del método de **cascada**, que mantiene las 5 etapas y 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 método 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**

  El método 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:

  - **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.

  - **EVOLUTIVO**: es un método parcialmente construido que puede pasar de ser prototipo a ser producto, 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 un producto de 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 método 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 y una nueva 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

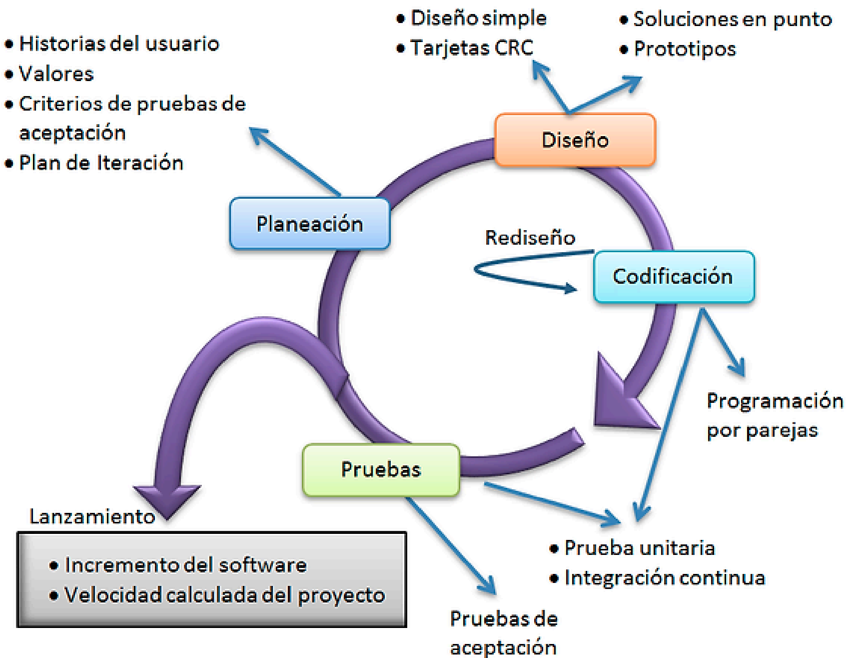
Si bien cada método representa una evolución del anterior, sus desarrollos **siguen dependiendo de una especificación funcional inicial**, característica que sigue sosteniendo a este grupo de métodos como parte de la metodología tradicional.

Métodos ágiles

- **RAD (Rapid Application Development):** creado por **James Martin en 1991**. Presentado como el desarrollo rápido de productos de software que permite construir sistemas en poco tiempo. Se puede decir que es el primer acercamiento a lo que hoy conocemos como metodología ágil, aunque conserva el método de Prototipado, al cual se le incorporaron las herramientas CASE que lo convierten en un método interactivo**.**
- **RUP (Rational Unified Process / Proceso Unificado de Rational):** es el método formal y oficial del “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 de la empresa IBM: RUP que se vende como producto independiente. RUP no es un sistema con pasos firmemente establecidos, sino un conjunto de métodos adaptables al contexto y necesidades de cada organización. Este método comprende tres principios claves, fusión del concepto iterativo e incremental con el espiral, está 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 empresas "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).

Aquí un gráfico con el flujo de trabajo propuesto por este XP:



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

KANBAN: creado en Toyota (Japón) a **inicios de los 50'** y adaptado al desarrollo del software a **principios del 2000**. Es un método 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 que 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 productos al mismo tiempo que maximiza la creación de valor para el cliente. "Lean" significa magro. Es por ello que utiliza la mínima cantidad de recursos, es decir, los estrictamente necesarios para el crecimiento, eliminando 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: la primera versión formal fue presentada por **Ken Schwaber y Jeff Sutherland** en una conferencia en **1995**. 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 método, sólo se enfoca en la gestión de proyectos bajo la metodología 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 enfoque iterativo e incremental para la organización de las funcionalidades del producto aprovechando las prácticas de **XP** para su desarrollo, la gestión de las tareas del 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.

UNIDAD 5: MÉTODOS DE GESTIÓN

Como hemos visto, el método Cascada fue el método de gestión original de la metodología tradicional y el más utilizado, mientras que Scrum fue el último, y por tanto reunió varias prácticas de sus predecesores, siendo el más aplicado en la actualidad.
A continuación profundizaremos sobre las características y prácticas de estos 2 métodos.

Método tradicional: Cascada

Ya hemos mencionado el origen de este método en la unidad anterior. Por lo que en la presente unidad profundizaremos sobre el error de interpretación del artículo presentado por Royce.

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 artículo (1)*).
De esta manera, amén del error, el método de "Cascada" quedó en la historia como el primer método introducido en la ingeniería del software (de hecho Royce ni siquiera lo nombró de esa manera). (1): La historia detrás del error

Así es que Cascada implementa directamente la Metodología Tradicional, motivo por el cual, para este caso hay veces que nos tomamos la licencia de utilizar indistintamente un término u otro como si fuesen sinónimos. Pero como ya hemos visto no lo son.

En la presente sección no necesitamos ampliar mucho más sobre este método, dado que sus características y problemáticas son las heredadas de la metodología tradicional.
Per se, en términos prácticos, el método no propone actividades alternativas a las propuestas por las etapas del ciclo de vida, ni explicita una manera particular de gestionar cada una.

Método ágil: Scrum

Características

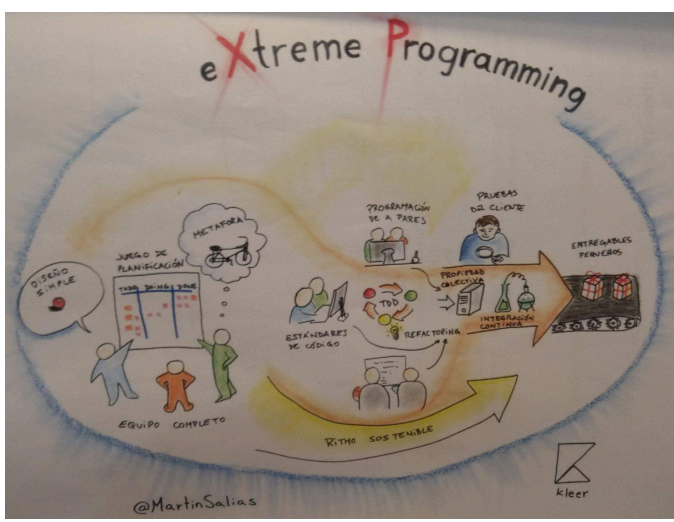
Scrum es un método bastante **simple y liviano**, no es más que un marco (tal como el de una puerta) que brinda una estructura general sin involucrarse en detalles técnicos ni de micro-gestión. De hecho Scrum es un marco **"incompleto"** de manera intencional. Sólo define las partes necesarias para implementar, 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 una mera guía para la mejora continua y la interacción humana, lo que genera como consecuencia la organización del trabajo. Motivo por el cual es sumamente importante ser conscientes de las cualidades humanas que cada persona puede aportar para que el marco sea exitoso.

La oportunidad de mejora que propone Scrum y su simplicidad permite, **a medida que se va necesitando**, combinar procesos, técnicas y prácticas (existentes o definidas por el propio equipo). Se puede decir que Scrum ayuda a las personas, equipos y organizaciones a generar valor e impacto mediante soluciones adaptativas para problemas de cierta complejidad.

Si bien la guía de Scrum no hace referencia a las prácticas técnicas ni de gestión, los equipos de desarrollo optaron por apropiarse de las prácticas técnicas de XP y de las prácticas de gestión de Kanban sin alejarse de la filosofía de Lean, que reduce los desperdicios y se centra en lo esencial.

ASPECTOS TÉCNICOS: dividir los requerimientos para el desarrollo

XP fue el primer método en romper con esta característica de definir todos los requerimientos en la especificación funcional y al inicio del proyecto, generando funcionalidades pequeñas que se puedan desarrollar en iteraciones: historias de usuario (US). A raíz de este cambio surgió la necesidad de ajustar el desarrollo con técnicas apropiadas para funcionalidades pequeñas que eleven su calidad: TDD. El foco de XP no se basa puntualmente en la gestión, sino en las prácticas netamente técnicas que aportan calidad y agilidad a la etapa de desarrollo del ciclo de vida.



GESTIÓN: organización de las tareas

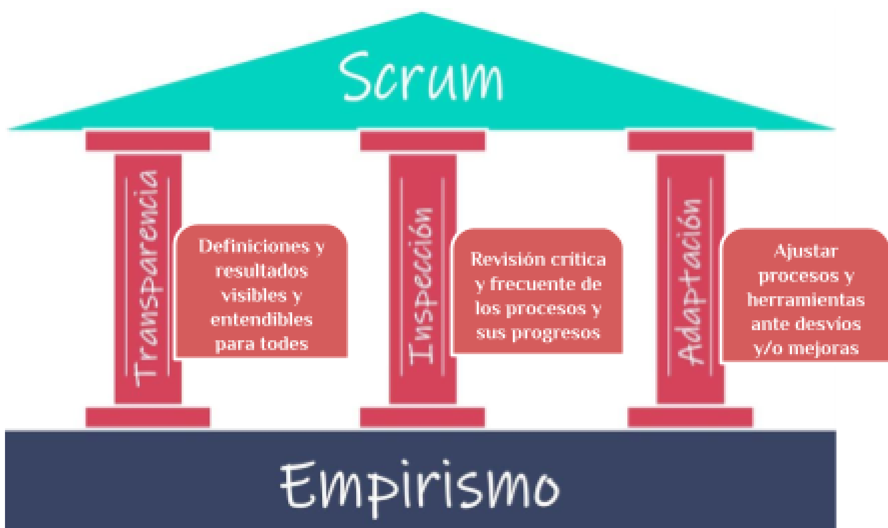
Casi en paralelo surge el método Kanban, el cual sí se focaliza en cuestiones de gestión, para lo cual utiliza tableros que organizan las tareas y visibilizan su estado, aportando calidad y valor incluso a la manera de organizar el trabajo.

Así es que el método Scrum, en la práctica, fue evolucionando más allá de la guía. Para cada iteración, los requerimientos se dividen en historias de usuario que se gestionan mediante un **tablero Kanban**. Y para el desarrollo, se aplican las técnicas **pair-programming**, **TDD, BDD e integración continua** que otorgan calidad al producto en una etapa más temprana.

Scrum se basa en el empirismo, el cual afirma que el conocimiento proviene de la experiencia y que las decisiones se deben tomar sobre la base de los hechos conocidos. Por lo que Scrum genera una propuesta experimental y empírica basada en la prueba y error a partir de la propia experiencia de aplicar el marco una y otra vez en cada iteración, provocando mejoras en el proceso y en el entorno de trabajo.

Para llevar esto a cabo, Scrum cuenta con los siguientes **pilares empíricos y 5 valores**:

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

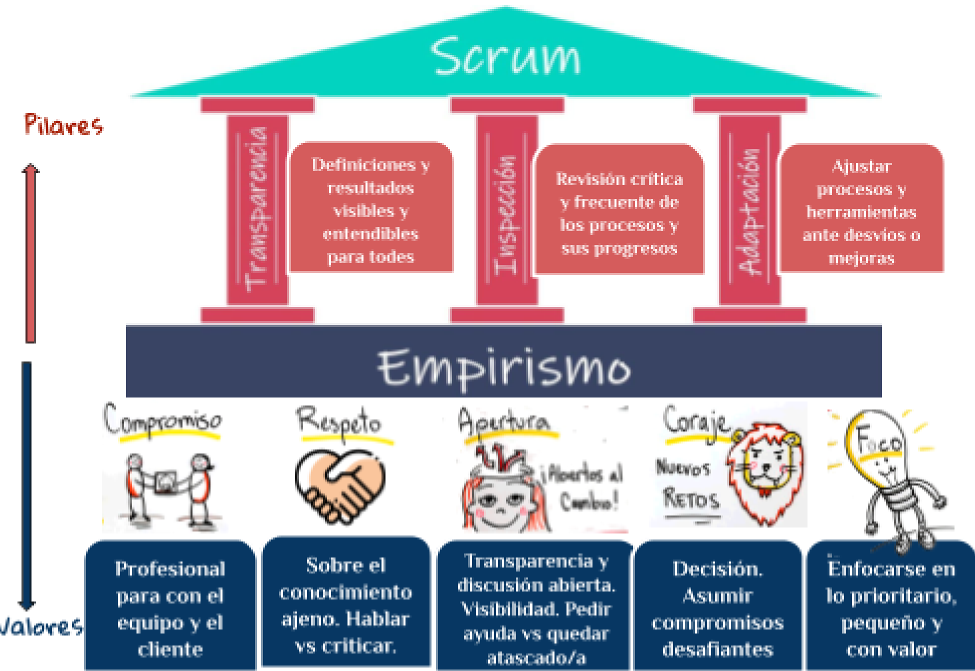
- **Compromiso**: los equipos Scrum tienen mayor control sobre sus actividades, por eso se espera de su parte el compromiso profesional para el logro del éxito.

- **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: gestión de 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. El equipo es quien decide su duración.

¿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 se aconseja que se realice reiteradas veces**, 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 con el período correspondiente, pasando al Product Backlog las incidencias adeudadas para retomar en el próximo sprint, o para cuando el negocio así lo requiera.

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. Si bien es el equipo quien decide su duración en base a su experiencia, es importante tener presente la volatilidad del contexto. Mientras más volátil sea (negocio más cambiante, tecnologías nuevas, etc) más corta será la duración del sprint. Lo importante es recordar que se logra mayor ritmo y previsibilidad teniendo **sprints de duración constante**.

El objetivo del sprint

Cada vez que se inicia un sprint se deberá definir cuál es su objetivo, es decir qué valor se entregará al cliente durante dicho período de tiempo. Durante el mismo, no se deben realizar cambios que lo comprometan, la expectativa de calidad no debe reducirse y el alcance debe ser transparentado y negociado con el cliente a medida que se va dando el aprendizaje.

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 (del sprint y del producto)
  - 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

Los artefactos son las herramientas que permiten al equipo organizar y visibilizar el trabajo **(transparencia)**.

Veamos cada uno de ellos:
- **Product backlog**
- **Sprint backlog**
- **Incremento**

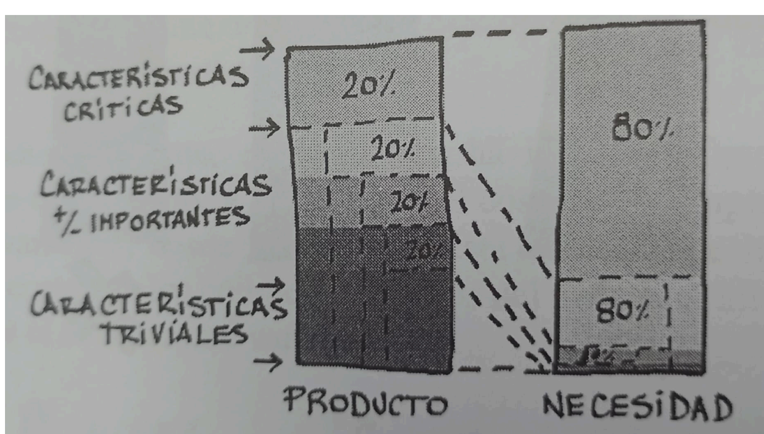
PRODUCT BACKLOG

El Product Backlog (P.B) o pila del producto es el principal artefacto de Scrum y representa el trabajo pendiente. Es decir que el PB es básicamente un listado ordenado de ítems que representan los requerimientos o cambios que forman parte del producto. En su mayoría se encuentran funcionalidades representadas por historias de usuario, pero el PB también contiene otro tipo de ítems. Por tal motivo, es que vamos a clasificar a todos los ítems como incidencias. Así es que el PB contiene incidencias y el orden particular de ellas dentro del PB representará la característica más relevante de este artefacto: la **priorización**. Por lo que es fundamental que el PB se encuentre priorizado.

PB eficiente: ¿cuál es el criterio de priorización?

Para priorizar el PB se utilizará la ley de Pareto o ley del 80/20, donde podemos decir que el 20% de las incidencias del producto resuelven el 80% de las necesidades del negocio que le da origen. Y, de manera recursiva el 20% del 80% restante de las incidencias, resuelve el 80% del 20% restante de negocio.

Podemos representar esta relación recursiva mediante el siguiente gráfico:



Aclaración: en el presente gráfico se denomina *Características* a las incidencias (funcionalidades).

Para poner en práctica la ley de Pareto será necesario tener en cuenta el detalle de información de sus requerimientos, es decir, qué incidencias tiene y el orden que ocupa cada una dentro del mismo. Un PB eficiente cuenta en su tope las incidencias pequeñas y completas más valiosas que se encuentren listas para su desarrollo, y que resuelven el 80% del negocio. Mientras que aquellas que son más grandes, generales, incompletas o poco valiosas se encontrarán en el fondo del mismo, como parte del 20% restante; las cuales se irán refinando en cada sprint y por tanto, escalando hacia el tope del mismo.

Fig. 1

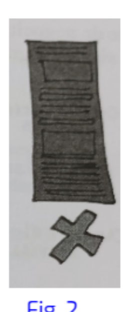


Fig. 2

Fig.3

Fig. 1: Sin priorizar. Todas las incidencias refinadas (pequeñas)

Fig. 2: Sin priorizar. Incidencias mezcladas refinadas (pequeñas) y sin refinar (grandes)

Fig. 3: Priorizado. Incidencias refinadas en el tope (pequeñas) y sin refinar debajo (grandes)

Por último, cabe destacar la importancia de contar con estrategias de priorización que permitan identificar las funcionalidades que aportan valor pero no de manera individual, sino en base a diferentes objetivos a cumplir alineados con las necesidades del negocio.

SPRINT BACKLOG (SB)

Este artefacto contiene el listado de incidencias del PB seleccionadas para trabajar durante un determinado sprint. Básicamente es un subconjunto del PB tomado a partir de su tope y partiendo de la premisa que el mismo se encuentra priorizado, y con ciertos objetivos a cumplir. Si bien el sprint backlog es un artefacto vivo, donde en ciertas ocasiones que ameriten se puede alterar el alcance del mismo, no es una práctica recomendable. Pero en caso de necesitar y con previa negociación será posible que se agreguen y/o modifiquen incidencias durante el sprint.

INCREMENTO

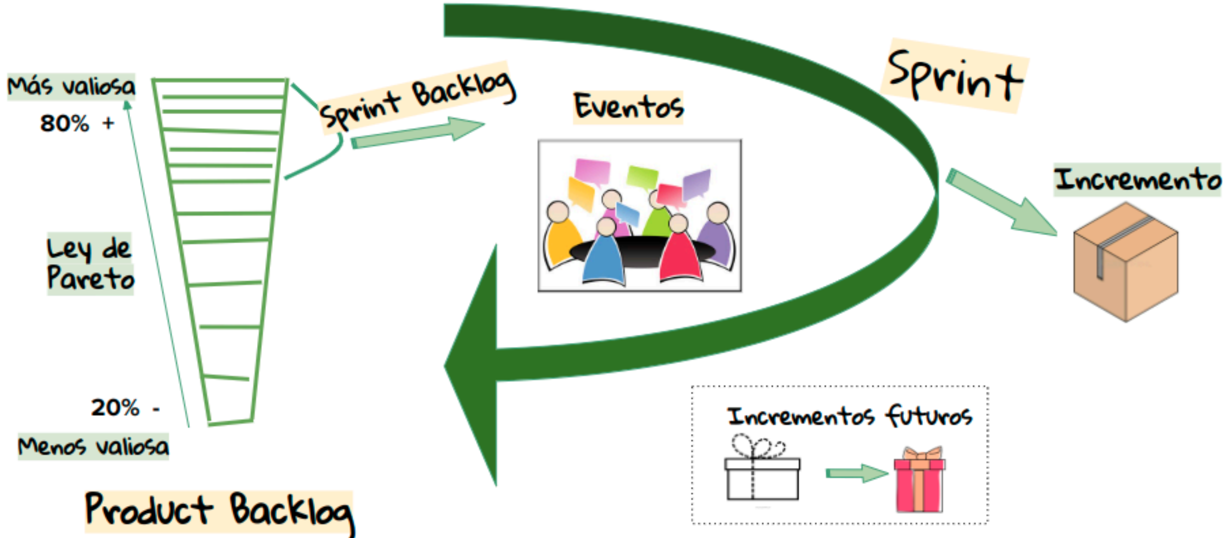
Se denomina incremento al resultado de las funcionalidades completadas durante un sprint. Así es que en cada sprint se irá entregando un nuevo incremento, cuya sumatoria con los ya entregados, comprenderán el producto de software "final".

Resumiendo…

Lo que se desea obtener es una versión del producto de software funcionando con calidad a partir de un conjunto de requerimientos (funcionalidades) a desarrollar en una iteración, a esta versión se lo denomina **Incremento,** y al conjunto de requerimientos a partir del cual se generó, se lo denomina **Sprint Backlog.** El cual contiene una parte del listado general de todas las funcionalidades del producto (incidencias), denominado **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 se muestra un gráfico que muestra la interacción entre los artefactos:



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 el **Sprint Backlog**, el cual se crea al iniciar el sprint.

Luego, durante el sprint, el equipo cumple con todos los eventos de Scrum para gestionar las tareas que se necesitan en cada caso y cumplir con el objetivo. Al finalizar el sprint se entrega un nuevo **incremento**.

Es importante mencionar una característica importante del agilismo que proviene de Lean: *"no producir con demasiada antelación"*. Esto implica que las incidencias en el Product Backlog no necesitan estar definidas con mucho nivel de detalle desde un principio, sino que se van refinando conforme se van sucediendo los sprints. Esto se debe a que los contextos suelen ser muy cambiantes.

RESPONSABILIDADES:

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

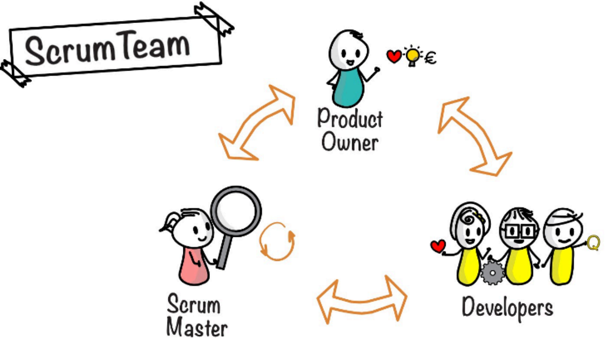
Scrum propone responsabilidades en lugar de roles para evitar acoplar ciertas tareas en una única persona, y evitar por un lado, generar un cuello de botella, y por otro, caer en los roles tradicionales.

¿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



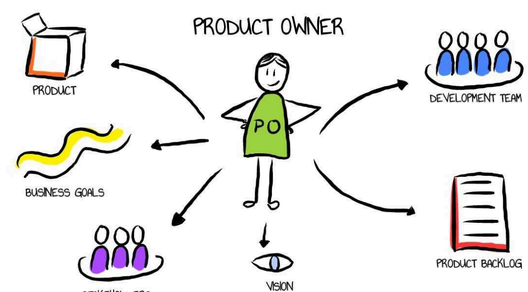
PRODUCT OWNER

El/la Propietaria del Producto (Product Owner - P.O) es la responsable de maximizar el valor del producto resultante del trabajo del equipo Scrum, definiendo las funcionalidades (incidencias) 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 que se apoya en alguien que oficie como tal.

El trabajo del/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.

Actividades más habituales del/la Product Owner además de gestionar el Product Backlog:

- 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:

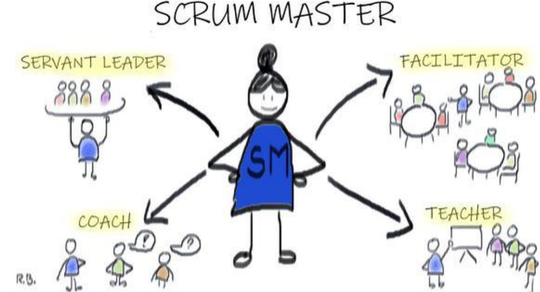
- Ayudar a encontrar técnicas para una definición eficaz del objetivo del producto y la gestión de los retrasos en el mismo
- Ayudar a definir de manera clara y concisa las incidencias del Product backlog según el objetivo del producto
- Facilitar la colaboración con las partes interesadas (stakeholders)

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.

Es importante entender que el/la SM **no debe ejecutar las acciones**, sino **facilitar** todo lo necesario para que **sean ejecutadas por quien corresponda**. Mucho menos involucrarse en las definiciones técnicas.

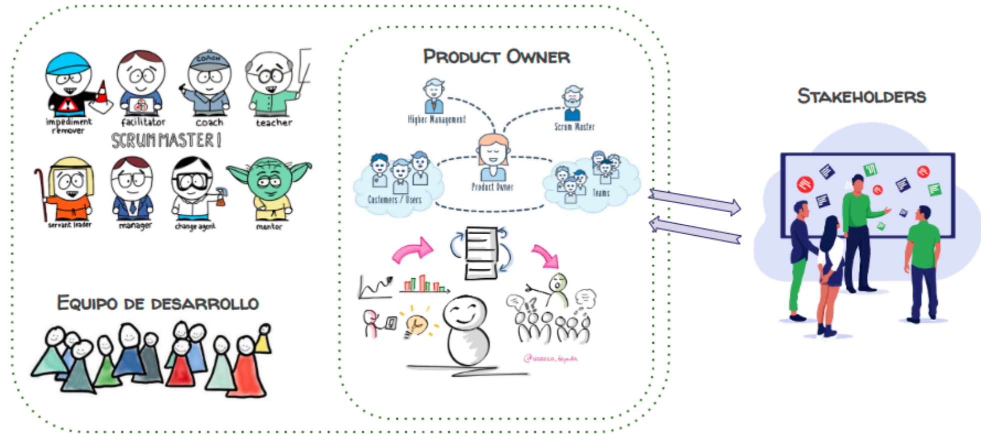
STAKEHOLDERS

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:

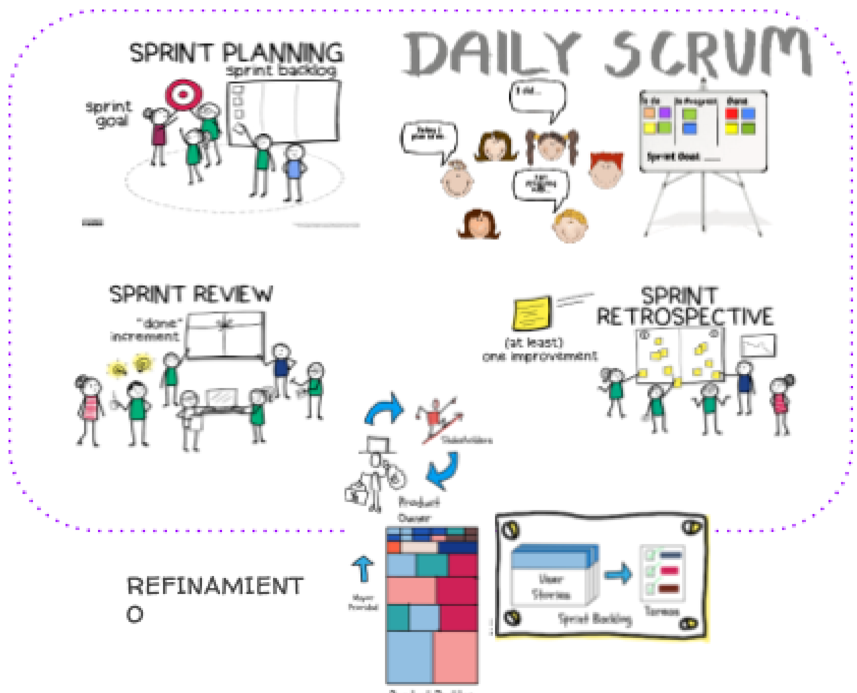
1. El/la Product Owner ordene el trabajo de un problema medianamente complejo en un Product Backlog
2. El equipo Scrum convierta una selección del trabajo en un incremento de valor durante un Sprint
3. El equipo de Scrum y sus partes interesadas (stakeholders) inspeccionen los resultados y realicen los ajustes necesarios para el próximo sprint
4. Repetir

El siguiente gráfico resume 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:



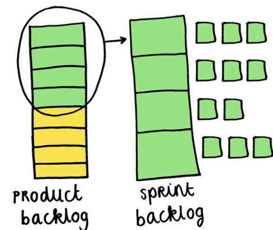
Veamos qué eventos propone Scrum y cuál es el objetivo de cada uno:

SPRINT PLANNING

Es el primer evento que dará inicio al sprint, cuyo **objetivo es planificar el trabajo** a realizar durante dicho sprint para lograr, en cada iteración el objetivo del producto.
Cada Sprint busca sumar su incremento al producto que se va creando.
La planificación del sprint aborda los siguientes pasos:

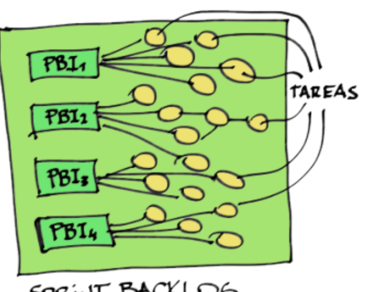
1. **Definir el objetivo del sprint (sprint goal)**, respondiendo a **¿qué valor se va a entregar en este sprint? (Compromiso)**
   El/la PO fue trabajando en este aspecto al mantener el PB refinado. Por lo que en este evento se revisan las funcionalidades del tope del PB y en base a ellas se analiza y define el objetivo. Si cuesta definirlo, es un indicio que el/la PO no utilizó una buena estrategia de priorización. Por lo que habrá que realizar esta tarea en el momento. **Pero se recomienda realizar esta tarea en el refinamiento del sprint anterior.**

2. **Seleccionar las incidencias del sprint**, respondiendo a **¿<u>qué</u> tareas se van a realizar para cumplir con el objetivo?**
   En conjunto con el/la PO se acuerdan las incidencias del product backlog que se seleccionarán en base al objetivo definido previamente.
   Ahora bien ¿cómo sabe el equipo qué incidencias y cuántas se deberán seleccionar?
   Para ello es importante aclarar que cada incidencia se encuentra estimada con puntos de complejidad, por lo que el equipo deberá realizar un "corte" en el PB en base a su capacidad de trabajo, según la velocidad del equipo.



3. **Crear las sub-tareas técnicas** respondiendo a **¿<u>cómo</u> 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 una opción de mantener el trabajo organizado y dar visibilidad del mismo, implica agrupar las incidencias en un **tablero Kanban**, para que el **equipo tenga visible siempre el estado de su trabajo.**
Es importante destacar que el tablero le debe ser útil y servir al equipo, **no a los/as 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)**

Definiciones del sprint Backlog: D.O.R y D.O.D
Siendo que el sprint Backlog se crea a partir del Product Backlog será necesario definir los criterios que le van a permitir al equipo trabajar con las historias de usuario dentro de un sprint.

D.O.R: definición de listo (Compromiso)
También conocida en su versión en inglés como Definition Of Ready (de allí la sigla), y corresponde al conjunto de criterios o características mínimas a partir de las cuales el equipo define que las incidencias están listas para ser incluídas en el sprint backlog. Un ejemplo típico para las US es:

- 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)

Incidencia que no cumple con el D.O.R no puede incluirse en un sprint backlog.

D.O.D: definición de hecho (Compromiso)

También conocida en su versión en inglés como Definition Of Done (de allí la sigla), corresponde al conjunto de criterios o características mínimas 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.

Ejemplo típico:

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

Incidencia que no cumple con el D.O.D no forma parte del incremento a revisar con el cliente. Estos criterios se definen en la Sprint Planning, y es muy probable que el equipo los defina en el primer sprint, y que en cada planning revise si amerita realizar modificaciones o ajustes para el sprint actual, en caso contrario se tienen en cuenta los criterios ya definidos.
Ambas definiciones deben estar consensuadas y visibles para todo el equipo Scrum.

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?

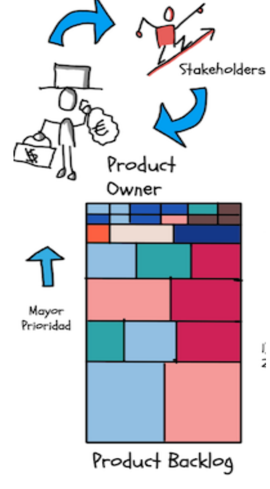
Tener en cuenta el objetivo del evento **y no su forma**. Scrum sólo propone una manera de lograrlo, pero no es la única, cada equipo deberá encontrar la manera de comunicar la información que crea necesaria. Como primer ejercicio, es recomendable seguir la propuesta del marco. En este evento participa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

REFINAMIENTO

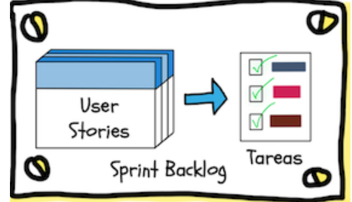
Si bien no es un evento oficial de Scrum, dado que no lo reconoce oficialmente dentro de su marco, se convirtió en una actividad muy necesaria y por lo tanto muy utilizada debido a que permite adelantar el trabajo pendiente.
El refinamiento se realiza sobre el Product Backlog, por lo que es responsabilidad del/la PO y tiene como principal objetivo **adelantar trabajo de cara al próximo sprint**. El refinamiento suele utilizarse desde 2 enfoques diferentes: uno más orientado a lo funcional y otro a lo técnico. Es el Equipo Scrum quien decide cuándo y cómo se realiza esta actividad, y si aplica uno o ambos enfoques. La idea de adelantar trabajo implica profundizar en el entendimiento de las incidencias que se encuentran en el product backlog (más allá del sprint actual) y así dejarlas preparadas para el próximo sprint, según el D.O.R. Siendo que Scrum es un método ágil, gran parte de sus tareas implica mantener el product backlog actualizado según los nuevos requerimientos, los cuales suelen ser muy cambiantes.

Veamos en qué consiste cada enfoque:

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.

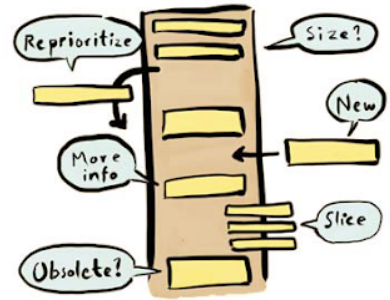


2. **TÉCNICO**: el/la PO se reúne con el equipo de desarrollo para comunicar mejoras y revisar la viabilidad de su desarrollo. Luego, una vez que las incidencias se encuentren refinadas funcionalmente hablando, el equipo de desarrollo deberá **crear las tareas técnicas (sub-tasks)** asociadas a cada 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.



Tareas que 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

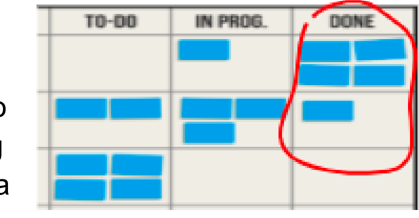


NOTA: claramente no siempre se deben realizar todas estas acciones juntas, sino las necesarias. Por otro lado, es importante destacar que, si bien el/la PO es quien realiza la mayoría de las tareas, el refinamiento sirve como espacio de comunicación entre todas las partes, para que el equipo **pueda 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.

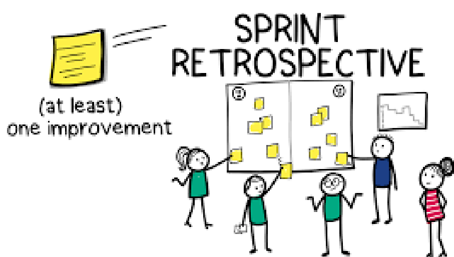


- **REVIEW DEL PRODUCTO:** el equipo Scrum presenta a las partes interesadas los resultados de su trabajo y se discute el progreso hacia el **Objetivo de Producto**. Así, el equipo de desarrollo provee al cliente/PO, el **incremento** según el objetivo planteado en la planning. Las partes interesadas **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.



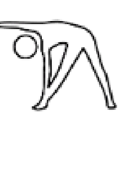
Valiéndose de técnicas de facilitación y análisis de causas raíces, se buscan tanto fortalezas como oportunidades de mejora. Luego, el Equipo Scrum decide por consenso **cuáles serán las acciones** de mejora a llevar a cabo en el siguiente Sprint. Estas acciones y sus impactos se revisarán en la próxima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

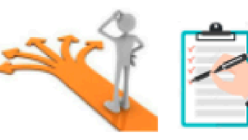
1. Armar el escenario
Es una forma de entrar en calor y conectar con las actividades siguientes



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

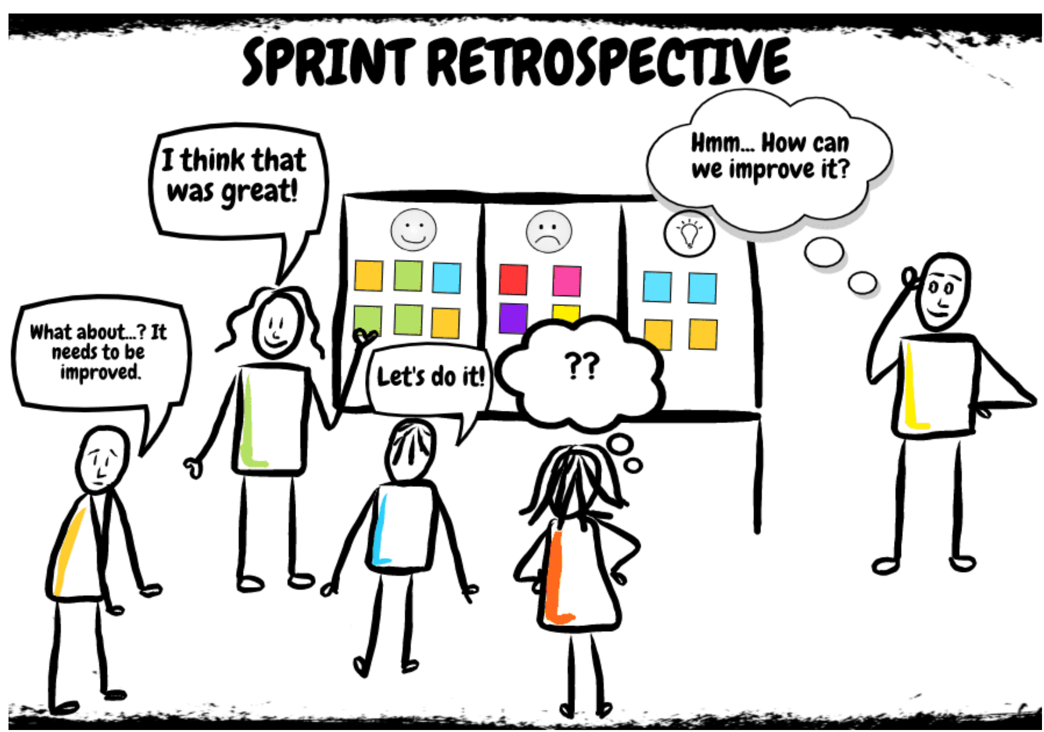
4. Decidir qué hacer
Momento de toma de decisiones. Definir y seleccionar acciones a realizar durante el sprint



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

A continuación se encuentra disponible 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 al/la PO, dependiendo de la confianza generada entre ambas partes 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 que hace referencia a ciertas características. Para que una acción sea SMART implica que deben cumplir con:

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 6: 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

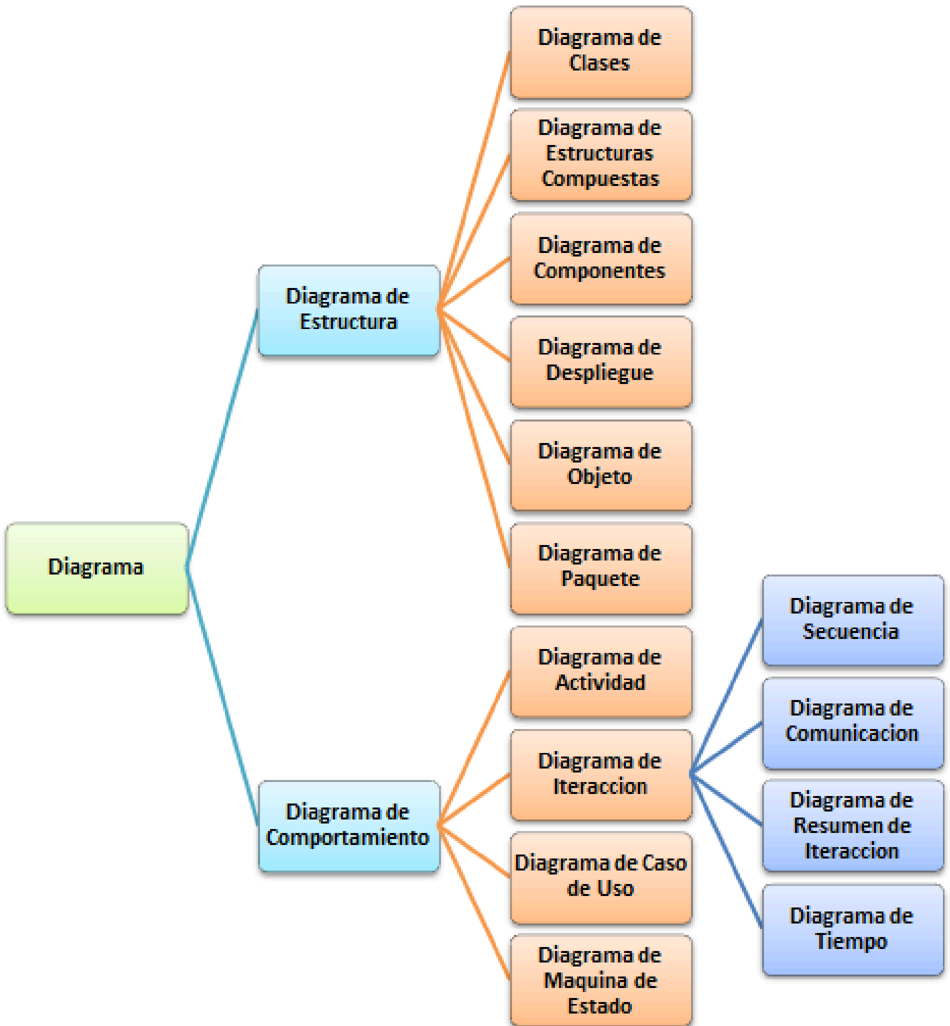
Flujo alternativo:
1. **3a - Usuario:** Número incorrecto
2. **4a_1 - Sistema**: Presenta tono de error y la comunicación termina.

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:

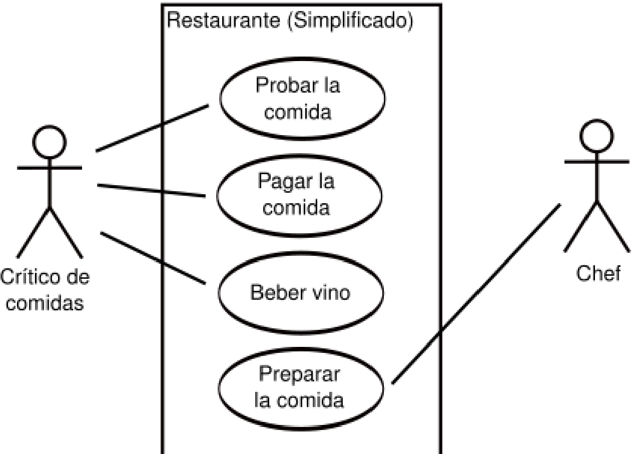
- Actores
- Relaciones
    - Generalización
    - Especialización
- Casos de usos

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:



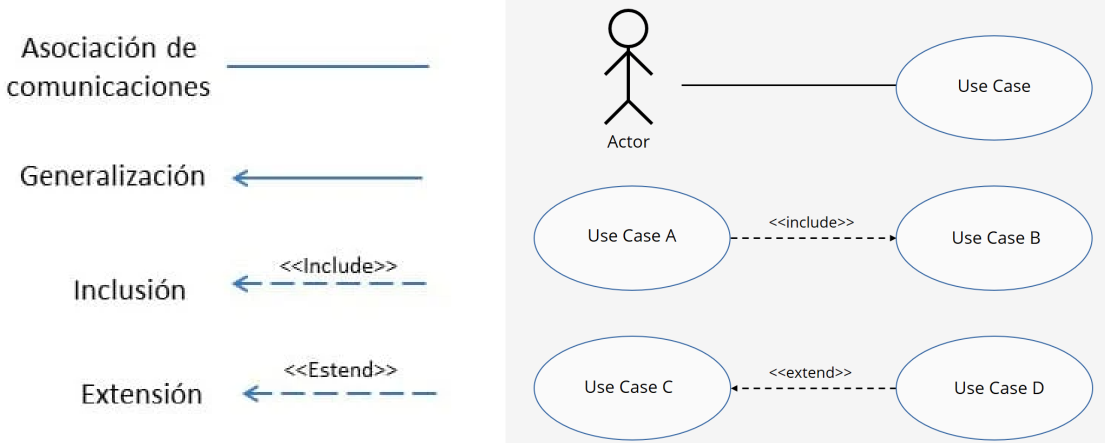
TIPOS DE RELACIONES

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.

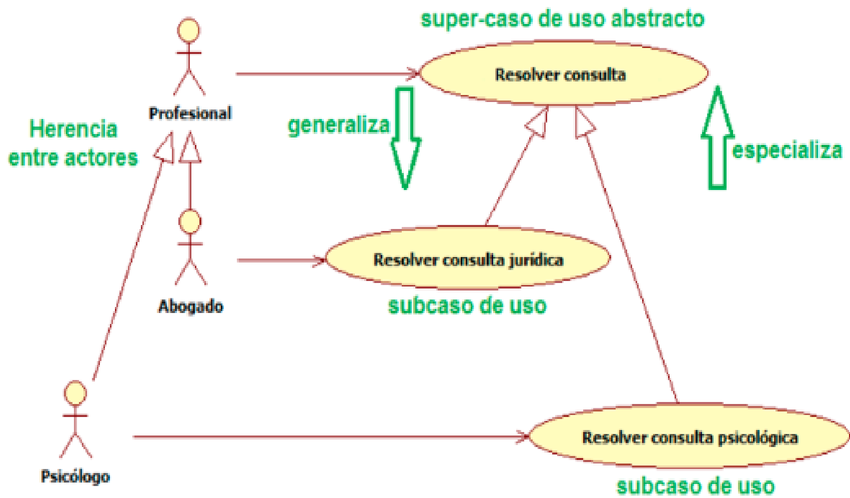
Veamos su simbología:



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

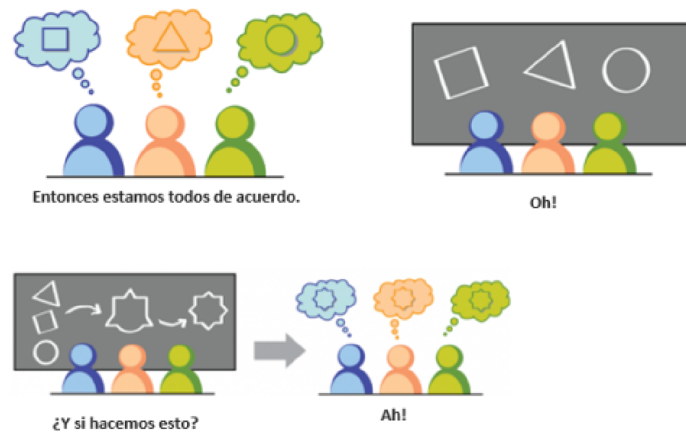


Scrum: Product Backlog

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 forman parte del producto y cuáles no), definir prioridades (tiempo, costo, calidad), 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 este último punto quede documentado 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)

Elevator's Pitch
Como primera actividad se recomienda realizar la técnica **Elevator pitch**, la cual tiene como foco principal definir la **visión del producto u objetivo del producto.** 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 "venderla" en tan sólo pocos minutos y que capte la atención generando impacto en el/la posible inversor/a.

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)

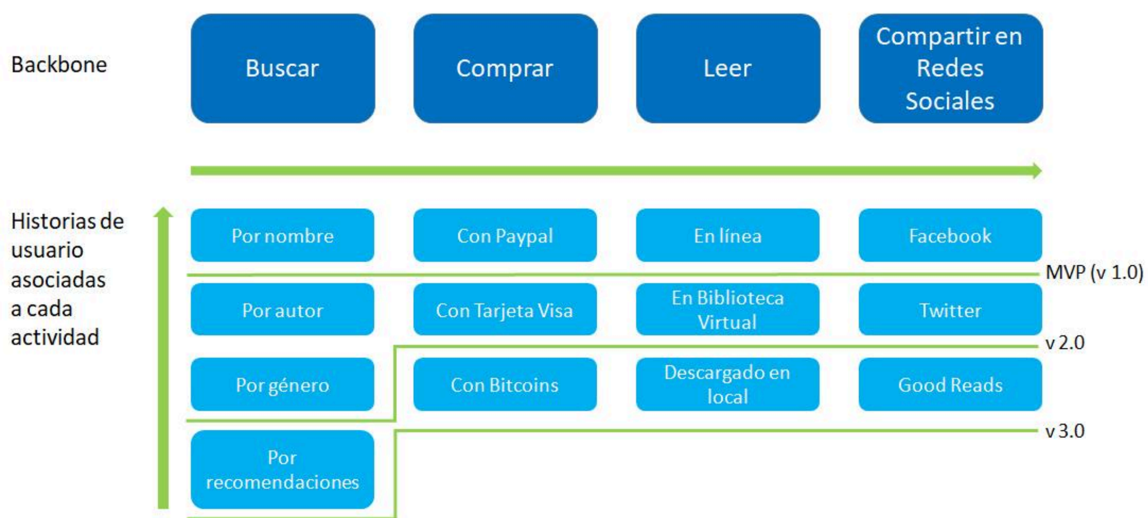
El User Story Mapping es una técnica creada por Jeff Patton que permite ordenar visualmente las funcionalidades de un producto en formato de mapa bidimensional. Primero se detallan las funcionalidades generales (actividades) como columnas, denominadas **Backbone** (columna vertical) formando así el customer 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 comprenden dicha funcionalidad del backbone. Éstas deberán estar priorizadas según el valor que representen al negocio, de manera decreciente, quedando en el tope aquellas que aportan más valor.

La técnica consiste en plasmar, cual brainstorming, todas las funcionalidades posibles del producto, sin restricciones ni profundos análisis para fomentar la creatividad y captar todas las ideas diversas. Luego, como segundo paso, se realiza una revisión de las propuestas para poder organizarlas, descartando aquellas que no son viables o que no aportan valor. Como último paso se priorizan, para finalmente diagramar las diferentes versiones del producto.
La primera versión tiene un significado particular, dado que será la versión con las funcionalidades mínimas que le "darán vida" al producto por primera vez y sobre la cual se sentarán los cimientos para el resto. Por este motivo, y teniendo en cuenta el enfoque ágil, esta primera versión representa el **producto mínimo viable**, identificada como **MVP** por su sigla en inglés. Así se genera valor desde la primera versión del producto.

MVP (Minimum Viable Product)

El MVP representa la versión 1.0 del producto, dado que define el **Producto Mínimo Viable**. Siendo que 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. Para conocer cuál es el MVP se deberá identificar dentro del USM cuáles son aquellas funcionalidades que integran esta versión.



Esta es la última actividad que se realiza en el evento inception. Con esta documentación, el/la PO ya estará en condiciones de armar el **Product Backlog** para comenzar con el desarrollo ágil. En caso de necesitar, con el product backlog creado y la organización provista por el USM, se podrá realizar una estimación, a muy alto nivel. La aproximación de cuán certera será la estimación dependerá de la madurez del equipo. Para un equipo nuevo, que se arma desde cero, la estimación será menos precisa que para un equipo ya armado, estable y con una velocidad conocida.

Incidencias: documentación de iteraciones

Para crear el **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 en inglés).

TIPOS DE INCIDENCIAS: Items Product Backlog

- Épica (epic)
- Historia de usuario (user story)
    - Sub-task (sub-tarea)
- Task (tarea)
- Error (bug)
- Spike

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

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

UX y Mockups gráficos

Los criterios de aceptación como se acaba de mencionar, se pueden detallar tanto por escrito, como en formato gráfico, lo cual aporta un enfoque visual que enriquece la información. A este formato se lo denomina **mockup.**
Un mockup no es nada más ni nada menos que un boceto con información visual sobre la pantalla y sobre la manera de visualizar la funcionalidad. Si bien los mockups suelen ser creados por el equipo de diseño, en caso de no contar con dicho rol, lo puede crear el/la PO incluso sea de manera manual. La visualización gráfica de la pantalla esclarece mucho el entendimiento de la necesidad.

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:** identificador único de la US
- **Título**: representa un breve resumen de la descripción, por lo cual debe ser un verbo que describa la funcionalidad
- **Descripción**: información de la card con su respectivo formato (como/necesito/para)
- **Criterios de aceptación**: obtenidos en la etapa de la confirmación
- **Adjuntos**: para subir el/los mockups
- **Estimación** (en puntos de historia): para especificar la complejidad de desarrollar (programación y testing) la US

Características de una US: INVEST

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 eficiente debe cumplir con el acrónimo **INVEST**:

Independiente **N**egociable **V**aliosa **E**stimable **S**mall (pequeña) **T**esteable

- **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.

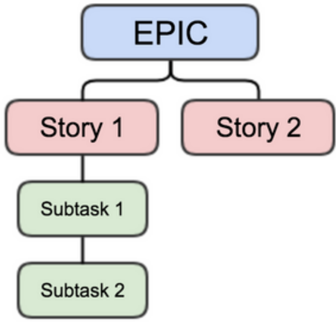
Tener en cuenta que el/la PO puede crear directamente una historia de usuario eficiente en base a la información relevada y analizada, pero muchas veces parte de una funcionalidad más grande, general o incompleta como lo es la épica. Por lo que, a partir de ésta, deberá crear sus correspondientes historias de usuario. Esta tarea suele ser bastante compleja de manera que es importante contar con una técnica que permita dividir dicha épica en historias de usuario. A esta técnica se la conoce como **slicing de US** (rebanar US). Para no entorpecer la lectura del resto de las incidencias, la explicación de esta técnica se dejará para más adelante.

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**

TASK (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 luego 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**.

SPIKE

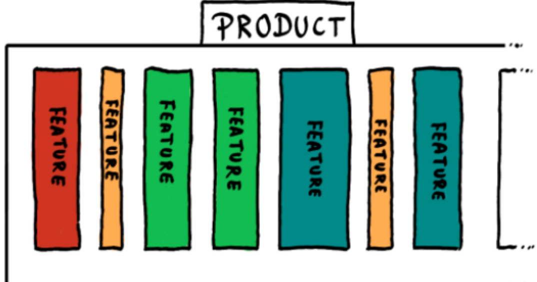
Esta incidencia se define cuando se necesita realizar una investigación técnica. La estructura consta de un **ID**, un **título** y una **descripción**. La cual debe expresar la tarea de investigación junto con la definición de done correspondiente (output esperado como finalización de la investigación).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

THEME (tema)

Si bien el theme o tema **no es un tipo de incidencia**, es importante mencionarlo como concepto. El tema es 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).

Técnicas de 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:

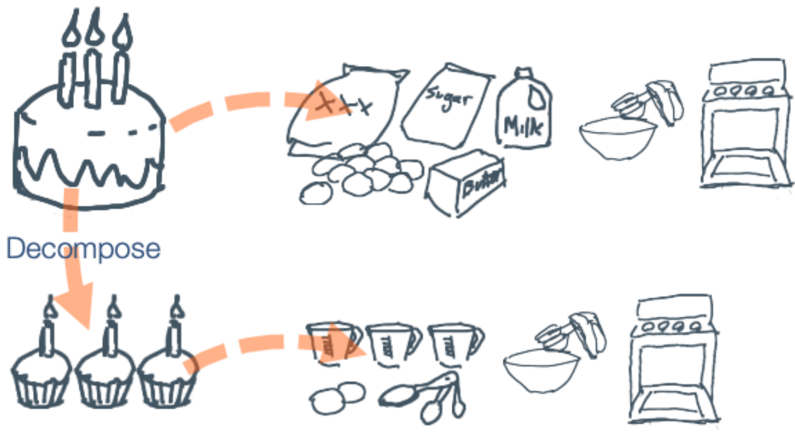


En esta metodología debemos entender que no existe final, salvo que retiremos el producto del mercado o decidamos no evolucionarlo más. Esta diferencia es vital, no estamos ante proyectos con fecha de inicio y fin, sino productos que se van incrementando con el tiempo en función de las necesidades que 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.

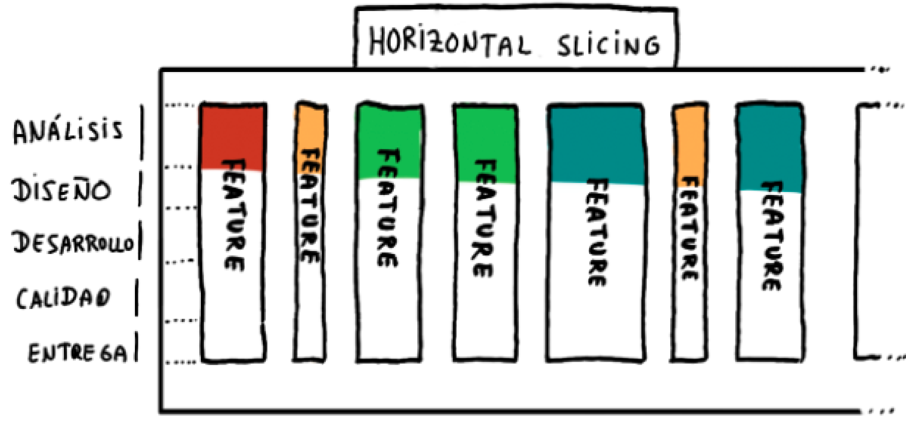
Como ya hemos mencionado, las épicas son quienes se van a “dividir” en US, pero antes de ahondar en la técnica adecuada, les dejo una breve aclaración.
Notar que el término división se encuentra entre comillas, esto se debe a que si bien la traducción literal del término conlleva a dicha palabra, conceptualmente 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**:

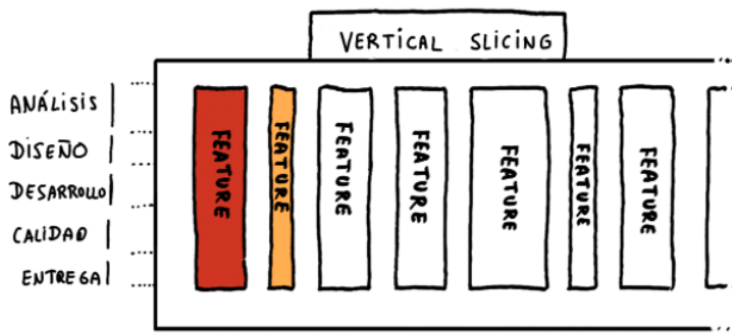


HS: descomposición por capas (Horizontal Slicing)
Cuando se plantea un cambio de paradigma de Cascada a Agile, muchos equipos siguen descomponiendo el trabajo de manera horizontal (Horizontal Slicing), planteando las tareas según las etapas de ciclo de vida del software.



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).

VS: 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: sitios 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 un backlog de historias INVEST.
Bonus de US: aquí el sitio con el resumen de las distintas características que puede contener una US.

Diferencia entre Requisito, CU y US

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

- Los **requisitos del sistema** están escritos desde la perspectiva del producto entendido como parte del sistema y no desde la interacción del usuario con el producto (teniendo en cuenta todo el sistema). Representan las funcionalidades generales en estado puro. Los requisitos del sistema forman parte de la especificación funcional en un desarrollo con el método cascada.
- Los **casos de uso (CU)** están escritos como una serie de interacciones entre el usuario y el producto. Hacen hincapié en el contexto de cara al usuario, es decir a las funcionalidades que utiliza cada usuario identificado en el sistema. Representan la manera de capturar los requisitos del producto desde el punto de vista del usuario. Al igual que los requisitos del sistema, los casos de uso también se incluyen en la especificación funcional de un desarrollo con cascada.
- Las **historias de usuario (US)** se utilizan para describir lo que el usuario necesita realizar y se centran en el valor que genera el uso del producto, en vez de una especificación detallada de lo que el producto debería realizar. Además se conciben como un medio que fomenta la colaboración entre todas las partes involucradas. Las US forman parte del product backlog de un desarrollo con Scrum.

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

UNIDAD 7: CALIDAD

Concepto de calidad

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 lo que sí podemos garantizar es un producto con “la mayor calidad posible” o mejor dicho, con una calidad “*aceptable*”. Podemos detectar que esta expresión resulta un poco ambigua, motivo por el cual, en la presente unidad conoceremos algunas técnicas y prácticas que nos serán útiles para acercarnos un poco más a este objetivo tan ambicioso.

Antes de avanzar será necesario aclarar “una pequeña-gran” diferencia:

Se considera que un producto de software con calidad debe *ser correcto (corrección) y funcionar correctamente (correctitud)*. Pareciera ser que estamos hablando de lo mismo pero hay una sutil diferencia y bastante implícita en la semántica de cada expresión.

Corrección: ¿cómo reconocer cuando un producto es correcto?

Un producto es correcto básicamente cuando satisface la necesidad del cliente, es decir cuando cumple con sus expectativas, y por tanto, con los requerimientos documentados (ya sea en una especificación funcional o mediante un product backlog). Por el contrario, no será correcto si el feedback del cliente termina en la frase *“No es lo que pedí”***.** En este caso es muy probable que los requerimientos no hayan sido documentados según sus necesidades. Este escenario dependerá en gran parte de la metodología de trabajo empleada.

Correctitud: ¿cuándo un producto funciona correctamente?

Un producto funciona correctamente cuando respeta y coincide con los requerimientos documentados con la mayor calidad posible, es decir, “sin errores”. Pero como un producto 100% libre de errores no es factible, el nivel de calidad “aceptable” dependerá de los acuerdos entre todas las partes. De esta manera tanto el cliente como el equipo deberán negociar y definir la tasa o grado de calidad que se considera aceptable al entregar el producto.
Por el contrario, cuando no se llega a cierto grado de calidad debido a la cantidad de defectos encontrados o la gravedad de los mismos, se dice que el producto no funciona correctamente.

Habiendo aclarado esto, será necesario que al momento de construir un producto de software debemos asegurar tanto la **corrección** como **correctitud** del mismo al momento de su entrega. Dicho en otras palabras, desarrollar el producto que el cliente realmente solicitó y que cumpla con la tasa de calidad acordada.

Diferencia entre Error, falla y defecto:

Antes de continuar será necesario diferenciar sobre ciertos términos que suelen utilizarse como sinónimos pero ciertamente no lo son.

Es muy común utilizar indistintamente los términos **error, falla y defecto (bug)**, pero hay una sutil diferencia entre ellos que no radica en el resultado final sino en la fuente de su origen.

- **Errar es humano**, es decir que las personas somos quienes cometemos errores, no los productos ni los sistemas.
- Lo que **falla** son los **productos** (dentro de un sistema) o el propio sistema a causa de los errores humanos.
- Los defectos son la manera que tenemos los humanos de reportar los errores para poder corregirlos.

Reportar un error implica crear una incidencia de tipo bug (técnicamente denominado así en la jerga informática). Este bug formará parte de lo que se denomina en términos de gestión al “reporte de bugs”. Este reporte es el listado de bugs (errores) a corregir en el producto.

Clasificación de pruebas

En las distintas unidades hemos realizando mucho hincapié en el producto desde una visión funcional, motivo por el cual las pruebas se alinearán con dicho atributo.

Hasta ahora hemos mencionado las pruebas como parte de un producto dentro de su etapa correspondiente del ciclo de vida. Pero en esta sección ampliaremos el concepto de prueba al término de calidad de una manera más genérica, presentando las diferentes pruebas que se pueden realizar en un sistema informático, más allá de los productos que lo integran.

Tal como hemos mencionado en la primera unidad, **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 el sistema funcione, debemos contemplar y proveer distintos aspectos de calidad.

Así es que contamos con pruebas clasificadas en diferentes tipos:

- Pruebas Funcionales
  - Manuales
  - Automatizadas
- Pruebas no funcionales

A continuación veremos en qué consiste cada tipo de prueba. Y si bien la clasificación de las pruebas son parte de la ingeniería del software y no se asocian directamente a una metodología de trabajo, se decidió incluir un apartado que explica la forma de implementarlas y gestionarlas en cada metodología.

Pruebas funcionales

Tal como su nombre lo indica son aquellas pruebas que se basan exclusivamente en validar las funcionalidades del producto según sus requerimientos funcionales.

Como parte de este tipo de pruebas contamos con:

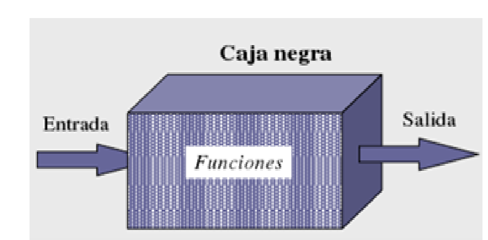
- Pruebas unitarias
  - Caja Negra
  - Caja Blanca
- Pruebas de integración
- Pruebas de humo (Smoke test)
- Pruebas de regresión
- Pruebas de aceptación de usuario (U.A.T)
- Pruebas exploratorias

A su vez, cada una de estas pruebas puede clasificarse según el enfoque **Manual** o **automatizado**. Por ahora veremos en qué consiste cada una conceptualmente y luego profundizaremos 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 Scrum son las que validan el cumplimiento de los criterios de aceptación de las historias de usuario. Mientras que en Cascada son las que validan las funcionalidades de los casos de uso. A su vez estas pruebas, dependiendo de la visibilidad que se tenga de la información se pueden subclasificar en 2 categorías:

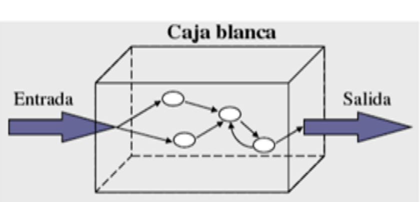
- **De caja negra:** validan la unidad funcional basándose exclusivamente en la interfaz y su correspondiente funcionalidad, **abstrayéndose de la lógica interna del producto**. Sólo se tienen en cuenta los input (datos de entrada) y sus respectivos outputs (datos de salida).

  La ejecución de estas pruebas implica la interacción del usuario con la interfaz del producto y como un producto se compone de muchas funcionalidades es necesario mantener un registro de todas las pruebas a realizar. Es por ello que se necesita diseñar casos de pruebas que faciliten esta tarea. Por este motivo, este tipo de pruebas tienden a ejecutarse de manera manual.
  Estas pruebas se realizan tanto en el método Cascada como en Scrum. Y por tal motivo merecen un apartado específico: ver Pruebas de caja negra

  “El objetivo consiste en interactuar con la interfaz para probar su funcionalidad, asegurando que las entradas y las salidas del producto cumplan 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 interna** de la funcionalidad. Cada una de estas pruebas validan la unidad funcional del código que hace al producto. Requiere conocer cómo se resuelve el problema en base a las entradas, para así validar el resultado de su salida. Por ello es que en este caso, las validaciones suelen ser programadas a diferencia de las pruebas de caja negra, que implican sólo la

interacción entre usuario - interfaz. Así es que quienes trabajan con este tipo de pruebas deben tener cierto conocimiento de programación, por lo menos en el lenguaje de la herramienta utilizada. Dada esta condición es que surge la necesidad de automatizar las pruebas en lugar de ejecutarlas manualmente a diferencia de las pruebas de caja negra.

Relación con las metodologías de trabajo

Las pruebas de **caja negra** se realizan en ambas metodologías. En el caso de la metodología tradicional, puntualmente en Cascada, en la etapa de desarrollo se diseñan todos los casos de prueba del producto completo, para luego en la etapa de prueba ejecutarlas de manera **manual**.

En cambio en el agilismo, al concretar todas las etapas en cada iteración, las pruebas se ejecutan a medida que se van entregando las funcionalidades desarrolladas del producto.
Lo importante es cumplir con este mini ciclo internamente en el equipo de manera fluida en lugar de ejecutar todas las pruebas al final de la iteración, y evitar así caer en un esquema similar al de cascada.
Las pruebas de **caja blanca** por lo general se suelen realizar solamente en la metodología ágil, donde los equipos de desarrollo aportan calidad mediante el uso de la técnica TDD. Técnica central del desarrollo ágil en el método XP, y heredado por Scrum. En paralelo, el equipo de testing, dependiendo de su expertise, también suelen realizar pruebas de caja blanca de manera **automatizada** a diferencia de las de caja negra. Las características de este tipo de prueba no se adecúan demasiado con el enfoque de la metodología tradicional.

PRUEBAS DE INTEGRACIÓN

Estas pruebas apuntan a probar la unión de las partes. Con las pruebas unitarias se prueba cada funcionalidad de manera independiente y atómica pero, ¿qué sucede al integrar varias funcionalidades?
En un producto chico estas pruebas pasan casi desapercibidas siendo que se pueden ejecutar sencillamente y sin muchas demoras, pero en productos grandes ya no es tan trivial. Realmente es necesario contar con tiempo para validar el producto como un todo. Así es surge la necesidad de contar con pruebas que validen la correctitud y corrección del producto de manera integral.
Las pruebas de integración pertenecen a la categoría de pruebas funcionales de caja negra, en la cual el equipo de testing se dedica a probar el producto como tal, focalizando en la interacción de sus partes, validando las reglas de negocio y la consistencia de los datos a lo largo de todo el producto.

Relación con las metodologías de trabajo

Las pruebas de integración son muy necesarias en la metodología tradicional siendo que se cuenta con el producto completo, de manera que es realmente necesario validar la corrección y correctitud del producto que garantice cierto nivel de calidad.

En cuanto al agilismo estas pruebas se van realizando implícitamente en cada iteración, luego de las pruebas unitarias, y conforme crece el producto con cada incremento. Así es que se terminan realizando casi naturalmente al aplicar la técnica de integración continua (IC).

PRUEBAS DE HUMO (SMOKE TEST)

Hemos mencionado previamente que en las pruebas de integración se debe validar el producto completo, y que su principal desafío está dado por los productos más complejos o muy grandes. Por tal motivo, nos vamos a focalizar en dichos contextos para comprender el valor de este tipo de pruebas.

A partir de este contexto es que nos surgen las siguientes preguntas: ¿cómo probar el producto completo de manera integral? ¿por dónde comenzar? ¿tenemos el tiempo necesario para ello?.

La variable fundamental que generó la necesidad de contar con este tipo de pruebas es básicamente el tan preciado **"Tiempo"**. En muchos casos, lamentablemente, no se dispone del tiempo necesario para realizar las pruebas de integración en profundidad. De esta manera es que surgen las pruebas de humo o más conocidas como Smoke test.

A diferencia de las pruebas de integración, en esta ocasión no se busca validar el producto completo de forma exhaustiva, sino realizar una serie de validaciones sobre las **funcionalidades más críticas** que, por lo general, forman parte del **camino feliz**.
El objetivo es realizar un **repaso rápido** de lo más **crítico** 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 entregar el producto. Este tipo de pruebas se ejecutan en última instancia, son el último recurso que nos permite asegurar cierta calidad del producto antes de su entrega.
El nombre hace alusión a las pruebas rudimentarias en ingeniería electrónica, sobre los equipos electrónicos en los que se comprueba que el encendido de un circuito no causa humo ni chispas.

Relación con las metodologías de trabajo
Este tipo de pruebas son muy utilizadas en proyectos ágiles. Al finalizar cada iteración se ejecuta el conjunto de pruebas que componen el camino feliz (pruebas de smoke) para asegurar la calidad y salubridad del producto en el momento previo a su revisión con el cliente.
Dadas sus características no son muy propicias para proyectos con metodología tradicional.

PRUEBAS DE REGRESIÓN
Como parte de las pruebas de integración es muy común notar que al agregar funcionalidad nueva o realizar modificaciones en el producto se terminan introduciendo errores. Esta situación conlleva a la necesidad de realizar pruebas de regresión.
Básicamente se desea 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 sólo con probar las funcionalidades modificadas o agregadas, también es necesario validar que estos cambios no produzcan efectos colaterales, sobre todo negativos, sobre las 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, generando un subconjunto de pruebas. Siendo que no es posible ejecutar todos los casos de prueba diseñados, por tratarse de un universo tan amplio y sabiendo que el tiempo es finito, será necesario identificar y seleccionar aquellas pruebas que integrarán el plan de regresión.

Las pruebas de regresión pueden incluir:

- Revisión de los procedimientos y recursos que forman parte del sistema para descartar que la causa de los errores provengan de aspectos externos al producto.
- Revisión de los datos (BBDD o repositorio de información correspondientes) para asegurar la integridad de la información.
- La ejecución propiamente de las pruebas que integran el plan de regresión

Relación con las metodologías de trabajo
Estas pruebas se realizan tanto en los proyectos tradicionales como en los ágiles según el método de trabajo utilizado. Por ejemplo en Scrum se ejecutan previamente al evento review. En cambio en cascada se realizan luego de la corrección de errores o ante la solicitud de cambios para prevenir y/o captar los errores generados a partir de dichos cambios.

PRUEBAS DE ACEPTACIÓN DE USUARIO (U.A.T)
Las pruebas de aceptación de usuario o su homólogo en inglés User Acceptance Test (U.A.T), 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 que 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.

Relación con las metodologías de trabajo
Estas pruebas especificadas de esta manera están orientadas a proyectos muy grandes y se utilizan principalmente en la metodología tradicional en una fase aislada. Mientras que en el agilismo se terminan realizando de manera implícita en cada iteración, y en cada integración mediante la técnica de integración continua (IC).

PRUEBAS DE USABILIDAD (UX)
Hasta ahora todas las pruebas mencionadas se utilizan para probar puntualmente las funcionalidades en término de correctitud y corrección pero también es importante analizar cómo es la experiencia del usuario. Este punto hace referencia al aspecto visual y de uso del producto, en el cual se busca indagar cuán fácil e intuitivo es para sus usuarios el recorrido por las funcionalidades del producto.
De estas pruebas se encarga generalmente el equipo de diseño y 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 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 agrada o desagrada al usuario/a utilizar el producto.

Relación con las metodologías de trabajo
Estas pruebas son utilizadas tanto en los proyectos tradicionales como en los ágiles, teniendo mayor preponderancia en este último. La diferencia radica en el momento en que se aplica. Cada uno se aplicará según el funcionamiento del método de trabajo aplicado.

PRUEBAS EXPLORATORIAS

Estas pruebas se suelen realizar para complementar el set de pruebas ya diseñadas, las de caja negra, con el objetivo de ampliar la cobertura de calidad y así captar con mayor facilidad bugs más complejos y flujos nuevos que en apariencia no parecer ser tan visibles.
A diferencia del resto de las pruebas, en las exploratorias no se dividen las etapas de diseño de casos de casos de prueba y de ejecución, sino que se realizan ambas tareas en simultáneo donde una alimenta a la otra.
Este tipo prueba 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 diseñan y ejecutan las pruebas con un objetivo concreto. Luego con las conclusiones obtenidas se va aprendiendo sobre el producto, información que será utilizada para seguir iterando, diseñando y ejecutando nuevas pruebas.
Cuando se realiza testing exploratorio no se “prueba por probar”, sino que antes de empezar se deben definir algunas cuestiones:

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 en pos de una mejora continua.

2. **TIEMPO:** será necesario establecer el tiempo de duración de la sesión para “no irnos por las ramas”.

3. **DOCUMENTACIÓN:** a medida que se va probando se van 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, sobre un documento, post-its, alguna herramienta de tracking de incidencias, etc. También es muy útil sumar elementos gráficos 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, básicamente como los/as usuarios/as.
Particularmente para este tipo de pruebas se necesitan las habilidades blandas más características de un/a tester:

El testing exploratorio bien planteado puede dar muy buenos resultados, pero todo depende de las habilidades de quien testea. Debe ser capaz de analizar en qué puntos podría fallar el producto (de acuerdo al objetivo establecido), analizar los riesgos que conlleva etc; todo ello basándose en el conocimiento e intuición que tiene sobre el producto.

Relación con las metodologías de trabajo
Este tipo de pruebas se pueden aplicar perfectamente tanto a proyectos tradicionales como a ágiles. Pero existe una tendencia de uso en los contextos ágiles debido a las características y pasos similares con dicha metodología. Los equipos de trabajo de los entornos tradicionales no se caracterizan por innovar en este tipo de prácticas, sobre todo cuando está en juego el producto completo.

PRUEBAS MANUALES VS AUTOMATIZADAS

Hasta ahora venimos explicando los diferentes tipos de pruebas funcionales, cada una con su objetivo y sus características, pero no hemos mencionado la manera de llevarlas a cabo. Las pruebas de unitarias se pueden llevar a cabo de 2 maneras: **manual o automatizada**.
Las pruebas **manuales** requieren básicamente de la interacción del/la tester con el producto, por tal motivo es que las pruebas de **caja negra** suelen adaptarse bien a este formato ejecutándose, generalmente, de manera manual. Sin embargo también se cuenta con una versión **automatizada**, gracias a herramientas como **Selenium** que permiten agilizar la interacción con la interfaz del producto. Por su parte, las pruebas de **caja blanca** rara vez tienden a ser manuales. Lo más próximo a ello recae en una zona gris, cuando el/la tester inspecciona el código con alguna herramienta afín, consulta algún servicio (api, json, etc) o directamente accede a la base de datos. Esta última acción suele ser de lo más común.
Los productos chicos no tienen un costo tan alto por contar con pruebas manuales, donde “la mirada de un ser humano para pensar como otro ser humano es sumamente útil”. Sobre todo es muy beneficiosa cuando se necesita encontrar bugs muy ocultos o sobre flujos muy complejos. Pero este tipo de pruebas manuales resultan bastante costosas para productos muy grandes, y poco beneficiosas para las pruebas de rutina, como las del camino feliz, donde las funcionalidades mínimas y críticas suelen ser una tarea bastante engorrosa y aburrida de llevar. Por tanto, para este tipo de situaciones se suele recurrir a la **automatización** de las pruebas ya diseñadas. Ésto implica programar los casos de prueba 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, y su subconjunto correspondiente de smoke test. A su vez se suele aprovechar este espacio para sumar pruebas de **caja blanca** que validan la lógica interna del código. Este aspecto se aprovecha muy bien mediante el uso de la técnica TDD, muy utilizado en las metodologías ágiles.

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, y como sabemos, un sistema incluye al producto, y la calidad debe ser transversal al sistema. Así es como 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.

Estas pruebas se subclasifican en:

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

PRUEBAS DE RENDIMIENTO O PERFORMANCE

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.

- **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 alto volúmen de información, como por su tiempo de respuesta al ser consultada.
- **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 para 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.

- **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 peticiones sobre la misma API o muchos usuarios enviando el mismo formulario. Un claro ejemplo de la importancia de este tipo de pruebas se da en muchos casos para ocasiones particulares como el “Black Friday” sobre productos e-commerce, o en fechas de inscripción sobre el SIU Guaraní.
  El comportamiento del sistema, incluido el tiempo de respuesta de la aplicación que perciben los usuarios, dependerá de la concurrencia que exista en un momento determinado sobre el mismo. Por este motivo es importante identificar cuáles son los niveles de concurrencia que encontramos en las pruebas de rendimiento:

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

- **Pruebas de stress:** tal como su nombre lo indica, estas pruebas buscan estresar al sistema, exigiendo 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 duplicando 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.

Relación con la metodología de trabajo
Siendo que las pruebas no funcionales no tienen el foco puesto en el producto, escapan a la metodología de trabajo utilizada para su desarrollo. De esta manera, el tipo de pruebas no funcionales que se realicen es independiente de la gestión del producto. Y por tal motivo, la elección de las pruebas a realizar dependerá del tamaño del proyecto, los recursos económicos, el personal técnico idóneo y la infraestructura con la que se dispone para tal implementación.

Pruebas de caja negra

Las pruebas de caja negra ameritan una sección independiente debido a que son muy utilizadas y con similar importancia para la **metodología tradicional como para el agilismo.** Recordemos que un producto se compone de muchas funcionalidades, motivo por el cual es imprescindible contar con un registro de todas las pruebas a realizar. A este registro se le denomina: **diseño de casos de pruebas**. Los casos de prueba se escriben bajo una determinada estructura. Esta tarea se realiza en la etapa de desarrollo del ciclo de vida del software. Y en la etapa de prueba se llevará a cabo cuando se ejecuten los casos. Recordemos que como parte de esta ejecución, cada caso de prueba puede fallar o ser exitoso. De haber fallado alguno, será necesario reportar con su correspondiente bug al equipo de desarrollo.

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:

- **ID:** identificador de la prueba. No debe haber 2 pruebas con el mismo ID.
- **Título:** el título debe describir la funcionalidad a probar. Así es que debe comenzar con un verbo en infinitivo.
- **Pasos + datos:** en este campo se unifican ambas informaciones. En los **pasos** se debe detallar cada acción a realizar junto con sus respectivos datos. Esta información será necesaria para la ejecución de la prueba.
- **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** se desea 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 explícitamente se espera ver. Hay una sutil diferencia. Veamos un ejemplo.

  Se cuenta con una prueba cuyo título es: Registrar un producto. Veamos un resultado esperado incorrecto y su versión correcta.

  - **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 caso se explica 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 la pantalla**, en el producto y no en la base de datos. Recordemos que es una prueba de caja negra, pero en muchos casos, dependiendo de los conocimientos técnicos del tester, el cual puede implementar una "caja gris", indicando información relevante de la base de datos.
- **Precondiciones:** indica, en caso de haber, las necesidades (condiciones) que se deben tener en cuenta para poder ejecutar la prueba con éxito. Por ejemplo, si el título del caso de prueba es "Modificar un producto", las precondiciones son: tener el producto Yerba registrado (siendo Yerba el utilizado en el campo pasos+datos). Este campo es opcional, dado que no todos los casos tienen precondiciones.
- **Bug_ID:** no es propiamente un campo del caso de prueba, pero de haber fallado la prueba, es necesario vincular el ID del bug reportado con dicha prueba, para mantener una trazabilidad entre ambos.

Clasificación de casos de prueba

Los casos de prueba se clasifican en casos positivos o negativos según cumplen con lo especificado o no en la especificación o US y en función de sus entradas:

- **Casos positivos:** aquellos que sí cumplen con los requerimientos por contar con entradas válidas. Verificando que el producto respete lo especificado, es decir, que sea correcto.
- **Casos negativos:** aquellos que no cumplen con lo especificado en los requerimientos. En general cuentan con entradas inválidas, condiciones inesperadas o no deseadas. Los/as testers crean este tipo de casos simulando usuarios malintencionados o que cometen errores fortuitos, para “captar” dicho error y garantizar que la funcionalidad no realice una acción incorrecta. Se verifica que al usar una entrada no válida, el producto no permita continuar con el flujo como si ésta fuese válida. Son los casos que, como testers, queremos ejecutar para “romper” o hackear la aplicación.

Tener en cuenta que tanto los casos positivos como los negativos son casos de prueba, y como tal, la ejecución de ambos pueden resultar en exitosa o fallida. La diferencia radica en que para esta última situación, se deberá reportar su bug correspondiente. Por lo que es importante, **no confundir un caso de prueba negativo con un bug.** Para evaluar si el caso falló o fue exitoso se deberá comparar el resultado que se espera (campo resultado esperado) con la información observada en pantalla producto de su ejecución.

En conclusión, la diferencia entre los casos positivos y negativos radica en la declaración de las entradas y el comportamiento especificado en base a los requerimientos. También es importante destacar el contexto que da origen a esta práctica. El diseño de casos de prueba surgió como una actividad propia de la etapa de desarrollo del ciclo de vida del software, y por ende heredada por el método tradicional Cascada. En dicho caso las pruebas se deben diseñar en base a los requerimientos plasmados en una especificación funcional, donde el objetivo esencial de ésta refleja la necesidad del cliente, y como tal, la información detallada apunta al “camino feliz” del producto, restando importancia u omitiendo directamente los flujos alternativos, accidentales o maliciosos. Como consecuencia, con dicho método suele aumentar la cantidad de casos de prueba negativos. Mientras que con los métodos ágiles la información detallada es más rica, dado que incluye los flujos alternativos por contar con funcionalidades pequeñas, y por ende, reduce la cantidad de casos negativos en comparación con el método anterior. Lo cual no implica que éstos no se diseñen, sino que en proporción tienen una tasa menor. De todas formas, cabe destacar que con ambos métodos, esta clasificación de pruebas es producto del ingenio y la visión crítica del equipo de testing en pos de incrementar la calidad del producto entregado.

¿Qué es lo que se prueba y qué no se prueba?

Es importante comprender el objetivo de las pruebas de caja negra para, al diseñar las pruebas, ser exhaustivos pero efectivos. Básicamente lo que se requiere como objetivo final y general es entregar un producto que sea “el correcto” y que funcione “correctamente”. Y por tal motivo, es fundamental no perder este foco. Lo que **se prueban** son **funcionalidades** de un producto y **no elementos de una página, ni sus datos de manera aislada**. Éstos forman parte de las pruebas como parte de la funcionalidad. Por ejemplo, si se cuenta con la funcionalidad “Buscar productos”, el título de la prueba deberá ser “Buscar un producto” y en el campo “Pasos + datos” se incluye el producto (dato) con el cual probar. Ciertamente no es correcto que el título sea, por ejemplo, “probar con yerba”. Aquí se perdió el foco de la prueba.

Por empezar, la funcionalidad corresponde a una búsqueda, por lo que dicho verbo debería ser parte del título, y el dato se debe aclarar en su campo correspondiente, desacoplando la funcionalidad de los datos. Más adelante veremos técnicas de diseño de casos de prueba para efectuar esta práctica. Tampoco es correcto: “que aparezca el botón buscar”. Nuevamente no es una funcionalidad, por lo que no se deben crear pruebas para validar elementos de html. Éstos se reúnen en una única prueba que se denomina “Completitud de pantalla” a profundizar en la próxima sección de ejemplos.

Ejemplos

Para los ejemplos a continuación focalizaremos en el campo “Título” y en el “Resultado Esperado” dado que suelen ser complejos de comprender. Para el caso de éste último, el objetivo radica en comparar dicho campo con lo que vemos en la pantalla. Si ambos coinciden, la prueba es exitosa, en caso contrario, si notamos una diferencia (por más mínima que sea) la prueba falló. Por este motivo es muy importante ser minuciosos/as al redactar el resultado esperado. Veamos a continuación varios ejemplos que ponen foco en este aspecto así como también en la clasificación de los casos de prueba.

Ejemplo 1:

Se cuenta con la siguiente historia de usuario:

Título: Registrar un cliente

Descripción: **Como** cliente **necesito** registrarme **para** comprar productos online de manera privada.

CA: se necesita un formulario con los campos Nombre y apellido, fecha de nacimiento, Nacionalidad, DNI, usuario y password, y un botón Aceptar, que al presionarlo registra al cliente y redirige al home mostrando su nombre y apellido en el lateral superior derecho indicando que inició sesión.

a) Caso de prueba positivo

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	

b) Caso de prueba negativo:

ID	Título	Pasos + datos	Resultado Esperado	Precondiciones
3	Registrar un cliente repetido	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**: ante.zane f) **Password**: 22AntoZ4ne 4. Clickear en botón **"Aceptar"**	Se debe mostrar un cartel con el mensaje: ***"Cliente ya registrado. No puede volver a registrarse"***	Que la cliente con DNI 39123987 ya se encuentre registada

Notar que en ambos casos el título hace mención a la funcionalidad que se está probando, donde se incluye en cada caso una particularidad que remarca la diferencia entre ambas pruebas.
Por otro lado, en el segundo caso, se trata de un caso negativo porque la US no menciona explícitamente que no se pueden duplicar los clientes, pero en pos de conservar la consistencia de los datos se debe validar esta situación evitando duplicar la información. El mensaje mostrado se debe acordar con el/la PO (o su homólogo Analista funcional) y quien desarrolla para evitar reportar un bug.

Ejemplo 2:

Veamos ahora un caso de prueba que se encuentra más avanzado en el workflow del producto.

Título: Buscar un producto

Descripción: **Como** cliente **necesito** buscar un producto **para** agregarlo al carrito de compras.

CA: un texto para ingresar el nombre del producto, y un botón "Buscar" que al presionarlo muestre en una lista todos los productos que coincidieron con el producto ingresado. La lista debe mostrar el nombre de los productos.

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

En este caso, en lugar de escribir detalladamente todos los pasos, sólo mencionamos los módulos (funcionalidades generales) 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" dando por hecho que el/la tester sabe llegar hasta allí, como se observa en el resto de los ejemplos
3. Indicando el/los IDs de los casos de prueba (CP) anteriores que hacen al camino para llegar a la funcionalidad que finalmente queremos probar.

Ejemplo 3:

Se cuenta con la siguiente historia de usuario:

Título: Pagar la compra online
Descripción: **Como** cliente **necesito** pagar los productos agregados al carrito de compras **para** efectuar la compra y poder recibirlos.
CA: se debe mostrar un label "Medio de pago" con las opciones: tarjeta de crédito y efectivo. Para efectivo, mostrar el mensaje: **"Se abonará con la entrega"**. Para la tarjeta de crédito, mostrar un form con un texto para ingresar el DNI, un selector con las tarjetas: Visa, Mastercard y American Express, un texto para el nro de tarjeta, un calendario para la fecha de vencimiento y un texto para el código de seguridad, al final un botón "**Pagar**", que al presionarlo debe registrar el pago y mostrar el mensaje: ***"El pago se efectuó correctamente".***

ID	Título	Pasos + datos	Resultado Esperado	Precondiciones
23	Pagar compra con tarjeta de crédito válida	1. Ingresar a pagos 2. Seleccionar Modo de pago: Tarjeta de crédito 3. Completar datos de tarjeta con: - DNI: 32456187 - Tarjeta: Visa - Nro Tarjeta: 2345 3222 1677 9816 - Fecha vto: 28/05/2027 - Cod. seg: 710 4. Clickear botón **Pagar**	Se muestra el mensaje: **"El pago se efectuó correctamente".** **Opciones**: 1) Se debe encontrar en la tabla clientes_pagos un registro con la tarjeta termina den 9816 asociada al cliente 432. 2) Al pegarle al servicio pago/visa debe traer la tarjeta termianda en 9816 asociada el cliente 432.	Contar al menos un producto en el carrito de compras. Que exista el cliente 432 asociado al DNI 32456187
24	Pagar compra con tarjeta de crédito inválida	1. Ingresar a pagos 2. Seleccionar Modo de pago: Tarjeta de crédito 3. Completar datos de tarjeta con: - DNI: 32456187 - Tarjeta: Visa - Nro Tarjeta: 1111 3333 0000 2222 - Fecha vto: 28/05/2027 - Cod. seg: 710 4. Clickear botón **Pagar**	Se muestra el mensaje: **"Los datos de la tarjeta no son válidos".** **Opciones**: 1) No se debe encontrar en la tabla **clientes_pagos** un registro con la tarjeta termina den 222 asociada al cliente 432. 2) Al pegarle al servicio **pago/visa** no debe traer la tarjeta termianda en 2222 asociada el cliente 432.	Contar al menos un producto en el carrito de compras. Que exista el cliente 432 asociado al DNI 32456187

- **Caso positivo:** el caso 23 es un caso positivo porque cumple exactamente con lo especificado en la US, tanto en término de datos como de mensaje. Este es el caso "ideal" dado que valida la situación feliz y elemental.
- **Caso negativo:** el caso 24 en cambio es negativo porque en la US no se especifica qué debe suceder si se intenta pagar con una tarjeta inválida. Ni siquiera solicita la validación de la misma. Pero como testers debemos asegurarnos que no se permita pagar con una tarjeta inválida, ya sea intencional o accidentalmente. Lo que clasifica al caso de prueba como negativo por realizar una acción no solicitada y con datos inválidos (no deseados). En cuanto al mensaje mostrado se debe consultar con el/la PO y coordinar con quien desarrolló la US. De la misma forma con su análogo en la especificación funcional para cascada.

Claramente faltan las pruebas para el pago en efectivo.

b) Ahora supongamos una nueva versión de la US anterior, donde los CA contienen más información. Ver texto en verde:

> **Título**: Pagar la compra online
>
> **Descripción**: **Como** cliente **necesito** pagar los productos agregados al carrito de compras **para** efectuar la compra y poder recibirlos.
>
> **CA**: se debe mostrar un label "Medio de pago" con las opciones: tarjeta de crédito y efectivo. Para efectivo, mostrar el mensaje: **"Se abonará con la entrega"**. Para la tarjeta de crédito, mostrar un form con un texto para ingresar el DNI, un selector con las tarjetas: Visa, Mastercard y American Express, un texto para el nro de tarjeta, un calendario para la fecha de vencimiento y un texto para el código de seguridad, al final un botón "**Pagar**" que al presionarlo, **primero valide que los datos de la tarjeta sean correctos, y en cuyo caso registrar el pago y mostrar el mensaje: "El pago se efectuó correctamente". En caso contrario no se debe registrar el pago y se debe mostrar el mensaje: *"Los datos de la tarjeta no son válidos".***

Con esta nueva US, el caso de prueba 24 se clasifica como positivo por más que sus datos sean inválidos, dado que esto fue especificado en la US.

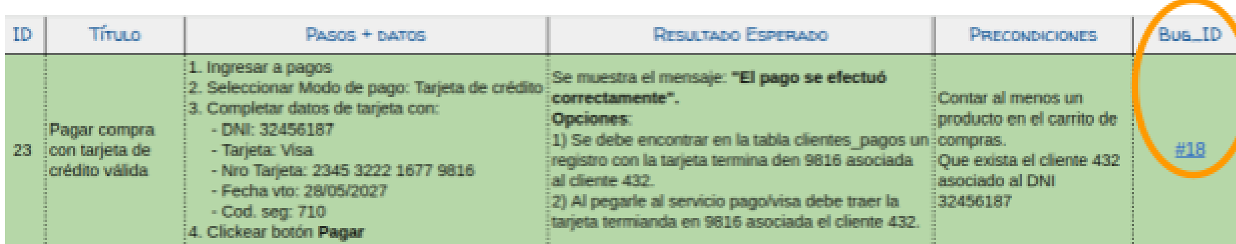
Observaciones:

- Notar que para indicar la clasificación de cada caso de prueba, marcamos en verde el caso positivo y en rojo el negativo. 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 según la decisión del equipo.

- En este ejemplo, se agregó en el "Resultado esperado" dos opciones de "caja gris", donde se valida que los datos efectivamente se hayan registrado manteniendo la su consistencia. Para esto hay varias opciones dependiendo del repositorio de datos con el que se cuenta, ya sea una BBDD, un servicio, etc. Incluso esta posibilidad dependerá del acceso que tenga el/la tester para consultar la información. Si bien se trata de una prueba de caja negra, es necesario que el equipo de testing tenga un mayo alcance de los datos para su revisión. Motivo por el cual será necesario solicitar acceso, ya sea a la BBDD directamente, o mediante una vista, un servicio, o una tabla de "histórico / Backup", etc, pero evitando intervenir en la lógica interna del producto, preservando así el concepto de caja negra.

c) Ejecución del caso de prueba

Supongamos que ejecutamos ambos casos, y que el caso 23 falló, por lo que tuvimos que reportar un bug. Debemos completar el Bug-ID al caso de prueba para su vinculación.



ID	Título	Pasos + datos	Resultado Esperado	Precondiciones	Bug_ID
23	Pagar compra con tarjeta de crédito válida	1. Ingresar a pagos 2. Seleccionar Modo de pago: Tarjeta de crédito 3. Completar datos de tarjeta con: - DNI: 32456187 - Tarjeta: Visa - Nro Tarjeta: 2345 3222 1677 9816 - Fecha vto: 28/05/2027 - Cod. seg: 710 4. Clickear botón **Pagar**	Se muestra el mensaje: **"El pago se efectuó correctamente".** **Opciones**: 1) Se debe encontrar en la tabla clientes_pagos un registro con la tarjeta termina den 9816 asociada al cliente 432. 2) Al pegarle al servicio pago/visa debe traer la tarjeta termianda en 9816 asociada el cliente 432.	Contar al menos un producto en el carrito de compras. Que exista el cliente 432 asociado al DNI 32456187	#18

Ejemplo 4: Completitud de pantalla
Hay una prueba que es un común denominador de todas las aplicaciones: validar 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 este caso el Resultado Esperado detalla 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. Si la pantalla tiene mucha información, es muy engorroso detallarla sobre todo los aspectos visuales, así es que en estas situaciones lo más práctico y efectivo es adjuntar un mockup.

Técnicas de diseño de casos de prueba
Para comprender este concepto empecemos por pensar en las respuestas a las siguientes preguntas: *¿cuántas pruebas podemos realizar para aportar calidad? ¿las pruebas son infinitas? ¿cómo elegir cuáles sí y cuáles no?*
Pues bien, básicamente la respuesta a la segunda pregunta es ¡Sí! Las pruebas pueden llegar a ser infinitas, no en un sentido matemático, sino metafóricamente hablando. Es decir que el ingenio lleva a la ocurrencia de muchos casos si tenemos en cuenta que la pregunta característica que ronda el cerebro de un/a tester es *"¿qué pasa si?"*, dado que es muy tentador buscar expandir los límites de los datos en una aplicación o intentar flujos complejos y rebuscados. Lo que sí es finito son las variables que suelen comprometer y pesar en la mayoría de los proyectos: **el tiempo y el presupuesto**, lo que impedirá que se escriban todas las pruebas que se nos ocurran, y que en muchos casos tampoco tengan sentido. Con esto nos referimos, por ejemplo, al hecho de evitar pruebas duplicadas en torno a los datos con los cuales probar. Veamos el siguiente escenario de ejemplo:
Tenemos un campo numérico que sólo permite valores entre 0 y 10 inclusive. Se debe validar este dato y mostrar un mensaje "A" para el caso que se encuentre en el rango, y un mensaje "B" para el caso contrario. Entonces, *¿es necesario crear los 11 casos de prueba que muestren el mensaje A?*, son unos cuantos pero es factible. Al realizarlos nos damos cuenta que los 11 casos son casi idénticos, excepto por los datos ingresados. Esto probablemente nos genere dudas y un cuestionamiento a la estrategia. Por otro lado, también debemos validar el caso contrario, entonces *¿cuántos casos habrá que realizar para asegurarnos que funcione bien la situación para el mensaje B?* Esto es un claro indicador de que no será factible abarcar todos los datos y habrá que seleccionar algunos y descartar otros, ya sea porque el caso se encuentra cubierto, porque se trata de una contradicción, o porque nunca se dará tal caso, etc.
Pero *¿cómo discernir qué casos ameritan y cuáles dejamos de lado?*
Para ello es que contamos con tres **técnicas de diseño de casos de prueba**. Las cuales nos brindan la posibilidad de identificar qué casos, en término de datos, son claves y cuáles debemos descartar para así aportar calidad al producto en tiempo y forma.

Las técnicas son:

- Valores límites
- Partición de equivalencias
- Tablas de decisión

TÉCNICA DE VALORES LÍMITES

Esta técnica, tal como su nombre lo indica, nos permite enfocarnos en los datos extremos y límites (cota superior e inferior) de los rangos de entrada.
¿Por qué es necesario realizar esto? simplemente porque son los errores más comunes en la programación, las condiciones lógicas. Como ya sabemos no es lo mismo un “>” (estricto) que un “>=” (mayor igual), es justamente donde se suelen cometer los errores. Es por ello que esta técnica nos permite identificar rápidamente cuáles son los datos necesarios con los cuales ejecutar la prueba. Son aquellos que rondan el límite del valor de restricción.
La técnica consiste en 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”. Así probamos menos casos pero los más críticos.

Veamos algunos ejemplos:

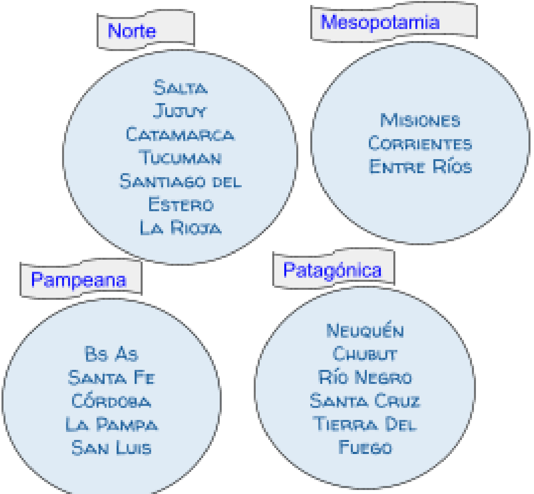
- Si estamos probando un campo edad para saber si la persona es mayor de edad o no, deberíamos crear los siguientes 3 casos de pruebas:
    1. Menor de edad (< 18): un CP con el dato 17
    2. Cumplio (= 18): un CP con el dato 18
    3. Mayor de edad (> 18): un CP con el dato 19

- 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): un CP con el dato 18:59
    2. Llegó justo (= 19:00 hs): un CP con el dato 19:00
    3. Llegó tarde (> 19:00 hs): un CP con el dato 19:01

- Necesitamos 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): un CP con el dato 4900
    2. Extraemos justo el límite (= 5000): un CP con el dato 5000
    3. Superamos el límite (> 5000): un CP con el dato 5100

TÉCNICA DE PARTICIÓN DE EQUIVALENCIAS

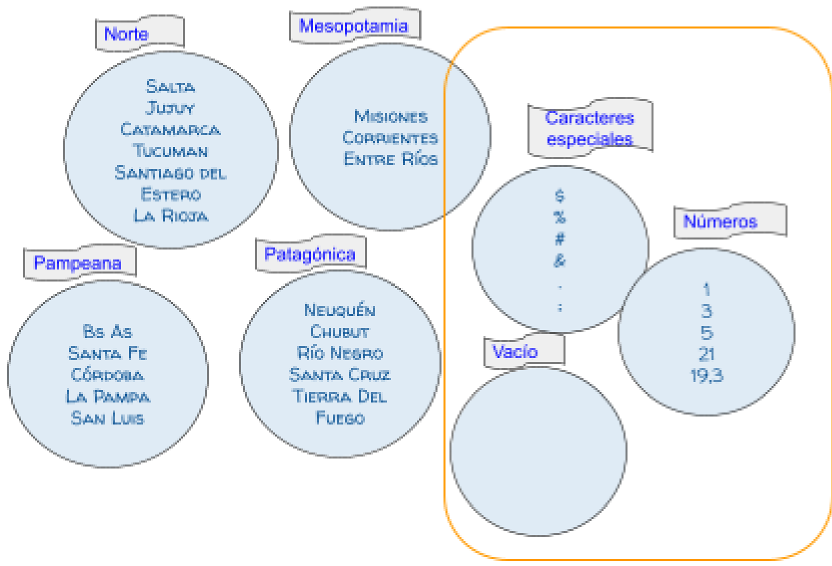
Esta técnica consiste en clasificar los diferentes tipos de datos y armar un conjunto para cada tipo, siendo que sus elementos (datos) tienen el mismo comportamiento en término de resultados sobre la funcionalidad a probar. A estos conjuntos se los denomina **Particiones**, y a la característica en común sobre el comportamiento de sus elementos: **equivalencias**.
De esta manera acotamos la cantidad de casos de prueba, creando casos representativos para cada conjunto, lo que reduce el esfuerzo de prueba sin perder cobertura funcional.
Es decir que se crea un sólo caso de prueba por cada partición con un dato particular, en vez de tener N casos equivalentes.
Veamos un ejemplo: se debe probar un buscador de provincias Argentinas que filtra por nombre de provincia y muestra la región a la que pertenece. Las regiones pueden ser: Norte, Pampeana, Patagónica y Mesopotamia, y las provincias se ingresan mediante un texto. Como ya hemos visto, no es necesario crear casos de prueba para cubrir todas las combinaciones de las 23 provincias con sus 6 opciones de región. Es excesivo.
Para reducir la cantidad de casos pero mantener cierto nivel de calidad vamos a aprovechar esta técnica. Clasificaremos a las provincias por regiones, armando una partición por cada región.



¿Por qué esta clasificación? Pues porque al buscar, por ejemplo, “Salta”, “Jujuy” o “La Rioja” obtendremos el mismo resultado: la región Norte. De la misma manera con el resto de sus elementos, siendo que todas las provincias de dicha región son equivalentes en términos del comportamiento de la funcionalidad *“Buscar provincias”*. De esta forma, se evita crear un caso de prueba por cada provincia, limitando a un sólo caso para una provincia en particular por cada partición. Hasta ahora todos estos casos de prueba son positivos, dado que validan el “camino feliz”. Por lo que faltaría crear las particiones que abarquen los casos negativos.
Siendo que las provincias se ingresan manualmente en lugar de un selector, será necesario contemplar los datos de los siguientes conjuntos:

1. **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 números enteros, como naturales o reales; dado que todos estos datos tienen comportamiento equivalente. Distinto sería si quisiéramos testear la mayoría de edad del ejemplo anterior.

Las particiones resultan:



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

En conclusipón, esta técnica asegura eficiencia al probar los escenarios con un valor representativo de cada partición

TÉCNICA DE TABLAS DE DECISIÓN

Esta técnica en general se utiliza para productos con excesivas funcionalidades y reglas de negocio muy complejas y combinatorias. 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 probar. 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).
Como conclusión podemos decir que esta técnica asegura una cobertura lógica exhaustiva, reduciendo la ambigüedad de los requerimientos.

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 de que alguna falle deberemos reportar su respectivo bug. Para ello contamos con una estructura básica para la interacción con el equipo de desarrollo.

Estructura de un bug

- 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": no informa el error
    - **Correcto**: "El login no valida las credenciales": Sí informa el error

- La **Descripción** debe explicar cuál es el error encontrado reflejando la diferencia entre el resultado que se espera 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 a la home y muestra una página en blanco.

  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 a la home y muestra una página en blanco.

  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 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

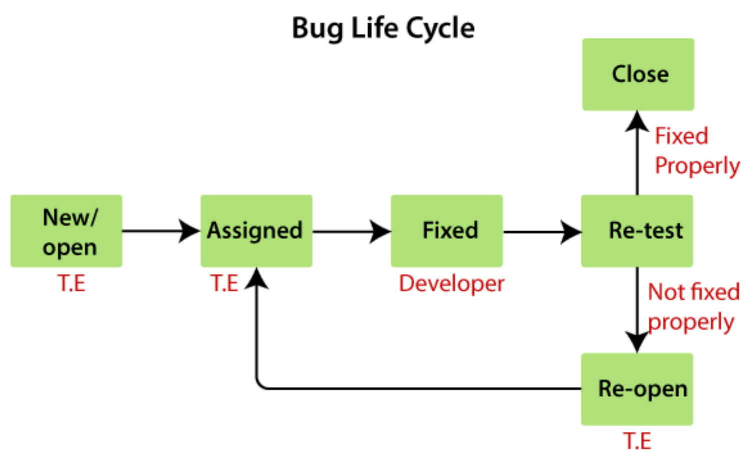
    Definir qué valor le corresponde a cada bug dependerá de la experiencia del tester que lo está reportando, además del conocimiento que se tenga sobre el producto con respecto al negocio.
    No confundir criticidad o severidad con prioridad, son medidas diferentes. La severidad la define el/la tester, mientras que la prioridad el/la PO / analista funcional o directamente el cliente.
    Un bug puede ser crítico, pero no prioritario para el negocio, porque por ejemplo, puede ser una funcionalidad que en breve deje de existir, o se deba modificar en base al negocio, por lo que, a pesar de tener errores graves, no amerita corregirlos.

Veamos como ejemplo el CP_23 que tiene asociado el Bug_id 18:

ID	Título	Descripción	Pasos para reproducir	Versión	Criticidad
18	No se informa el mensaje de pago de compra con tarjeta visa	Al pagar con una tarjeta de crédito Visa se debe visualizar el mensaje "El pago se efectuó correctamente", en cambio no se muestra ningún mensaje informando al respecto. Siendo que la transacción se realizó correctamente en la BBDD.	1. Ingresar a pagos 2. Seleccionar Modo de pago: Tarjeta de crédito 3. Completar datos de tarjeta con: - DNI: 32456187 - Tarjeta: Visa - Nro Tarjeta: 2345 3222 1677 9816 - Fecha vto: 28/05/2027 - Cod. seg: 710 4. Clickear botón **Pagar**	V1.1	Media

Workflow del ciclo de vida de un bug:

El reporte del bug implica una interacción con varios miembros del equipo (en caso del agilismo) o incluso diferentes equipos (en caso de cascada), por lo que será necesario cierta organización para mantener el control y realizar un seguimiento del estado de las incidencias de tipo bug reportadas. Así es que un bug tiene su propio ciclo de vida, en el cual pasa por diferentes estados desde que se inicia hasta que finaliza.
La definición de la cantidad de estados, el nombre de cada uno, y el workflow general quedará a criterio de cada equipo de desarrollo.
A continuación veremos los estados mínimos y estándares que conforman este ciclo de vida:



Tener en cuenta que antes de cerrar un bug, el equipo de testing deberá volver a ejecutar el caso de prueba afectado, dependiendo de su ejecución, para re-abrirlo o para indicar que efectivamente se resolvió. También cabe destacar que no siempre que un bug se resuelve o se cierra es porque se resolvió el error. Este cambio de estado se puede dar por otros motivos:

- No se pudo reproducir
- Quedó obsoleto
- Está duplicado

Técnica TDD

Esta práctica se llama **Test Driven Development (TDD),** y significa **Desarrollo guiado por pruebas**. Es importante hacer hincapié en su nombre, el cual explicita que se trata de una **técnica de desarrollo**, un manera de desarrollar, en lugar de un tipo de prueba.
Al “hacer TDD” no se están diseñando pruebas, **se está programando mediante pruebas**. Esta sutil diferencia es muy importante para evitar confundir los conceptos.
Esta técnica es muy empleada en los proyectos ágiles dado que favorece el enfoque iterativo e incremental. Mediante TDD se pueden programar las US del producto de manera incremental y aportando calidad al software. Por lo que, contrariamente, no se suele utilizar en la metodología tradicional.

Esta técnica consta de 2 etapas:

1. Escribir las pruebas primero
2. Refactorizar el código después

Aquí 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 se va iterando?*
Pues para ello vamos a unificar 2 tipos de pruebas: las de integración, con las de automatización. Pues así se podrá aprovechar la práctica de Devops, que se ajusta muy bien a esta necesidad: **La integración continua** (IC) o Continuous Integration (CI)

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.

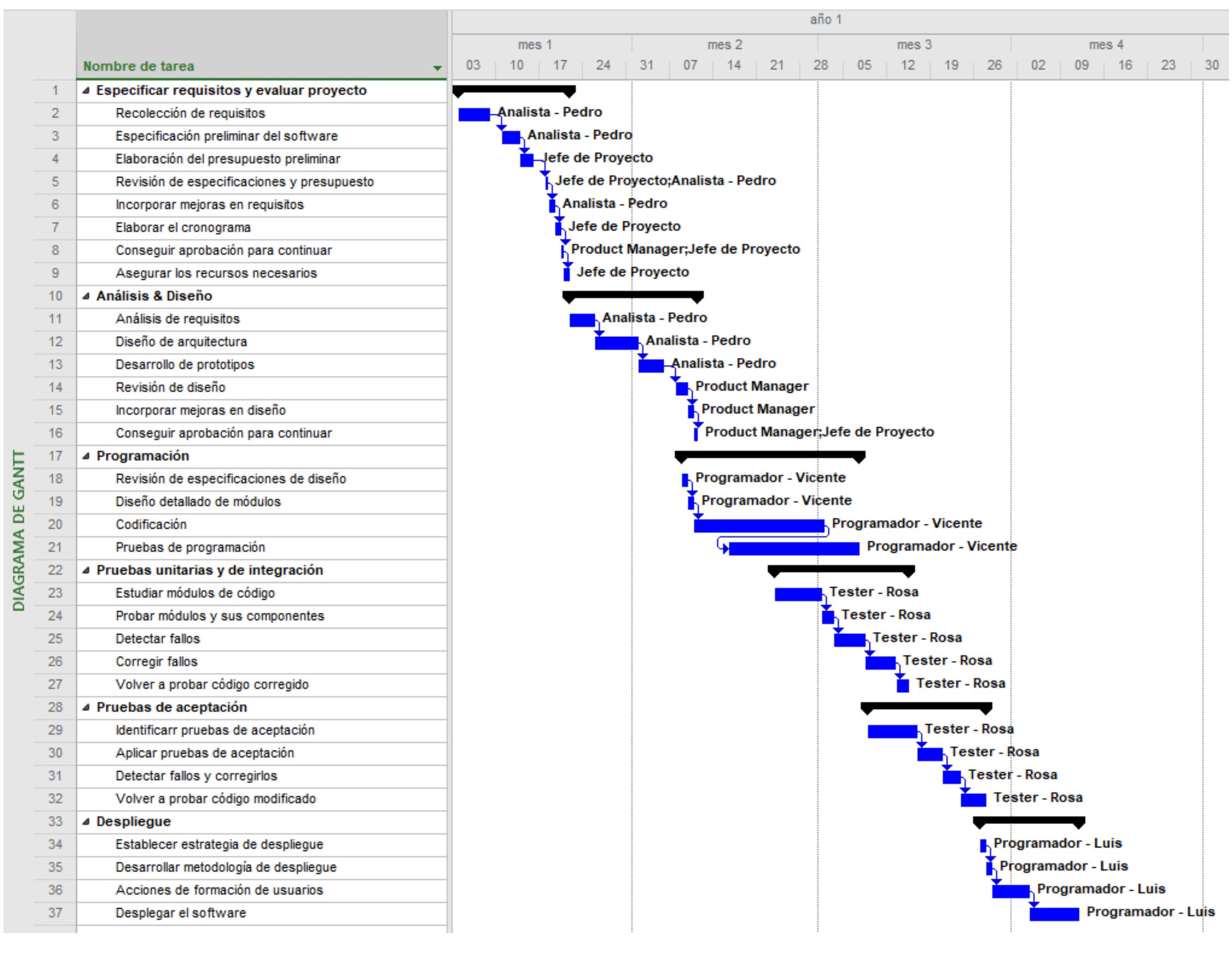
UNIDAD 8: MÉTRICAS

Como parte de la gestión de un proyecto en general, y de software en particular, será necesario tomar ciertas mediciones cuantitativas y estandarizadas para evaluar el rendimiento de un proyecto. Así, en base al estado del proyecto, se pueden detectar retrasos o progresos. Básicamente, funcionan como indicadores que permiten conocer si el proyecto se encuentra encaminado hacia el objetivo planteado. Para ello es necesario contar con herramientas y procesos que nos permitan recopilar información para analizarla y poder medir el estado del proyecto en un momento dado, para finalmente tomar las acciones futuras. A estas mediciones se las conoce como **métricas**. Las métricas son los indicadores que permiten transformar los datos brutos en información valiosa para la toma de decisiones.
Como las métricas forman parte de la gestión de un proyecto, tanto la manera de llevarlas a cabo como las herramientas utilizadas, dependerán de la metodología de trabajo empleada.
A continuación veremos las métricas más utilizadas para cada método: Cascada y Scrum.

Métricas en 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 estándar 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.** Básicamente es una herramienta gráfica que permite visualizar el cronograma, dependencias y evolución de tareas, facilitando estimaciones de entregas. **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 y no un dato preciso, la estimación por tiempo para tareas muy grandes incrementa el nivel de incertidumbre, y por tanto, decrementa la tasa de precisión para con la fecha de entrega final, y con ello la confianza del cliente. Esto se debe a que la herramienta es muy subjetiva 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).

Métricas en Scrum

Estimación por complejidad

Para la metodología ágil en cambio, tanto XP como Scrum, por las desventajas mencionadas previamente, no se suelen estimar las incidencias en tiempo, sino que se estiman en puntos de complejidad. Y como se estiman sólo incidencias que involucran desarrollo, que en su mayoría responden a las historias de usuario (**US**), estos puntos de complejidad se denominan: **Story Points o Puntos de Historia**. Además, por lo mencionado previamente, también será necesario estimar en puntos las incidencias de tipo **bug**. 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 asociada a su complejidad.
Otra diferencia con respecto a cascada, es que en este caso, la estimación la realiza el equipo completo, en lugar de depender del sesgo y la subjetividad de una persona, que por lo general no desarrolla: el PM.
En el refinamiento o, como última instancia en la planning, el equipo deberá estimar cada US y bug (de haber) para poder planificar el sprint. Como técnica estándar, para discernir que este acto sea lo más justo y democrático posible, se suele utilizar la técnica de **Planning Poker.**

Planning Poker

Esta técnica 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 que 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 valor de referencia para tener una con la cual comparar al momento de estimar el resto de las historias. A esta referencia se la llama **Pivot**. Una funcionalidad muy común como pivot es el **login**.

El proceso para estimar cada incidencia es el siguiente:

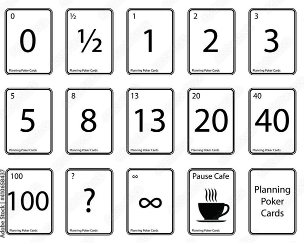
1. **ANÁLISIS:** se analiza la incidencia a estimar entre todo el equipo. Se busca entender en profundidad cuál es el requerimiento, analizando puntualmente los criterios de aceptación y todas las tareas técnicas que incluye. Si hay dudas se debate. Esta fase es muy importante dado que da inicio a las fases siguientes. Es necesario comprender la complejidad de la incidencia para poder establecer un puntaje.
2. **ELECCIÓN DEL PUNTAJE:** en base al análisis y al debate previo, y en comparación con el pivot, cada persona selecciona una carta de su mazo con el valor del puntaje estimado. Esta elección es netamente **personal e individual** con el objetivo de evitar sesgos 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 el evento, y eventualmente el producto y equipo. Es muy común que las personas con más seniority sesgen al resto del equipo, momento en el cual el/la SM debe facilitar el espacio para evitar esta situación, y horizontalizar la actividad.

3. **EXPOSICIÓN DEL PUNTAJE:** cada persona "juega" la carta, es decir que 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 a la fase 2) hasta conseguir unanimidad. Si luego de varias rondas esto no se logró por falta de consenso, se puede acordar democráticamente, el valor que eligió la mayoría.

Veamos a continuación 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.

Aquí las cartas de 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 incidencias más chicas se puede presentar un debate más reñido e interesante. No es igual de importante 1 punto que 3, hay una clara diferencia en la representación del esfuerzo.
En cambio, cuando la incidencia es muy grande, y todo el equipo coincide en ello, termina generando lo mismo que mida 20, 33 o 40 puntos, en cuyo caso ni siquiera amerita debate alguno. En este caso, 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 de la serie también vamos a encontrar algunos "especiales":

- **El valor 0:** indica que no se necesita estimarla
- **El valor 1/2:** indica que es demasiado pequeña, casi que no incide en la velocidad
- **El valor "?":** implica que se desconoce cómo encararla técnicamente o aún quedan dudas sobre el requerimiento. Para el primer caso, si todo el equipo siente lo mismo, se deberá crear un spike, mientras que en el 2do caso, será necesario volver a consultar las dudas. Si las mismas se pueden resolver en el momento, se estima; pero en caso contrario, se deja en el product backlog para consultar a futuro.
- **El valor "∞":** representa que la incidencia es demasiado grande y que en realidad se trata de una épica, por lo 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. Por lo que el tiempo dedicado era tan extenso que amerite hacer breaks.

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.

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 se cuenta a priori, con 2 instancias en las cuales podemos conocer el avance del equipo y 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 gráfico **Burndown Chart**.

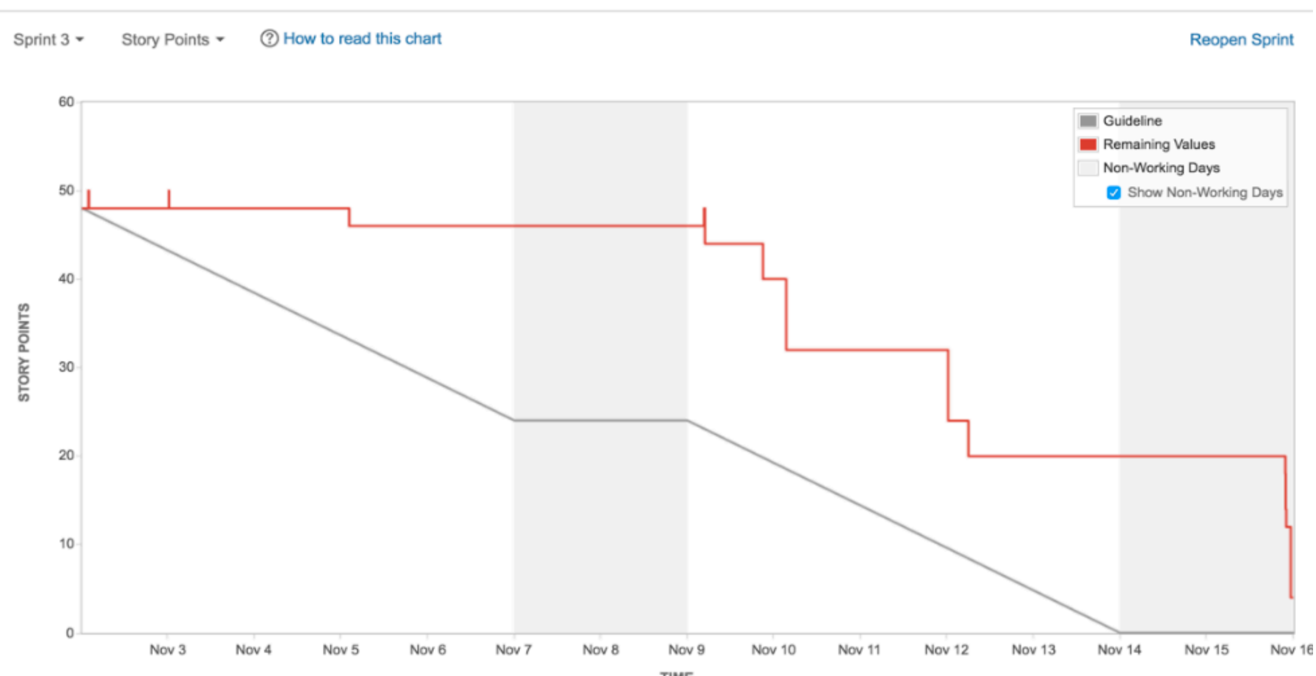
Este cuadro gráfico refleja, cual "foto", la evolución del trabajo y del producto durante el sprint. Pero para que este gráfico sea valioso, será necesario durante el sprint, que el equipo se comprometa a cerrar cada incidencia a medida que se vaya finalizando.
Si bien el concepto de "cerrar incidencia" depende del workflow utilizado, un clásico estándar implica "pasar" la incidencia 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, en el caso de Scrum, 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 cada vez menos puntos (incidencias) a resolver como equipo. Dicha acción se observa como parte de la línea de status dentro del gráfico.

Podemos analizar el gráfico desde 2 visiones:

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

Lo importante de estas visiones es que no existe ninguna que focalice sobre el trabajo individual. Ya que según los valores ágiles, el foco siempre está puesto en el trabajo de equipo en lugar del 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: velocity

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? ¿cuántas historias es capaz de resolver en un tiempo determinado (en un sprint)? ¿Nos sirve la experiencia previa?*
Si bien sabemos que el agilismo no utiliza el tiempo como unidad de medida en términos de métricas y de contratos, es necesario tenerlo en cuenta. Entonces, la pregunta que aflora es: *¿cómo sabe el equipo cuándo podrá entregar ciertas funcionalidades?*
Pues bien, para ello se cuenta con el concepto de **velocidad del equipo**. Ésta representa la unidad de medida que tiene el equipo en término del trabajo entregado. Puntualmente significa la **media** de trabajo que el equipo suele resolver en una iteración (sprint), representada en puntos de historia. Si bien continuamos sin mencionar el tiempo, para responder a la pregunta de "cuándo", será cuestión de realizar un cálculo.
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. Así es que, si bien la velocidad no está expresada en la variable tiempo, al relacionar los puntos con la duración del sprint se puede calcular un tiempo determinado.

Consideraciones importantes

- La velocidad del equipo va variando. Es común que en los primeros sprints aumente conforme el equipo vaya aprendiendo sobre el producto, hasta encontrar cierta estabilidad. Pero también puede ocurrir que decremente en caso de que el equipo sufra modificaciones que lo desestabilicen, como ser vacaciones, licencias, enfermedades, capacitaciones, o inclusive que sufra un 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, podrá calcular su capacidad, es decir que podrá determinar cuántas incidencias se compromete a entregar en dicho sprint. No olvidar tener en cuenta las circunstancias especiales del equipo, como ser las vacaciones, licencias, etc, dado que tener una persona menos en el sprint va a alterar la velocidad del equipo.
- Hay una sutil diferencia entre capacidad y velocidad. La Velocidad es lo que el equipo ha hecho (promedio histórico - pasado), mientras que la Capacidad es lo que el equipo puede hacer en el próximo sprint (ajustada por feriados, bajas o capacitaciones - futuro).
- Un punto vital es entender que la velocidad es única de cada equipo. No se pueden comparar los 20 puntos del "Equipo A" con los 20 puntos del "Equipo B", ya que el esfuerzo percibido es subjetivo. Se sugiere evitar la comparación entre equipos.

Conclusiones generales

Para destacar me gustaría concluir la importancia del valor **"Compromiso"** como parte de los pilares de Scrum.
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 mencionamos aquí algunos blogs que permiten 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: Mayo 2025 - Kleer