

Contenido

Unidad 1: Ingeniería de software en contexto	7
Introducción a la ingeniería de software. ¿Qué es?	7
Software	7
Ingeniería de Software.....	7
Estado actual y antecedentes. La crisis del software.	7
Causas	7
Disciplinas que conforman la ingeniería de software.	8
Ejemplos de grandes proyectos de software fallidos y exitosos.	8
Proyectos fallidos.....	8
Proyectos exitosos	8
Ciclos de vida (Modelos de Proceso) y su influencia en la Administración de Proyectos de Software.	8
Relación entre el ciclo de vida del proyecto y del producto	9
Clasificación de los ciclos de vida.....	9
Administración de proyectos dependiendo en ciclo de vida	11
Procesos de Desarrollo Empíricos vs. Definidos.	11
Procesos Definidos.....	12
Procesos Empíricos	13
Ciclos de vida (Modelos de Proceso) y Procesos de Desarrollo de Software.....	13
Ventajas y desventajas de c/u de los ciclos de vida. Criterios para elección de ciclos de vida en función de las necesidades del proyecto y las características del producto.	13
Criterios para la selección de un modelo de proceso.....	13
Diferentes modelos de ciclo de vida.....	14
Componentes de un Proyecto de Sistemas de Información.	16
Proyecto	16
Administración de proyecto de software	16
La triple restricción – enfoque tradicional.....	17
Líder de Proyecto	18
Equipo de proyecto.....	19
Stakeholders	19
Plan de proyecto	19
Planes de Soporte	24
Causas de fracasos de proyectos	24
Vinculo proceso-proyecto-producto en la gestión de un proyecto de desarrollo de software.....	25
No Silver Bullet.....	25
Unidad 2: Gestión Lean-Agile de Productos de Software.....	29
Manifiesto Ágil/Filosofía Lean	29

Manifiesto ágil	29
Filosofía Ágil	32
Pilares del empirismo	32
¿Cuándo es aplicable ágil?	33
¿Por qué ir a ágil?.....	34
El triángulo ágil	34
Filosofía Lean	35
7 Principios Lean	35
Relación Lean-Ágil.....	37
Requerimientos en ambientes lean ágil	38
Introducción al Desarrollo ágil	38
Requerimientos en ambientes ágiles – User Stories	38
Fundamentación de los requerimientos ágiles	38
La triple restricción – enfoque ágil	39
BRUF – Big Requirements Up Front	40
Product Backlog	40
Requerimientos Just in Time.....	40
Fuente de requerimientos	41
Tipos de requerimientos	41
Principios ágiles relacionados con requerimientos ágiles	43
User Stories	43
Partes de la User Story.....	44
Ventajas	44
Desventajas.....	45
Product Backlog y las User Stories.....	45
Tamaño	45
Story Points	46
Niveles de abstracción	46
Modelado de Roles	47
Criterios de aceptación	47
Pruebas de aceptación.....	48
DoR (Definition of Ready) – Definición de Listo	48
INVEST Model	48
DoD (Definition of Done) – Definición de Hecho.....	49
Spikes	49
Investment Themes (Temas de Inversión)	50
Estimación en ambientes ágiles.....	51
Errores en las estimaciones	51

¿Por qué estimamos?	51
Proceso de estimación.....	52
Métodos utilizados para estimar	52
Problemas de estimaciones tradicionales	53
Estimaciones en ambientes ágiles	53
¿Qué se estima?.....	55
Tamaño vs. Esfuerzo	55
Velocidad	55
Reestimación.....	56
Planificación	56
Tamaño	57
Story Point.....	57
¿Para qué se estima?	57
¿Cómo se estima la duración del proyecto?.....	58
Poker Planning o Poker Estimation.....	58
Estimaciones en Lean.....	59
Estimaciones en proyectos pequeños	59
Diferencias con las estimaciones en ambientes tradicionales	59
Frameworks de SCRUM a nivel de equipo y escala	60
SCRUM	60
Herramientas de SCRUM	69
Frameworks para escalar SCRUM.....	71
Framework Nexus	71
Framework LeSS.....	73
Métricas ágiles y en otros enfoques	76
¿Qué es una métrica?	76
Dominio de métricas.....	76
Métricas en el enfoque tradicional.....	78
Métricas básicas para un proyecto de software.....	78
Métricas en ambientes ágiles	78
Métricas de software en Lean (KANBAN)	80
Métricas de Producto de Software	81
Resumen métricas en cada enfoque	81
Gestión de productos de software – Planificación de productos – herramientas para definición de productos de software	81
Evolución de los productos de software.....	82
Planificación del producto ágil.....	82
Design Thinking	84

Lean UX	85
Explicación MVP, MVF, MMF con el dinosaurio Lean/Agile	85
Productos mínimos para la gestión de productos.....	88
Relación entre MVP, MMF, MMP, MMR	89
Unidad 3: Gestión del Software como producto	91
Conceptos introductorios de la gestión de configuración.....	91
Introducción.....	91
Definición de Gestión de Configuración	91
¿Cuál es el problema que se está tratando de solucionar?.....	92
Ítem de Configuración	92
Versión	93
Variante.....	93
Configuración.....	93
Repositorio.....	94
Release	95
Rama (Branch).....	95
El rol de las líneas base y su administración	95
Planificación de la gestión de configuración de software	96
Actividades relacionadas a la gestión de configuración	97
Identificación de Ítems de Configuración	97
Control de cambios	98
Auditorías de configuración	99
Informe de estado.....	101
Gestión de configuración en ambientes agiles	101
Continuous Integration	102
Continuous Delivery.....	102
Continuous Deployment – Estrategias de Deployments – Canary Deployments – Blue/Green Deployment.....	103
Unidad 4: Aseguramiento de calidad de Proceso y de Producto	104
Conceptos generales sobre calidad	104
Importancia de trabajar para y con Calidad. Ventajas y Desventajas.	104
Calidad	104
Gestión de calidad	104
Aseguramiento de calidad	104
Problemas en la calidad	105
Aseguramiento de calidad de Proceso y de Producto	105
Calidad en el desarrollo de Software.....	106
Principios de Calidad.....	106
Calidad para Quién – Visiones de Calidad	107

Calidad en el Software	108
Calidad del producto.....	108
Modelos de calidad de Producto	109
Calidad del Proceso.....	110
Estándares de Gestión de Calidad del Software:.....	111
Actividades relacionadas con el Aseguramiento de la Calidad del Software.....	113
Administración de la calidad de Software	113
Principales Modelos de Calidad existentes (CMMI – SPICE – ISO) y sus métodos de evaluación.	114
Lineamientos para la implementación de modelos de calidad en las organizaciones.	114
Modelos para la mejora de procesos	114
Modelo de Calidad para evaluar procesos	116
Roles – Grupos	119
Relación de CMMI con Ágil	120
Diferentes tipos de Auditorías: Auditorías de Proyecto y Auditorías al Grupo de Calidad.....	122
Auditorías de calidad de Software	122
Beneficios de las auditorías	122
Tipos de auditorías.....	123
Roles en una auditoría	124
Proceso de Auditorías: Responsabilidades. Preparación y ejecución. Reporte y seguimiento.....	124
Etapas de una auditoría	124
Herramientas y técnicas utilizadas en auditorías	127
Resultados/hallazgos de una auditoría.....	127
Reporte de auditoría	127
Métricas de auditoría.....	128
Calidad de Producto: Planificación de pruebas para el software- Niveles y tipos de pruebas para el software. Técnicas y herramientas para probar software. Técnicas y Herramientas para la realización de revisiones técnicas del software.....	128
Verificación y validación	128
Revisiones técnicas – Peer Review	129
Definición de estándares	132
Testing en ambientes Ágiles.	132
Contexto.....	132
Definición de Testing	133
Principios del Testing	133
Mitos del Testing.....	136
¿Cuánto Testing es suficiente?	136
Conceptos importantes	137
Niveles de prueba	139

Modelo en V.....	140
Ambientes de Testing	140
Proceso de pruebas	141
Artefactos de Testing	142
El Testing y el Ciclo de Vida.....	143
Estrategias de prueba	143
Tipos de prueba	149
Test Driven Development – TDD	150
Mejora continua de procesos con Kanban	150
Kanban	150
Definición	150
Prácticas de Kanban.....	151
¿Cómo aplicar Kanban?	152
Métricas de Kanban	153
Valores de Kanban	153
Principios directores	154
Principios fundacionales	154
Anexo	156
Diferencias entre Scrum y Kanban.....	156
Ser ágil y hacer ágil	157
Cuadro comparativo Gestión Tradicional vs. Gestión ágil	158
Preguntas frecuentes de la ingeniería de software (Sommerville)	159
Cuadro comparativo Auditorías vs. Revisiones técnicas	160
Cuadro comparativo CMMI por Etapas y Continuo.....	161
Respuesta a pregunta de parcial	162

Unidad 1: Ingeniería de software en contexto

Introducción a la ingeniería de software. ¿Qué es?

Software

El software es un conjunto de programas en conjunto con toda la documentación requerida para definir, desarrollar y mantener los programas ejecutables que se entregan al cliente (archivos de configuración utilizados para la ejecución, documentos para el usuario, documentos que describen la estructura del sistema), como así también las herramientas utilizadas para la construcción de este.

Otra forma de definir el software es como, la información o conocimiento que se presenta en distintos niveles de abstracción, desde los requerimientos del sistema (abstracto), hasta el código que ejecuta el mismo (detallado).

Ingeniería de Software

Es una disciplina que se encarga de todos los aspectos de la producción del software, desde la primera etapa de especificación del sistema hasta el mantenimiento del sistema después de que se pone en operación.

Esta ingeniería es importante ya que:

- Con mayor frecuencia se requiere producir de manera rápida y económica sistemas confiables.
- Resulta más barato a largo plazo utilizar métodos y técnicas de ingeniería de software para los sistemas de software. La mayoría de los costos consisten en cambiar el software después de que se pone en operación.

Además, la ingeniería de software se apoya en varias realidades, entre ellas:

- En primer lugar, se debe hacer un esfuerzo para entender el problema antes de desarrollar una aplicación de software. Es decir, se debe entender el problema y las necesidades del cliente antes de proponer una solución.
- El diseño es una actividad crucial en el desarrollo de un software.
- Realizar un correcto diseño del software, trae consigo dos claros beneficios, calidad de software y facilidad de mantenimiento.

La ingeniería de software nace tras las crisis del software, en la cual existía (y existe) un conjunto de dificultades o errores ocurridos en la planificación, estimación de los costos, productividad y calidad de un software, debido, principalmente, a la baja eficacia que presentan una gran cantidad de empresas al momento de desarrollarlo.

Estado actual y antecedentes. La crisis del software.

El término crisis del software hace referencia a un conjunto de hechos relativos al software, planteados en la conferencia de OTAN en 1968 por Friedrich Bauer, quien recalco la dificultad para generar software libre de defectos, fácilmente comprensibles y que sean verificables. Sin embargo, este término fue utilizado anteriormente por Dijkstra en El Humilde Programador.

Causas

- La evolución de la tecnología del hardware mediante los circuitos integrados permitió la posibilidad del desarrollo de sistemas más grandes y complejos. El salto en el hardware no fue acompañado por un salto en el desarrollo del software, esto en gran parte se debe a que el software es una actividad humana cuyo producto es abstracto e intangible.
- En combinación con lo anterior, la demanda creciente de estos sistemas complejos, la subestimación de la complejidad que supone el desarrollo de software, los cambios a los que tienen que ser sometidos los productos para ser adaptados a las necesidades del cliente y la falta de una disciplina que intervenga en todos los aspectos de la producción del software fueron las causas de esta crisis.

En la actualidad, muchos de los sistemas desarrollados fracasan debido a fallas en el software, que principalmente son consecuencia de dos factores:

- Demandas crecientes: a medida que la tecnología y las técnicas de desarrollo evolucionan, la demanda de sistemas más grandes y complejos aumenta. Los usuarios necesitan que se construyan y distribuyan de manera rápida. La problemática aquí es que los métodos existentes (tradicionales) de ingeniería no permiten lograr esta "espontaneidad", por lo que se deben desarrollar y poner a prueba nuevas técnicas y métodos de gestión y desarrollo.
- Bajas expectativas: muchas de las compañías actuales de desarrollo de software no utilizan métodos de ingeniería de software en su trabajo diario. Por lo tanto, su software con frecuencia es más costoso y menos confiable de lo que debiera. La consecuencia, recae en la conformidad de las empresas y de los usuarios ante un software, es decir, las empresas se conforman con que el software solo "funcione", sin tener en cuenta el valor agregado que el software le debe aportar al negocio del usuario. Es necesaria una mejor educación y capacitación en ingeniería de software para solucionar este problema.

Disciplinas que conforman la ingeniería de software.

- Disciplinas técnicas: nos referimos a actividades que aportan al desarrollo de software como **producto**. Ellas son:
 - Toma de requerimientos
 - Análisis de requerimiento
 - Diseño de software
 - Implementación de software
 - Prueba de software
 - Despliegue de un software
 - Documentación de software
 - Capacitaciones a usuarios
- Disciplinas de gestión: hacen referencia a actividades de planificación, monitoreo y control del proyecto de desarrollo de software. Se utilizan, por ejemplo, metodologías LEAN Agile de gestión de proyectos.
- Disciplinas de soporte: son disciplinas transversales a los procesos de software, que permiten verificar la integridad y calidad de un producto de software. Entre ellas se incluyen:
 - Gestión de Configuración del Software (SCM)
 - Toma de métricas
 - Aseguramiento de calidad (QA - Quality Assurance)

Ejemplos de grandes proyectos de software fallidos y exitosos.

[Proyectos fallidos](#)

[Proyectos exitosos](#)

Ciclos de vida (Modelos de Proceso) y su influencia en la Administración de Proyectos de Software.

Se denomina ciclo de vida a una serie de pasos a través de los cuales un producto o un proyecto progresan en el tiempo. Es decir, que es una representación simplificada de un proceso, el cuál define elementos del proceso (actividades estructurales, productos del trabajo, tareas, etc.) y el flujo del proceso (o flujo de trabajo), que especifica la relación y el orden de dichos elementos. Es decir, son modelos genéricos de los procesos de software; y establecen los criterios que utiliza para determinar si se debe pasar de una tarea a la siguiente. Y son independientes de los procedimientos de cada actividad del ciclo de vida.

Los modelos de proceso pueden incluir las mismas actividades, pero cada uno, desde una perspectiva particular, pondrá más énfasis en algunas respecto a otras y definirán flujos de procesos diferentes.

Además, establece los criterios que se utilizan para determinar cuándo se procede de una tarea a otra.

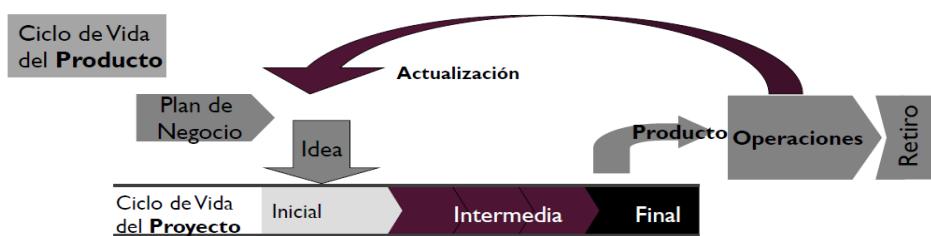
Los modelos de ciclos de vida especifican:

- Las fases del proceso (requerimientos, especificaciones, diseño, etc.)
- Orden en el cual se llevan a cabo.

Relación entre el ciclo de vida del proyecto y del producto

No existe un impacto entre los ciclos de vida de cada uno, sino que el ciclo de vida del producto siempre es mayor que el ciclo de vida del proyecto, ya que el ciclo de vida del proyecto dura lo que dura el desarrollo del software. En cambio, el ciclo de vida del producto dura hasta que el software se deje de utilizar, dependiendo esto de la madurez que tenga el producto en base a las necesidades del mercado.

Es posible que un producto tenga varios proyectos en su ciclo de vida, debido a, constantes cambios y/o actualizaciones que se vayan realizando. Por lo que, dentro del ciclo de vida del producto, se pueden desarrollar varios ciclos de vida de proyectos.



Clasificación de los ciclos de vida

Modelo Secuencial

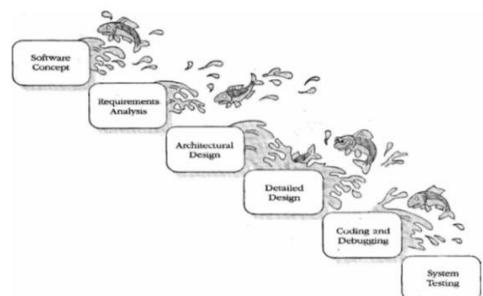
Este modelo dispone de las actividades de forma lineal, es decir, que el proyecto progresa a través de una secuencia ordenada de pasos (fases) y una actividad no puede iniciar sin que la precedente haya sido finalizada. El avance entre las fases se da tras una revisión al final de cada una de estas para determinar si el software está listo para avanzar.

Se suele utilizar en aquellos sistemas donde los requerimientos se comprenden bien y son exhaustivos ya que éstos se congelan al finalizar la etapa de toma de requerimientos.

Generalmente, este tipo de modelos está dirigido por documentos, es decir, el trabajo principal del producto es la documentación del software entre las distintas fases.

Escenarios de proyectos donde elegirlo:

- El modelo en cascada puro ayuda a minimizar los gastos generales de planificación porque se puede hacer toda la planificación por adelantado. No proporciona resultados tangibles en el desarrollo de software hasta el final del ciclo de vida, pero la documentación que se genera a lo largo de este proporciona indicaciones significativas de progreso a lo largo del ciclo de vida.
- Es más sencillo utilizar un método de gestión común a lo largo de todo el proyecto, por lo que aún es común el uso de este modelo.
- El modelo en cascada puro funciona bien para productos en los que se tiene una definición clara y estable, y cuando se trabaja con metodologías técnicas claras.

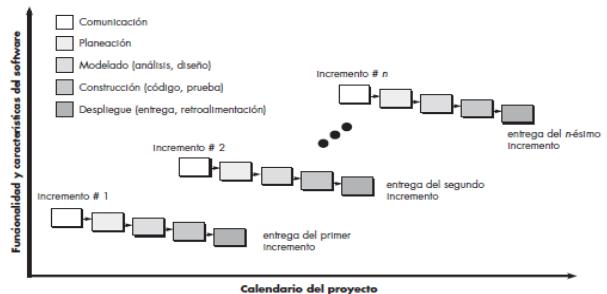


Dificultades

- Los proyectos raramente siguen un flujo secuencial.
- El modelo secuencial exige que los requerimientos sean completamente explícitos desde un principio y esto en la realidad no suele suceder. No puede lidiar con la incertidumbre.
- El cliente obtiene una versión funcional del producto en etapas muy avanzadas del proyecto.
- Encontrar un defecto implica un gran rediseño en la solución, ya que está prácticamente todo hecho.
- Dependencia entre los equipos de trabajo. Un equipo no puede comenzar hasta que el otro no haya finalizado.
- El desarrollo está dirigido por un plan, y cada etapa general documentación para realizar un monitoreo constante contra el plan, por lo que esa documentación puede llegar a ser muy burocrática y excesiva.

Modelo iterativo/incremental

Este modelo aplica sucesivas iteraciones en forma escalonada a medida que avanza el calendario de actividades. Cada iteración produce un incremento de software funcional potencialmente entregable. Usualmente los primeros incrementos incluyen las funciones básicas/críticas que más requiere el cliente. Este enfoque vincula las actividades de especificación, desarrollo y validación. El sistema se desarrolla como una serie de versiones (incrementos) y cada una añade funcionalidades a la versión anterior.



Este ciclo de vida se utiliza en metodologías ágiles.

Ventajas frente al modelo secuencial

- Se reduce el costo de adaptar los requerimientos cambiantes del cliente y el retrabajo es menor ya que en cada incremento se obtiene una retroalimentación que permite adaptar el modelo de las necesidades del cliente.
- El cliente es parte del proceso de desarrollo. En el modelo secuencial, el cliente debía esperar hasta la última instancia para recibir el producto y no era capaz de juzgar el avance del producto a través de documentos de diseño de software.
- Es posible que la entrega sea más rápida la entrega e implantación de software útil al cliente, aún si no se ha incluido toda la funcionalidad. Los clientes tienen la posibilidad de usar y ganar valor del software más temprano de lo que sería posible con un proceso en cascada.
- Muy útil para sistemas de requerimientos cambiantes.
- La especificación, desarrollo y validación están entrelazadas en lugar de separadas y aisladas.

Dificultades desde una perspectiva administrativa

- Se invisibiliza el proceso.
- Alto costo de documentar cada incremento.
- La estructura del software tiende a degradarse frente a una cantidad considerable de incrementos.

Modelo Recursivo

El modelo recursivo es utilizado para gestionar los riesgos del desarrollo de sistemas complejos a gran escala. Requiere de la intervención del cliente. El modelo en espiral es un modelo recursivo, que consta de 4 etapas recursivas:

1. Determinar objetivos.
2. Identificar y resolver los riesgos.

3. Desarrollar y probar.
4. Planificar la próxima iteración.

Para llevarlo adelante, se subdivide el proyecto en varios mini proyectos, intentando resolver en cada uno los riesgos más relevantes hasta que no quede ninguno.

Dicho en otras palabras, se inicia con algo en forma completa, como una subrutina que se llama a sí misma e inicia nuevamente. Se presenta un prototipo que va mejorando con cada vuelta. Lo positivo, es que se generan productos independientes de la implementación, que pueden ser reusables en sistemas de características similares.

Dificultades desde una perspectiva administrativa

- Puede ser más costoso en tiempo y dinero readaptarlos para reutilizarlos para diferentes proyectos.
- La tecnología puede ser obsoleta.
- Puede carecer de mantenimiento o documentación, lo cual dificulta mucho las tareas.
- Se genera una versión del producto recién al final.

Administración de proyectos dependiendo en ciclo de vida

La elección de un ciclo de vida afecta de forma directa a la administración que se debe realizar sobre el proyecto, ya que cada uno de los modelos de proceso poseen características y enfoques distintos. Por ejemplo, si se realiza un análisis de riesgo y se determina que los requerimientos pueden variar mucho a lo largo del desarrollo, se debería elegir un ciclo de vida en el cual el cambio sea permisible, como los iterativos. En este caso, la administración y la planificación del proyecto se ve afectada en la elección del ciclo de vida.

Además, durante el desarrollo del proyecto, la gestión de este se realiza en consecuencia de esta elección, ya que no es lo mismo gestionar un proyecto que se desarrolla de manera iterativa, que uno que se desarrolla de manera secuencial o en cascada.

Procesos de Desarrollo Empíricos vs. Definidos.

Proceso de desarrollo de software

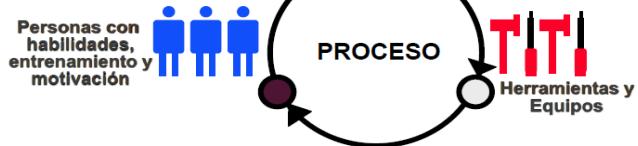
Un proceso es una secuencia de pasos ejecutados para un propósito dado (IEEE), mientras que un proceso de software se define como un conjunto de actividades, métodos, prácticas y transformaciones que la gente usa para desarrollar o mantener software y sus productos asociados.



Las actividades del proceso de desarrollo son la especificación de requerimientos, el análisis, el diseño, la implementación y la prueba. El orden y la relación entre las diferentes actividades está determinada por la elección del ciclo de vida.

Se tienen tres factores determinantes del proceso:

- Procedimientos y Métodos: La adaptación de los procesos establecidos para su implementación, dependiendo del contexto, es esencial para la calidad resultante. Definir las actividades que se harán y cuáles no. Además, es relevante que esta información esté documentada para lograr transparencia. Es decir, tener definidos y escritos los procedimientos.
- Personas motivadas, capacitadas y con habilidades: Es factor primordial para obtener la calidad del producto del proceso, ya que éste se determina del esfuerzo de las personas. Sin ellas, no se puede lograr construir ningún



producto. Por ello, deben estar capacitadas y con habilidades para realizar sus tareas asignadas de la manera correcta y motivados, para lograr aún mayor eficiencia. Recordar que el software es una actividad humano-intensiva.

- **Herramientas y Equipos:** Materiales necesarios para llevar a cabo el proceso que nos permiten que las entradas se transformen en salidas. Se recomienda automatizar la mayor cantidad de actividades posibles, lo que mejora la eficiencia de las personas y puedan concentrarse en tareas que requieran de su capacidad.

Los procesos de software son complejos, por lo que no hay un proceso ideal; además, la mayoría de las organizaciones han diseñado sus propios procesos de desarrollo de software. Los procesos han evolucionado para beneficiarse de las capacidades de la gente en una organización y de las características específicas de los sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere de un proceso de desarrollo muy estructurado (proceso definido). Para los sistemas empresariales, con requerimientos rápidamente cambiantes, es probable que sea más efectivo un proceso menos formal y flexible (filosofía ágil).

Consideraciones

1. Estas actividades varían dependiendo de la organización y el tipo de sistema que debe desarrollarse, pero deben incluir: productos, roles, responsabilidades y condiciones
2. El proceso debe ser explícitamente modelado si va a ser administrado.
3. Un proceso de desarrollo se puede aplicar a varios ciclos de vida. La elección del ciclo de vida depende de la estrategia y este debe permitir seleccionar los artefactos a producir, definir actividades y roles, y modelar conceptos.
4. Las personas utilizan herramientas y equipos para llevar a cabo los procedimientos que componen el proceso de desarrollo.

Actividades fundamentales

1. **Especificación de software:** clientes junto con ingenieros definen el software a producir y sus restricciones.
2. **Desarrollo:** diseño y programación del software.
3. **Validación:** verificación para asegurar que es lo que el cliente quiere.
4. **Evolución:** donde se modifica el software para reflejar los requerimientos cambiantes del cliente y mercado.

Procesos Definidos

Los procesos definidos son considerados deterministas: ante la misma entrada se pretende que se obtendrá la misma salida. Asumen que, si se aplican una y otra vez, se obtendrán siempre los mismos resultados, a pesar de cambiar de equipo.

1. Están inspirados en las líneas de producción.
2. La administración y el control provienen de la predictibilidad del proceso. Es decir, el objetivo es lograr una repetibilidad, para poder obtener una previsibilidad de esfuerzo, riesgos, costos, etc. Asociados al desarrollo de software.
3. Puedo usar cualquiera de los ciclos de vida.

En la realidad, no es posible obtener el mismo resultado ante mismas entradas ni con el mismo equipo, ya que depende sumamente del contexto, momento y las personas, por más parecido sean estos factores.

La administración y control del proceso, en muchos de los casos, toman más importancia que el avance del software en sí (más importante la documentación del avance, que el avance en sí).

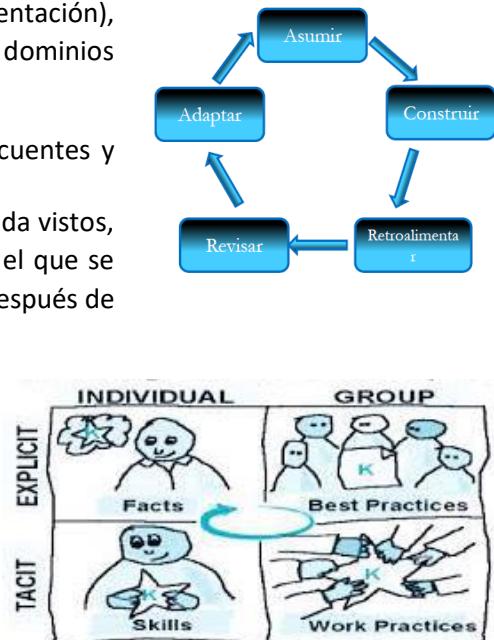


Procesos Empíricos

Los procesos empíricos se conforman en base a la experiencia interna y externa de las personas que intervienen en un contexto particular. NO es considerado determinista ya que no se asume que ante la aplicación del mismo proceso se obtendrán los mismos resultados. Dado que el factor diferencial es la experiencia sensible, en estos procesos se pueden obtener diferentes resultados dependiendo del contexto en el cual se apliquen.

1. Se basan en ciclos cortos de inspección y adaptación (retroalimentación), porque ante un análisis, se ajustan de mejor forma en aquellos dominios complejos donde prima la creatividad y/o complejidad.
2. La administración y el control es por medio de inspecciones frecuentes y adaptaciones para lograr buenas prácticas.
3. No se pueden combinar con cualquiera de los tipos de ciclos de vida vistos, se suele recomendar que sea el modelo iterativo-incremental, y el que se suele prohibir es el modelo secuencial debido a que no es viable después de 2 años (lo dice una teoría).

La experiencia que se obtiene en cada uno de los procesos no es extrapolable, es decir, que solo funciona en ese mismo proyecto, y se necesita adaptar los procesos a cada contexto en especial. Los conocimientos obtenidos en el proceso se deben hacer explícitos al grupo de desarrollo, y convertir las habilidades individuales en buenas prácticas del trabajo en grupo (compartir conocimiento). Además de la inspección y adaptación, los procesos empíricos poseen un tercer pilar muy importante, que es la transparencia de la información ante todos los integrantes del equipo de trabajo.



Ciclos de vida (Modelos de Proceso) y Procesos de Desarrollo de Software.

Ventajas y desventajas de c/u de los ciclos de vida. Criterios para elección de ciclos de vida en función de las necesidades del proyecto y las características del producto.

Criterios para la selección de un modelo de proceso

Es necesario evaluar los niveles de riesgo, la claridad y cantidad de requerimientos, los conocimientos técnicos con los que cuenta el equipo de desarrollo, los plazos que se tienen para ejecutar el proyecto y la posibilidad de poder validar los resultados del trabajo con el cliente, también el trabajo en equipo.

1. Si los requerimientos son claros y exhaustivos, en un contexto de baja incertidumbre en el que el cliente no requiere obtener el producto rápidamente, es posible aplicar el modelo secuencial.
2. Si la situación anterior no se presenta y el cliente desea obtener resultados lo antes posible, con las funciones principales del sistema, entonces sería conveniente la elección del modelo iterativo- incremental.
3. Si el objetivo es disminuir los riesgos presentes en el desarrollo de sistemas grandes y complejos, entonces el modelo recursivo puede ser una buena opción.

Ciclo de Vida	Procesos de desarrollo que lo admite	Características que lo hacen elegible
Secuencial	Definidos	<ul style="list-style-type: none"> ● Alta certeza (baja incertidumbre). ● La organización tiene una forma tradicional de trabajar. ● No se pueden realizar entregas parciales del software (se debe entregar todo junto). ● Eliminar riesgos de planificación.
Iterativo/incremental	Definidos o Empíricos	<ul style="list-style-type: none"> ● Existe incertidumbre. ● Volatilidad de requerimientos. ● Posibilidad de entregas parciales. ● Uso en etapas tempranas del producto.
Recursivo	Definidos	<ul style="list-style-type: none"> ● Mucho control de riesgos en cada iteración. ● No se pueden realizar entregas parciales.

Diferentes modelos de ciclo de vida

Code and Fix

Se desarrolla sin especificaciones o diseño y se lo modifica hasta que el cliente este satisfecho. Los cambios se realizan durante el mantenimiento, lo que es muy caro.

Este modelo tiene dos ventajas:

- No se dedica tiempo a planificar, documentar, hacer control de calidad o cualquiera actividad que no sea codificación pura. Es decir, se salta directamente desde un principio a la codificación y los progresos se ven inmediatamente.
- Requiere poca experiencia, cualquiera que sepa un lenguaje de programación puede familiarizarse con este modelo.

Puede ser útil para proyectos pequeños que se tiene la intención de desecharlos después de construirlos.

Modelo en cascada puro

Se sigue una secuencia de pasos ordenada y se realiza revisión al final de cada fase. Es conducida por la documentación con fases que no se superponen.

- Permite encontrar errores en etapas tempranas.
- Hay exceso de documentación y falta de resultados hasta el final.
- Se utilizan cuando la definición del producto es estable o los requerimientos de calidad dominan a los costos y tiempos.
- Funciona bien cuando se tiene un equipo técnicamente débil o con poca experiencia porque provee al proyecto una estructura que ayuda a minimizar el esfuerzo desperdiciado.
- Este modelo no es flexible, se requiere especificar completamente los requerimientos al comienzo y puede pasar mucho tiempo hasta que se tenga software funcionando. Olvidarse de algo puede ser un error muy costoso.

Modelo en cascada con fases solapadas

Hay un fuerte grado de solapamiento. La documentación es menor, pero es más difícil hacer el seguimiento de progreso.

Modelo de entrega por etapas

Se ven signos tangibles de progreso. Es útil cuando el cliente necesita funcionalidad de inmediato y las necesidades se modifican. Debe haber una planificación rigurosa para que funcione.

Modelo en cascada con retroalimentación

Es una variación del modelo clásico en que es posible volver a una etapa anterior antes de llegar al final, aunque es muy caro hacerlo.

Modelo en cascada con subproyectos

Es un modelo secuencial. Se avanza en cascada hasta el diseño arquitectónico, de ahí en más se avanza en partes dividiendo en Subproyectos.

Modelo en espiral

Busca minimizar los riesgos haciendo un análisis de riesgos y buscando alternativas al iniciar cada fase. Al final de cada fase se planifica la siguiente y se evalúa si se sigue adelante. Este modelo parte el proyecto en mini proyectos que permite manejar mejor los riesgos.

- Permite evaluar la factibilidad de un nuevo producto.
- En proyectos grandes la identificación de los riesgos puede costar más que el desarrollo.
- Es un modelo adecuado para proyectos de vida largos, que puede utilizar equipos diferentes en cada ciclo y muestra el progreso al final de este.
- Por lo general a medida que el costo incrementa, el riesgo disminuye. Mientras mas tiempo y dinero gastas, menos riesgos tomas, que es lo que se quiere en un proyecto de desarrollo rápido.
- Puede ser difícil definir objetivos e hitos que identifiquen que se está listo para avanzar a la siguiente capa del espiral.

Modelo evolutivo

Se fija el tiempo o el alcance mientras el otro se va modificando. Es útil cuando los requerimientos no son claros y se realizan entregas tempranas al cliente.

- El problema es que es un modelo recursivo que repite todas las tareas y depende de que los diseñadores desarrollen un sistema fácil de modificar.

Diseño para cronograma

No se asegura alcanzar la versión final, aunque si una entrega del producto para la fecha definida con los aspectos más importantes. Es útil cuando no hay seguridad sobre las capacidades de programación.

Diseño para herramientas

Consiste en incluir solo la funcionalidad soportada por las herramientas de software existentes. Se usa en proyectos muy sensibles al tiempo. Puede combinarse con otros ciclos de vida, pero no se podrán implementar todos los requerimientos.

Prototipación evolutiva

Se da cuando hay cambios rápidos de requerimientos y los clientes no se comprometen con los requisitos. Requiere menos documentación, pero no se pueden programar los release. Se desarrolla el concepto del sistema a medida que se avanza. Es útil cuando los desarrolladores no están seguros de que algoritmos o arquitectura optima utilizar.

Componentes de un Proyecto de Sistemas de Información.

Proyecto

Un proyecto de software es un esfuerzo temporal que requiere del acuerdo de un conjunto de especialidades y recursos para la creación de un producto, servicio o resultado único. Es una unidad organizativa para adaptar el marco teórico del proceso, dependiendo de las personas y recursos con los que se cuente.

Define el proceso y tareas que se van a realizar, el personal que se encargará de las actividades y los mecanismos que se implementarán para valorar riesgos, controlar el cambio y evaluar la calidad.

Características

Únicos: El resultado de un proyecto será único e irrepetible para cualquier otro proyecto, por más parecido o igual sean sus elementos componentes.

Todos los proyectos por más similares que sean tienen características propias que los hacen únicos, como por ejemplo el calendario (es decir, cronogramas diferentes), o presupuestos.

1. Orientado a objetivos:

No ambiguos: deben ser lo suficientemente claros para guiar el desarrollo del proyecto.

Medibles: es necesario medir el objetivo para determinar el avance sobre el proyecto.

Alcanzable: deben ser factibles de realizarse por el equipo (que el equipo pueda hacerlos).

Alcanzable: deben ser factibles de realizarse por el equipo (que el equipo pueda hacerlos).

Duración limitada de tiempo: Esto implica que un proyecto tiene un principio y un fin, liberando los recursos y personas. Cuando se alcanzan los objetivos del proyecto, este llega a su fin. Un ejemplo claro de lo que no es un proyecto es una línea de producción, ya que esta se ejecuta indefinidamente en el tiempo.

Conjunto de tareas interrelacionadas basadas en esfuerzo y recursos: Se definen las tareas, sus dependencias, los recursos y personas asignadas, permitiendo manejar la complejidad inherente de los sistemas.

Administración de proyecto de software

La administración de proyectos es la organización y coordinación de personas y recursos que permiten desarrollar un producto o servicio de acuerdo con el presupuesto, tiempo y funcionalidades acordadas con el cliente, asegurando que se entregue un producto o servicio de alta calidad. La administración de proyectos es una parte fundamental para que el proyecto sea exitoso. Esta disciplina tiene dos grandes actividades:

- Planificación.

- Monitoreo y control.

La administración incluye:

1. Identificar los requerimientos.
2. Establecer objetivos claros y alcanzables.
3. Adaptar las especificaciones, planes y el enfoque a los diferentes intereses de los involucrados (stakeholders).

Una buena gestión no puede garantizar el éxito del proyecto. Sin embargo, una mala gestión usualmente implica una falla en el proyecto: el software puede entregarse tarde, costar más de lo estimado originalmente o no cumplir las expectativas de los clientes. (esto se verá mejor explicado en la triple restricción).

Un proyecto planificado puede fracasar también, lo importante de la planificación es el acto de planificar, no el resultado. El acto de ponerme en el ejercicio de pensar tanto: riesgos, objetivos, alcances, estimaciones, etc.

Metas más importantes de la administración de proyectos:

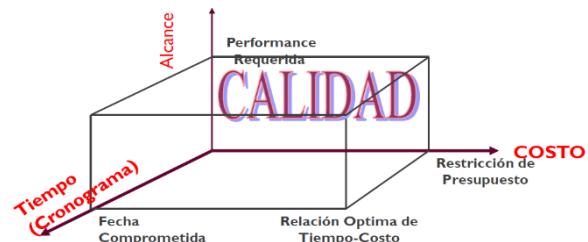
1. Entregar software al cliente en el tiempo acordado.
2. Mantener los costos dentro del presupuesto general.
3. Entregar software que cumpla con las expectativas del cliente.
4. Mantener un equipo de desarrollo óptimo y con buen funcionamiento.

Atributos particulares o complejidades en la administración de proyectos de software:

1. El producto es intangible: No poder ver ni tocar el producto que se obtiene al ejecutar el proyecto dificulta la medición del progreso del proyecto. Una construcción civil es visible y se puede evidenciar el progreso en función de las partes construidas.
2. Los proyectos de software suelen ser excepcionales: Los conocimientos aprendidos en un proyecto de software son difícilmente extrapolables a otro proyecto, como validación de riesgos, aplicación de nuevas tecnologías, etc. El factor tecnológico es una de las causas.
3. Los procesos de desarrollo son variables y específicos de la organización: A pesar del intento de estandarizar los procesos de desarrollo, la realidad es que no es posible predecir de manera confiable qué proceso es el adecuado para adoptar en el proyecto.

La triple restricción – enfoque tradicional

Las variables de restricción de un proyecto son 3: tiempo, costo y alcance. El balance de estos tres factores afecta directamente la calidad del proyecto, ya que un proyecto de alta calidad es aquel que entrega el producto, servicio o resultado requerido, satisfaciendo los objetivos en el tiempo estipulado y con el presupuesto planificado. Es responsabilidad del Líder de proyecto balancear estas variables.



El concepto de la triple restricción hace referencia a que no es posible fijar de forma arbitraria a las 3 variables, sino que será posible fijar 2 y la tercera se ajustará a lo definido.

No se negocia la calidad, si no puedo modificar tiempo y/o costo, debido a que es como entregar un software sin probarlo. La calidad del producto que le entrego al cliente se ve afectada cuando no puedo equilibrar estos 3 factores.

Alcance – ¿Qué trata de alcanzar?

Son los requerimientos que se incluyen en el proyecto y definen la funcionalidad que tendrá el producto a entregar al cliente. En las metodologías tradicionales el alcance es lo primero que se determina y se deja fijo en función de las

necesidades del cliente, mientras que en ágil se va definiendo, refinando y adaptando en base a las retroalimentaciones del cliente. Esta variable es la que primero se negocia con el cliente.

Tiempo - ¿Cuánto tiempo me llevaría?

Define la fecha de calendario en la cual se compromete a finalizar el proyecto y entregarle el producto resultante al cliente. En las metodologías ágiles, el tiempo es fijo.

Costo - ¿Cuánto debería costar?

Define el costo de todos los recursos implicados en el desarrollo del proyecto. Esto incluye equipamiento, infraestructura, equipos, salarios, entre otros.

En la realidad, el costo y el tiempo del proyecto varían de forma directa con la definición del alcance del producto, es decir, el alcance es directamente proporcional al tiempo y al costo. Cuando el alcance aumenta (mayor cantidad de funcionalidades), el tiempo y dinero (costos) necesarios también deben aumentar, para lograr abordar un proyecto más grande. El problema en la aplicación de esta visión en enfoques tradicionales es que, al utilizar procesos definidos con ciclos de vida en cascada, los requerimientos se definen al comienzo del proyecto, por lo que cualquier modificación o cambio en el alcance del producto final, modifica el tiempo y presupuesto de este. Estos cambios, generan una baja en la calidad del producto final ya que, que se sacrifican funcionalidades o bien, se saltan etapas de testing, etc. debido a que el cliente siempre espera que se entregue lo pactado, respetando el presupuesto y tiempo de entrega.

Líder de Proyecto

En un enfoque tradicional, el líder de proyecto realiza la toma de decisiones en su totalidad sin la inclusión de la opinión del equipo. El líder asigna las tareas que deben realizar los integrantes del equipo, y además, maneja todas las relaciones con todos los stakeholders del proyecto (clientes, gerencias, contratistas, stakeholders en general).

Usualmente los profesionales técnicos no disponen de habilidades necesarias para tratar con la gente, por lo que es necesario una figura de líder de proyecto con las siguientes aptitudes:

1. Motivación: habilidad para alentar al personal técnico a producir a su máxima capacidad.
2. Organización: habilidad para adaptar el proceso existente (o inventar nuevos), para lograr que el concepto inicial se transforme en el producto final.
3. Innovación: habilidad para alentar a las personas a crear y sentirse creativas.
4. Empatía.
5. Comunicación.
6. Liderazgo.

También cuenta con habilidades duras, relacionadas a los conocimientos del producto, técnicas y herramientas. Dado que el líder de proyecto tiene un enfoque basado en la resolución de problemas, también son características propias las siguientes:

1. Resolución de problemas: identifica conflictos técnicos y organizativos, estructura la solución o motiva a otros profesionales para que la desarrollen, aplica lecciones aprendidas en otros proyectos y adapta el proceso de resolución ante inconvenientes.
2. Identidad administrativa: tiene la confianza de asumir el control cuando es necesario.
3. Logro: recompensa las iniciativas y los logros, incentivando al equipo a correr los riesgos de manera controlada.
4. Influencia: define la estructura del equipo en función de la retroalimentación que él mismo suministra mediante palabras y gestos.

Entre las responsabilidades del líder de proyecto se encuentran, en las nuevas metodologías:

1. Definir el alcance del proyecto.
2. Identificar a los involucrados, recursos y presupuesto.
3. Detallar las tareas, estimar los tiempos y requerimientos.
4. Identificar y evaluar riesgos.
5. Preparar planes de contingencia.
6. Controlar hitos y participar en las revisiones de las fases del proyecto.
7. Administrar el proceso de control de cambios.
8. Producir reportes de estado.



Equipo de proyecto

Es un grupo de personas comprometidas con alcanzar un conjunto de objetivos de los cuales se sienten mutuamente responsables.

Entre sus características se encuentran:

1. Cuentan con conocimientos, habilidades técnicas, motivación, experiencias y diversas personalidades. Además, deben tener un sentimiento de mejora continua y aprendizaje mutuo.
2. Usualmente son un grupo pequeño, esto permiten lograr una comunicación efectiva entre los miembros del equipo.
3. Trabajan en forma conjunta con espíritu de grupo, convirtiéndose en un equipo cohesivo donde prima la sinergia de este. Para eso, se debe lograr un equipo maduro y estable, es decir, que no se agrega o cambia un integrante constantemente.
4. Sentido de responsabilidad como una unidad y se focalizan en el cumplimiento de las metas grupales.

En equipos cohesivos:

1. El grupo establece sus propios estándares de calidad.
2. Los individuos aprenden de los demás y se apoyan mutuamente.
3. El conocimiento se comparte.
4. Prevalece la mejora continua.

Stakeholders

Son los interesados del proyecto, incluye el equipo de proyecto, el equipo de dirección, el líder de proyecto y el patrocinador.

Plan de proyecto

El plan de proyecto, (o también llamado plan de desarrollo de software en el ámbito del software), es un artefacto de gestión que cumple la función de “hoja de ruta” de un proyecto, en la cual se documentan todas las decisiones que se toman a lo largo del desarrollo; en términos de: el proceso, la división de tareas a realizar, asignación de responsabilidades, equipos de trabajo, calendario de desarrollo, actividades de soporte (planes de contingencia, mecanismos para control de cambios, evaluar calidad y valorar riesgos).

En un plan de proyecto se establece qué se va a hacer (alcance del proyecto), cuándo se va a hacer (calendario), cómo se va a hacer (actividades o tareas para cubrir el alcance) y quién lo va a hacer (responsables de cubrir las actividades). Responder a estas preguntas, implica las siguientes actividades:

1. Definición del alcance del proyecto.
2. Definición de proceso y ciclo de vida.

3. Estimación.
4. Gestión de riesgos.
5. Asignación de recursos.
6. Programación de proyectos.
7. Definición de métricas.
8. Definición de controles.

1. Definición del alcance del proyecto

El alcance de un proyecto, junto a su objetivo, responden al qué se está desarrollando. Es importante diferenciar el alcance de un producto y el del proyecto en sí. El alcance de un producto define qué funcionalidades debe realizar el software para satisfacer los requerimientos del cliente y se mide contra la Especificación de Requerimientos (ERS) (objetivos del producto), en cambio, el alcance del proyecto define todo el trabajo y solo el trabajo necesario que se debe realizar para cumplir con el desarrollo de este y poder entregar el producto o servicio con todas las características y funciones especificadas. El alcance del proyecto se mide contra el Plan de Proyecto (objetivos del proyecto).

Tanto el alcance como el objetivo de un proyecto puede ser modificado a lo largo del proyecto, pero esto no necesariamente tiene que cambiar el alcance del producto. Por otra parte, el alcance de un producto sí condiciona los alcances de un proyecto. (Puede ocurrir que no siempre sea así)

Es importante aclarar que el alcance del proyecto es menor que el alcance del producto. Ejemplos de alcances de un proyecto, pueden ser el realizar la toma de requerimientos, realizar Testing de determinados componentes, hacer el desarrollo del código de un componente, etc.

2. Definición del proceso y ciclo de vida

Al definir el proceso que se va a utilizar, se responde al cómo se va a desarrollar el proyecto. Es importante tener en cuenta la información del contexto del desarrollo: objetivos, alcances, personas y recursos disponibles, clientes, forma de trabajo de personas, etc. Por ejemplo, no es lo mismo utilizar un proceso y ciclos de vida para trabajos presenciales que remotos.

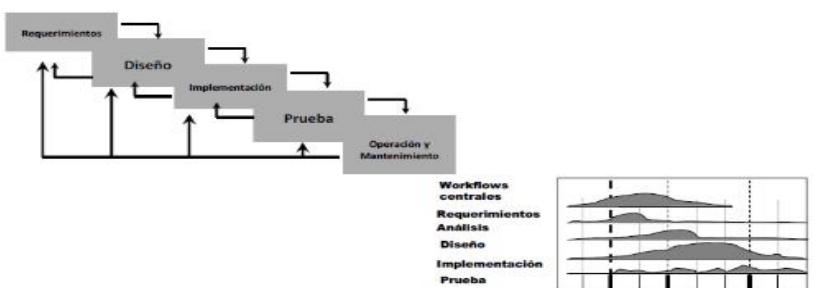
También se debe tener en cuenta la complejidad del software a desarrollar (alcances y objetivos) para así poder adaptar el proceso definido que se elija al proyecto en el cual se está trabajando.

El ciclo de vida del proyecto es lo que define cuánto de cada tarea se debe hacer y en qué momento del desarrollo del proyecto hacerlo.

El ciclo de vida de un proyecto define lo siguiente:

1. Qué trabajo técnico debería realizarse en cada fase.
2. Quién debería estar involucrado en cada fase.
3. Cómo controlar y aprobar cada fase.
4. Cómo deben generarse los entregables.
5. Cómo revisar, verificar y validar el producto.

En metodologías tradicionales, se permite elegir cualquier tipo de ciclo de vida (cascada, iterativo e incremental, espiral, etc.), en cambio en metodologías ágiles esto no es posible (si o si tiene que ser iterativo e incremental).



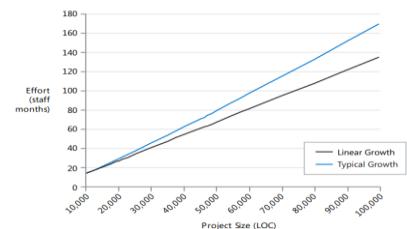
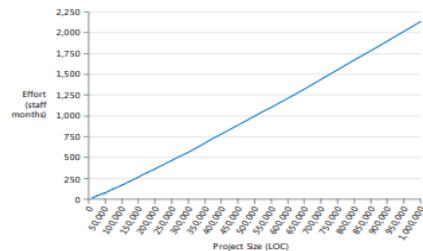
3. Estimación

En un proyecto de desarrollo de software, se trata de estimar para predecir el valor de un elemento relacionado con el proyecto o con el producto, por ejemplo, el tiempo que va a llevar, qué costo va a tener el sistema, cuántas personas necesito para realizar el desarrollo, etc. En un enfoque tradicional, esta estimación de valores es realizada únicamente por el líder de proyecto. En la metodología tradicional, existe un orden definido y recomendado para realizar las estimaciones:

1. **Tamaño del producto:** para estimar el tamaño de un proyecto se utilizan los requerimientos tomados en la etapa de requisitos. Esta estimación es una de las más importantes a realizar, ya que el aumento o disminución del tamaño del producto, afecta de gran manera a otros aspectos, por ejemplo, el esfuerzo aumenta drásticamente a medida que las líneas de código de un proyecto crecen, lo mismo sucede con la estimación de costos y de tiempos (calendarización). Esto es debido a que un aumento del tamaño trae consigo un aumento de complejidad del software, que puede repercutir de muchas maneras en el desarrollo. La mayoría de las veces, el error de las estimaciones recae en estimar costos, tiempos y esfuerzo sin tener una estimación del tamaño, y luego en el transcurso del desarrollo, no variar este costo, tiempo y esfuerzo si varía el tamaño del producto (gran desventaja de la metodología tradicional → no responder a cambios de forma flexible). El tamaño de un producto se puede estimar en muchas escalas, por ejemplo: líneas de código, cantidad de requerimientos, puntos de función ajustados, etc.

2. **Esfuerzo:** refiere a la cantidad de horas persona lineales haciendo una actividad por vez, dentro del desarrollo del producto. Esta estimación intenta responder cuántas horas personas lineales serán necesarias para construir el producto, previamente estimado su tamaño. Como dijimos anteriormente, el esfuerzo varía directamente con el tamaño de un proyecto, ya que, para un proyecto grande, es necesario tener más de una persona cumpliendo el mismo rol, o agregar roles diferentes. Cada rol, puede tener un valor de hora distinto o bien, se puede realizar una estimación en horas planas, en donde no se hace distinción de valor por rol. El esfuerzo es el tópico que más repercute en el costo del proyecto, abarcando casi el 80% de estos.

Además, el efecto de las habilidades personales de los integrantes del grupo define el esfuerzo final del proyecto, es decir dependiendo de la fuerza o debilidad en las habilidades de las personas, los resultados del esfuerzo proyecto pueden variar en una cantidad indicada (gráfico). Por ejemplo, un proyecto con los peores analistas de requisitos requeriría un 42% más de esfuerzo que el nominal, mientras que un proyecto con los mejores analistas requeriría un 29% menos de esfuerzo que el nominal.



3. **Calendario:** la estimación del calendario del proyecto permite responder al cuándo se van a realizar las tareas definidas. En el momento de la estimación, no es necesario realizar una calendarización muy específica a nivel de día, sino que se escala a tareas a realizar en, por ejemplo, un mes del año o incluso un trimestre. Es necesario realizar esta estimación para poder darle una idea al cliente de cuánto tiempo se va a demorar la finalización del proyecto, y como en los demás aspectos, es necesario ya tener una buena estimación del tamaño y del esfuerzo que se requiere, ya que el tiempo de duración del proyecto es proporcional a su crecimiento.
4. **Costos:** de forma similar a la estimación de la duración del proyecto, la estimación de costos depende directamente del tamaño y del esfuerzo que se necesite para el desarrollo del software, por eso es lo último que se estima. De la misma manera, la estimación de costos es muy útil para darle una idea al cliente de cuál es el costo aproximado del proyecto.

4. Gestión de riesgos

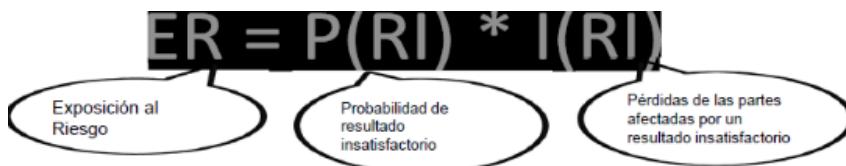
Un **riesgo** es la probabilidad de ocurrencia de una pérdida o daño que afecte de forma directa al proyecto y pueda comprometer el incumplimiento de su objetivo.

Es de suma importancia identificar y definir los riesgos a los que está atado un proyecto, para así poder analizarlos, y realizar una cuantificación de estos, teniendo en cuenta la exposición del proyecto ante los mismos, para así definir a cuáles se debe atender y sobre cuáles se debe hacer un seguimiento.

Los podemos clasificar en:

1. Riesgos del proyecto: aquellos que alteran el calendario o los recursos del proyecto. Un ejemplo de riesgo de proyecto es la renuncia de un diseñador experimentado.
2. Riesgos del producto: aquellos que afectan la calidad o el rendimiento del software a desarrollar. Un ejemplo de riesgo de producto es la falla que presenta un componente que se adquirió al no desempeñarse como se esperaba.
3. Riesgos organizacionales o de negocio: aquellos que afectan la calidad o el rendimiento del software a desarrollar. Un ejemplo de riesgo de producto es la falla que presenta un componente que se adquirió al no desempeñarse como se esperaba.
4. Riesgos organizacionales o de negocio: aquellos que afectan a la organización que desarrolla o que adquiere el software. Por ejemplo, un competidor que introduce un nuevo producto.

Para medir los riesgos se utiliza lo que se conoce como exposición al riesgo (probabilidad x impacto), la cual se obtiene como el producto entre la probabilidad de ocurrencia de la amenaza por el impacto que generaría si realmente ocurriera. Este valor permite organizar y gestionar los riesgos, y se debe hacer foco en reducir la exposición del riesgo.



Antes los riesgos identificados, es posible tomar 3 actitudes:

- Negación: hacer de cuenta que los riesgos no existen.
- Gestión Reactiva: esperar a que el riesgo ocurra y recién ahí ver qué hacer.
- Gestión Proactiva: ir trabajando sobre el riesgo generado planes de mitigación y de contingencia.

Al tomar una actitud proactiva, la identificación y análisis de estos riesgos, permite definir para cuáles de esos riesgos identificados es importante definir planes de mitigación y contingencia. La Identificación y análisis permite reconocer los riesgos y analizar su importancia, luego los planes de mitigación permiten bajar la exposición del proyecto ante ese riesgo identificado y analizado. Por último, los planes de contingencia permiten actuar en caso de que un riesgo suceda, y poder volver a la normalidad lo antes posible.

Entre los mayores riesgos en el entorno de desarrollo de software se encuentran:

- Cambio de requerimientos en el medio de un desarrollo (volatilidad de requerimientos).
- Abandono de integrantes del equipo de desarrollo (alta demanda en mercado laboral).
- Falta de capacidad del equipo.
- El gran avance tecnológico, es decir, no tener en claro el funcionamiento de las últimas tecnologías.

Gestión del riesgo

1. **Identificación de riesgos:** se identifican los riesgos que suponen una mayor amenaza al proceso de ingeniería de software, al software a desarrollar o a la organización que lo desarrolla (al recibir los requerimientos).
2. **Análisis de riesgos:** para cada riesgo identificado se realiza un juicio acerca de la probabilidad y del impacto. No existe una forma certera de realizar esto. Se utilizan intervalos de probabilidad y clasificaciones de gravedad. El criterio dependerá de la experiencia de quien realice el análisis de riesgos.
3. **Planeación del riesgo:** para cada riesgo analizado, se definen estrategias para manejarlos. Estas estrategias consisten en considerar acciones a tomar para minimizar o evitar el impacto sobre el proyecto como consecuencia de la ocurrencia de la amenaza que representa el riesgo. Otras estrategias consisten en desarrollar planes de contingencia, el cual define un plan para enfrentar una situación controversial.
4. **Monitorización del riesgo y control:** proceso que permite determinar que las suposiciones acerca de los riesgos de producto, proceso y organización no han cambiado. Esto permite revalorar al riesgo en términos de posibles variaciones de su probabilidad e impacto. Este proceso se aplica en todas las etapas del proyecto.



5. Asignación de responsabilidades (o recursos)

La asignación de responsabilidades a personas (no le agrada el concepto de recurso humano), permite asignar roles a los integrantes del equipo de trabajo. En el entorno de procesos definidos (PUD, por ejemplo), estos roles se ven definidos en el concepto de trabajadores.

Dentro de un mismo proyecto, una persona puede desenvolverse en más de un rol, ya que puede que el presupuesto del desarrollo de software no sea lo suficientemente grande para permitirse tener una persona distinta por rol, o bien, el proyecto no sea lo suficientemente grande o complejo para justificar dicha adición.

Para dejar explícitos los roles y quienes son las personas que los cumplen, generalmente se construye una tabla, la cual contiene la información de la persona, el rol y responsabilidades de esta en el contexto del proyecto.

Se debe hacer mucho foco en la selección del personal de un proyecto, ya que las capacidades de los integrantes del equipo afectan a la calidad del software y al correcto desarrollo del proyecto en términos de tiempo, por ejemplo, ya que el desarrollo de software es una actividad HUMANO-INTENSIVA.

6. Programación de proyectos (calendarización)

En esta etapa se trata de definir en detalle, cada tarea que debe ser cumplida en el desarrollo. Se toma como base lo definido en la estimación del calendario realizada previamente, pero se refina al máximo detalle posible cada actividad.

Cuando se habla de detallar las tareas, se hace referencia a descomponer el proceso de desarrollo en actividades refinadas a nivel de días, identificando quien la realiza, cuando debe realizarse y cuánto esfuerzo debe llevar en forma teórica.

Generalmente la calendarización se realiza a través de un Diagrama de Gantt, realizado a través de alguna herramienta, como Microsoft Project.

7. Definición de métricas

Las métricas se definen como una medida numérica que aporta visibilidad sobre el avance o estado del proyecto, proceso o producto en un momento determinado del desarrollo. Al realizar la planificación del proyecto, es necesario definir de forma clara y no ambigua, cada una de las métricas asociadas a cada dominio (producto, proyecto o proceso) en conjunto con la forma de calcularlas. Además, es imprescindible que las métricas sean representativas

para el entorno, es decir, que representen un aumento en la información y beneficien la practicidad del desarrollo y seguimiento.

Las métricas del proyecto se consolidan para crear métricas de proceso que sean públicas para toda la organización del software.

A diferencia de las metodologías ágiles, acá se definen métricas que miden y exponen los avances. En cambio en ágil, el mayor indicador de avance de un producto, son los incrementos que se entregan en cada iteración.

Las métricas básicas para un proyecto de software se clasifican en:

- Tamaño del producto
- Esfuerzo
- Calendario (tiempo)
- Defectos
- Costos

8. Definición de controles

En la planificación de los controles del desarrollo, se toma como base las métricas ya definidas, y verifica que todo se está haciendo en concordancia con lo establecido. En esta planificación se definen reuniones periódicas, reportes o informes necesarios, etc.

Planes de Soporte

- Planes de Testing.
- Planes de Mantenimiento.
- Planes de subcontrataciones.
- Planes de calidad.
- Planes de capacitaciones.
- Planes de iteraciones (si el ciclo de vida es iterativo/incremental).

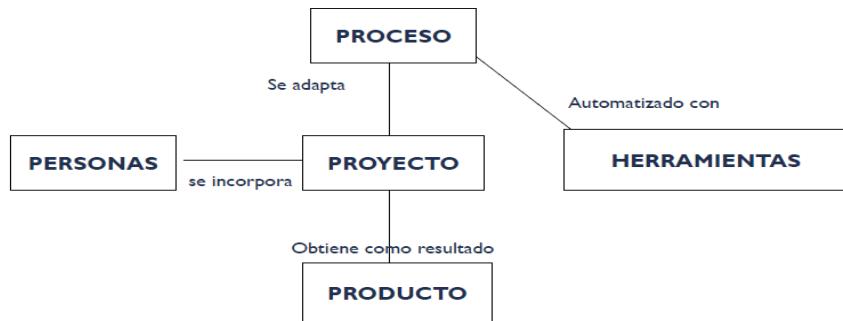
Causas de fracasos de proyectos

- Fallas de toma de REQUERIMIENTOS (el 80%).
- Fallas al definir el problema.
- Planificar basado en datos insuficientes.
- La planificación la hizo el grupo de planificaciones.
- No hay seguimiento del plan de proyecto.
- Plan de proyecto pobre en detalles.
- Planificación de recursos inadecuada.
- Las estimaciones se basaron en “supuestos” sin consultar datos históricos.
- Nadie estaba a cargo.

Vínculo proceso-proyecto-producto en la gestión de un proyecto de desarrollo de software.

Se dice que el proceso, automatizado con herramientas, se adapta al proyecto, al cual se incorporan personas, y de este se obtiene como resultado un producto.

En el enfoque tradicional, previo a la definición del proyecto, es necesario determinar los objetivos y el ámbito del producto para de esta manera poder realizar las estimaciones pertinentes. Los objetivos identifican las metas globales del producto sin especificar como se van a lograr. El ámbito del producto define las funciones y comportamientos que caracterizan al producto que utilizará cliente.



Para obtener como resultado un producto o servicio (ya sea SaaS o SaaS), es necesario que el proyecto adopte un proceso de desarrollo para poder definir qué actividades, métodos, prácticas y transformaciones se utilizarán, mediante las cuales se va a obtener el software. Dentro de cada proyecto de desarrollo de software, es necesario utilizar un proceso diferente dependiendo del tipo de producto que se desea desarrollar, ya que puede que la complejidad de este sea un determinante en la elección del proceso. Es por esto, que es necesario adaptar los procesos a cada uno de los proyectos en los cuales se trabaja, es decir, utilizar solo lo que verdaderamente hace falta para el desarrollo de este, ya que no existe un proceso ideal que sirva para cualquier tipo de proyectos. Es decir, el proceso es una definición teórica de lo que debería hacerse para tomar los requerimientos y transformarlos en un producto o servicio, guados por el objetivo de lo que se quiera construir.

El proceso proporciona un marco conceptual en donde se establece un plan completo para el desarrollo del software. Define como se va a desarrollar el software, estableciendo un conjunto de actividades. Las disciplinas de gestión permiten que las actividades del marco conceptual se adapten a las características del proyecto de software y a los requerimientos del equipo. Las disciplinas de soporte son independientes del marco conceptual y recubren al modelo del proceso, es decir que atraviesan todas las actividades del proceso de desarrollo. Se debe definir el modelo de proceso a utilizar: secuencial, iterativo o recursivo. Luego, el marco conceptual que incluye las actividades fundamentales del desarrollo del software se adapta al modelo elegido.

Un proyecto de desarrollo está integrado por un equipo de trabajo, dentro del cual deben estar los roles bien definidos, para que cada uno de los integrantes asuma la responsabilidad que le corresponda, y se tengan en claro los alcances de cada uno de estos roles. El equipo va a basar su trabajo en el proceso adoptado, en el cual se van a tomar sólo aquellas actividades y formas de trabajo que deberán utilizar y seguir para lograr el desarrollo del producto/servicio deseado.

Las personas son la parte más importante, debido a que el desarrollo de software es una actividad humano-intensiva. Si se adopta el “mejor proceso” para el desarrollo de un software, y además se destina mucho esfuerzo a planificar el proyecto de desarrollo, de nada servirá si no tenemos un equipo capacitado que reúna las habilidades y herramientas necesarias para el desarrollo del mismo.

En el desarrollo de un proyecto, se busca utilizar diversas herramientas que se adapten y permitan automatizar o facilitar las actividades que están definidas en el proceso adoptado. Por ejemplo, el uso de un software de diseño para realizar el análisis y diagramación de clases.

No Silver Bullet

<https://www.studocu.com/es-mx/document/universidad-anahuac/computacion-ii/no-silver-bullet/11916501>

[No hay balas de plata: Lo esencial y lo accidental en la ingeniería del software](#)

- Contrastar progreso del hardware (algo tangible, construible, que se puede mejorar en su construcción), en comparación con el software (intangible, complejo inherentemente)
- Las balas de plata son aquellas soluciones que podrían hacer frente a la crisis del software: no existe un desarrollo de software que demuestre avances de progresión (simplicidad, fiabilidad y productividad) en la construcción del software en menos de una década.
- Dificultades que plantea:
 - **Esenciales**: todo aquello que trae aparejado el construir software: conceptos abstractos, estructuras de datos, algoritmos, invocación de funciones, etc. (carece de precisión y detalles). 4 aspectos la caracterizan:
 - Complejidad:
 - propiedad inherente, debido a la abstracción del software
 - al crecer el tamaño del software, su complejidad crece de manera exponencial, no lineal
 - Esto conduce a muchos problemas: dificultad de comunicación entre miembros de equipos, deficiencias en el producto, errores en cumplimiento de fechas, excesos en costos, etc.
 - Conformidad:
 - el software debe realizar aquello que el cliente necesita, esto es: debe adecuarse a sus requerimientos.
 - Mutabilidad:
 - el software está sujeto a constantes cambios. A diferencia de objetos tangibles, el software muta y sufre cambios a lo largo del tiempo. En cambio, en muchos objetos esto no es así, ya que se reemplazan por nuevos productos (no tomo el objeto y lo modifco, creo uno nuevo).
 - Invisibilidad:
 - al ser algo no visualizable, se dificulta mucho trabajar con representaciones geométricas, como puede ser con estructuras mecánicas. Los grafos son una buena herramienta que permiten modelar muchas situaciones de un software.
- **Accidentales**: tiene que ver con aquellos aspectos que dificultan la construcción del software, pero han sido solucionados a lo largo del tiempo. Esas soluciones son:
 - **Lenguajes de programación de alto nivel**: permiten abstraer el proceso de construcción del software, independizándose del hardware en el que se ejecutarán. Esto elimina la complejidad propia de una máquina, que nada tiene que ver con el software que se desea construir.
 - **Tiempo compartido**: reducción de los tiempos de respuesta de un software. Esto permite no desviar el foco de lo que se desea construir.
 - **Entornos de programación unificados**: facilitan la construcción del software al integrar librerías, formato de archivos, frameworks, etc.

- **Resolución de dificultades accidentales**

- Programación orientada a objetos: permite a los diseñadores del software, abstraerse de tecnicismos para expresar su diseño simplemente en un lenguaje de máquina. Esto facilita el diseño, pero sólo elimina una dificultad accidental, no elimina la propia dificultad esencial de diseñar software.
- Inteligencia Artificial: si bien la IA permite resolver muchas cuestiones que reemplazan acciones humanas, siempre tiene que haber un equipo de personas qué programe y “enseñe” a esa IA cómo actuar. Con lo cual, el factor humano no se reemplaza del todo.
- Sistemas expertos: son sistemas que aplican la IA y permiten sugerir diferentes situaciones como estrategias de Testing, optimizaciones de código, sugerencias en nombrado de variables, etc. Igualmente, no se quita el factor humano ya que se necesita encontrar expertos y desarrollar técnicas eficientes para extraer lo que ellos saben para destilarlo dentro de bases de reglas de estos sistemas. El prerequisito esencial para construir un sistema experto es tener un experto
- Programación automática: esto se planteó como un reemplazo a los programadores, pero en lugar de ser eso, terminan siendo un lenguaje de más alto nivel que el programador debe parametrizar.
- Programación gráfica
 - Permite programar a un nivel más alto de abstracción que la programación tradicional, pero sigue estando el factor humano presente, con lo que solo se logra dar una mayor abstracción al programador sin que este tenga la necesidad de aprender sintaxis del lenguaje.
- Verificación de programas:
 - Lo que se logra con la verificación de programa, es establecer que un programa pasa las pruebas matemáticas de verificación que se proveen.
 - No permite verificar que se cumplan todas las necesidades del cliente que fueron previstas en el software en forma de funcionalidades, sino que verifica que las funcionalidades desarrolladas no tengan errores.
 - Además, las pruebas matemáticas pueden también estar sometidas a errores.
- Environments and tools
 - La mayoría de los nuevos entornos inteligentes tan solo nos permiten librarnos de errores semánticos y sintácticos simples.
- Estaciones de trabajo poderosas
 - El incremento de potencia y memoria de las estaciones de trabajo individuales no soluciona el déficit en el desarrollo el software.
 - La edición de programas y documentos son soportadas totalmente por las máquinas de hoy en día.
 - Una mejora en tiempos de compilación no representa un avance significativo.

- **Resolución de dificultades esenciales**

- Comprar, antes que construir: explorar en el mercado soluciones que se puedan conseguir para reemplazar lo que se quiera construir. Si existe un mercado que lo ha demandado, comprarlo siempre será más barato que construirlo. Esto se debe a que al tener una gran cantidad de clientes que lo compran, el costo de ese software se amortiza. Mirarlo como que el uso de N copias de software, multiplica la productividad de ese software N veces.

Lógicamente este enfoque es difícil de utilizar para necesidades muy puntuales, donde no existe una porción de mercado que lo haya demandado previamente.

- Usar prototipos y refinar requerimientos: la parte fundamental de construir software es saber qué se debe construir. La extracción iterativa de requerimientos y su posterior refinamiento permiten tomar las necesidades de los clientes en sucesivas iteraciones, debido a que ellos mismos no saben expresar ni conocer muchas veces cuáles son esas necesidades que tienen. Además, es prácticamente imposible que sepan expresar con detalle y precisión todos sus requerimientos para el software que necesitan.

El desarrollo de prototipos rápidos permiten cubrir esa extracción iterativa de requerimientos, al simular interfaces y funciones principales de un software que se necesita construir. Esto permite obtener un feedback valioso del cliente, para que todos comprendan mejor cuáles son las funcionalidades que se necesitan, y cómo se deben ejecutar.

- Desarrollo incremental: (crecer y no construir software): Incrementar el software añadiendo funcionalidades mientras estas se prueban, usan y testean.

Tener un software con pocas funcionalidades, pero funcionando, es más animador que tener un avance de un software complejo que no funciona.

- Personas capacitadas: que sigan buenas prácticas. Esto permite tener un buen equipo, pero, al fin y al cabo, desarrollar software requiere de creatividad, por lo que priorizar la creatividad en un equipo puede traducirse en grandes resultados (estructuras rápidas, pequeñas, simples construidas con menos esfuerzo).

Esto plantea que tener en un equipo con buenos diseñadores, es tan importante como tener buenos líderes que dirijan ese equipo. Por lo que se debe recompensarlos de la misma forma, debido a su gran valor para el desarrollo del software.

Unidad 2: Gestión Lean-Agile de Productos de Software

Manifiesto Ágil/Filosofía Lean

Manifiesto ágil

Es un documento redactado en 2001 por expertos de la Ingeniería en Software, que promueve la filosofía del desarrollo ágil del software.

Valores ágiles

El Manifiesto Ágil, expresa los siguientes valores, los cuales priorizan ciertos aspectos por sobre otros, pero sin restar importancia a ninguno de ellos. Esto es:

1. Individuos en interacciones, por sobre procesos y herramientas: es preferible un equipo de personas motivadas con un buen funcionamiento y conocimiento, donde la comunicación sea fluida, pero con herramientas pobres para su trabajo, que tener un equipo de individuos desmotivados o poco funcional con las mejores herramientas y procesos. Así lo expresa el manifiesto, debido a que **el desarrollo de software es una actividad humano-intensiva**, y los procesos ágiles intentan capitalizar las fortalezas de cada individuo (como así también paliar las debilidades individuales), en lugar de intentar homogeneizar un equipo, para así lograr una armonía que permita fortalecer la unión y motivación dentro del equipo. Los procesos y herramientas deben ser un complemento para mejorar la eficiencia y éstos se deben adaptar al equipo y no al revés.
2. Software funcionando por sobre documentación extensiva: priorizar el software asegura que se tengan versiones estables, incrementales y mejoradas del software al finalizar cada iteración. Esto permite que a medida que el software crezca en cuanto a funcionalidades, tras cada iteración puede ir mostrándose a los usuarios finales y obtener una retroalimentación de ellos, para que el equipo de desarrollo se asegure de estar trabajando siempre en aquellas funcionalidades de mayor valor para el cliente y aquellas que satisfagan sus expectativas.

Solo se documenta aquello que agregue valor al producto y esté centrado en el cliente (valor agregado). Existe la necesidad de mantener información sobre el producto de software y sobre el proyecto que se realiza para obtener un producto y el enfoque ágil lo que plantea es que **se debe generar la documentación cuándo sea necesario. No plantea “no generar documentación”**.

Los documentos no pueden sustituir, ni pueden ofrecer la riqueza y generación de valor que se logra con la comunicación directa entre las personas y a través de la interacción con el software.

3. Colaboración con el cliente por sobre negociación contractual: esto permitirá no sólo tener una mejor noción de aquellos requerimientos de negocio que necesita el cliente, sino también que el equipo ágil se integre a todas las partes interesadas en el software, para que en conjunto puedan trabajar y lograr los mismos objetivos. Esto también reduce la dependencia de documentación extensiva ya que cada miembro del equipo participa en la toma de decisiones que solían requerir comunicación escrita.

Implica comprometer al cliente como parte importante del producto que se está construyendo.

No se trata de tener a un cliente lejano que no sabe lo que sucede durante el desarrollo, si no tener a uno que esté comprometido con cada entrega, sobre todo al momento de probarlo y dar el feedback.

A veces las negociaciones contractuales imponen ciertas restricciones que impactan de manera negativa en las partes, lo cual se traduce en un mayor distanciamiento en la relación equipo de desarrollo-cliente. A pesar de que los contratos sean necesarios, sus términos y condiciones, con frecuencia dificultan el ida y vuelta planteado en el principio anterior, por lo que se intenta priorizar que el cliente colabore con el equipo de desarrollo, para así trabajar en conjunto tras un mismo objetivo que sea común a ambas partes.

El problema de las negociaciones contractuales en muchos casos en las empresas empiezan cuando el cliente tiene ya una especie de acuerdo formal (contrato firmado) con el equipo por un monto y un plazo y una cantidad de características del producto de software que hay que entregar y el cliente viene y dice “me olvidé

de esto” o “necesito esto” y ahí es donde empiezan los puntos de fricción por qué es donde comienzan las discusiones porque ya existe un contrato firmado donde se establecieron todas las pautas.

4. Responder a cambios por sobre seguir un plan: esto se plantea debido a que es imposible que los usuarios sepan con detalle y certeza todas las funcionalidades que necesitan, y a su vez, con el paso del tiempo surgirán nuevas necesidades, las cuales en un primer momento no se habían detectado. Esto está ligado al entorno cambiante en el que se desempeñan los clientes que demandan un software, por lo que un equipo ágil está abierto a los cambios, para poder responder ante ellos de una manera rápida y eficiente entregando el mayor valor posible para el cliente y los usuarios, en lugar de apegarse y seguir un plan definido confeccionado en etapas tempranas de requerimientos.

Los principales valores de la gestión ágil son la inspección y la adaptación, diferentes a los de la gestión de proyectos tradicional: planificación y control que evite desviaciones del plan. (Los cambios son vistos de buena forma).

Analizando en concreto el proceso de desarrollo de un equipo ágil, al finalizar cada iteración, la experiencia de desarrollar el incremento genera cierto conocimiento en el equipo, que afecta a la planificación en los distintos niveles. De forma similar, mostrar o desplegar esos incrementos a los usuarios finales, también generará nuevo conocimiento para esos procesos de planificación.

Cualquiera sea el motivo, este nuevo conocimiento puede inducir a cambios, los cuales el equipo ágil deberá incluir en su planificación, de forma que siempre se apunte a entregar un producto con el mayor valor posible.

Principios del Manifiesto Ágil

1. Nuestra mayor prioridad es satisfacer al cliente: esto está ligado al valor 3 y 4, debido a que un equipo ágil desarrolla y entrega funcionalidades en el orden especificado por el Product Owner. Es ese rol quien determina la prioridad e importancia de cada funcionalidad o Feature, de manera que en cada incremento o Release del producto, se esté retornando al cliente el mayor valor posible.
Esto está soportado por el hecho de trabajar con User Stories, las cuales plantean necesidades que tienen un valor significativo para los usuarios del software. De esta forma, el equipo de desarrollo se asegura de estar respondiendo a esas necesidades, al implementar User Stories, y no trabajar con tareas aisladas que no aportan valor para el cliente y/o los usuarios.
2. Aceptar que los requisitos cambien, incluso en etapas tardías de desarrollo: ligado a los valores 3 y 4, donde se plantea que el equipo ágil está abierto al cambio, al no seguir un plan estructurado que se confecciona en etapas tempranas, como plantea la metodología tradicional.
Esto permite que el equipo sea flexible, y responda siempre ante las necesidades cambiantes de los clientes y/o usuarios.
3. Entregar software funcional frecuentemente: esto se relaciona al valor 2, donde es imprescindible que un equipo ágil al finalizar cada iteración (las cuales duran períodos cortos de entre 1 a 4 semanas), haya sido capaz de transformar uno o más requerimientos en código desarrollado completamente, testeado y potencialmente desplegable. Es importante destacar que ese código pueda ser desplegado (que cumpla con la DoD), debido a que se debe asegurar que cumpla con lo que el usuario/cliente necesita, de forma que se le entregue el mayor valor posible tras cada iteración, y no entregar una funcionalidad desarrollada parcialmente.
4. Técnicos y no técnicos trabajando juntos durante todo el proyecto (responsables de negocio, desarrolladores, diseñadores): valores 1 y 3, y lo que plantea es la gran importancia de la participación del rol conocido como “Product Owner”, en conjunto con el equipo de desarrollo de software. Este rol, que generalmente ocupado por una persona del “lado del cliente”, debe ser alguien capaz de tomar decisiones sobre el producto, y que además sea capaz de representar el interés y necesidades comunes de todos los usuarios a los que estará destinado el desarrollo del software.

En un enfoque de proceso ágil, la participación del Product Owner es de suma importancia, ya que es quien guía y prioriza tras cada iteración, aquellos requerimientos que serán implementados. Además, generalmente es el encargado de escribir las User Stories, las cuales en cierta forma sirven de recordatorio y guía para el equipo de desarrollo, para conocer qué es lo que deben implementar.

Comúnmente, el P.O. no posee esa disponibilidad que plantea el enfoque ágil, convirtiéndose en el talón de Aquiles en el proceso de desarrollo del software y siendo uno de los principales motivos por los que se fracasa en los enfoques ágiles.

5. Desarrollamos proyectos en torno a individuos motivados: este principio está ligado al valor 1, donde la filosofía asume que aquellos individuos que participan en un proyecto de desarrollo del software deben estar lo suficiente motivados y comprometidos con los objetivos de este. Los participantes de un proyecto se deben ver entre sí como a un único equipo con objetivos comunes, donde las personas que asumen diferentes roles intentan colaborar entre sí para cumplir con esos objetivos, y no se limitan a cumplir con sus tareas y delegar los problemas en el otro.
6. El método más eficiente de comunicar información es con conversaciones cara a cara: ligado con el valor 1, este principio tiene un sustento psicológico, el cual expresa que las palabras expresadas como sonido o texto, sólo representan un bajo porcentaje de la comunicación que se desea lograr hacia otra persona, siendo que la mayor parte está ligada a gestos, expresiones faciales, movimientos corporales, retroalimentación rápida, etc.
7. La mejor métrica de progreso es el software funcionando: ligado con el valor 2, donde el manifiesto ágil plantea que la mejor forma de demostrar el avance en un proyecto de software es a través de la entrega de software funcional y de valor para el cliente. En las metodologías tradicionales existen muchas métricas y documentaciones sobre las cuales se basan para medir el avance de un proyecto, pero poco sentido tiene tener una excelente documentación del proyecto y producto, si el producto en sí no refleja el valor que se intenta entregar al cliente.
8. Los procesos ágiles promueven el desarrollo sostenible: ligado con los valores 2 y 4, esto apunta a la estabilidad en equipos de trabajo a lo largo del tiempo, lo cual se traduce en una mejora continua de relaciones y formas de trabajo del equipo. Aunque en la realidad esto se ve afectado en gran forma por la gran migración de trabajadores entre diferentes empresas del mercado laboral informático, la filosofía ágil promueve que estos equipos se mantengan a lo largo del tiempo, para así mejorar su dinámica de trabajo. Además, mantener un equipo estable a lo largo del tiempo permite que el equipo alcance una madurez tal, en la cual es mucho más fácil realizar estimaciones sobre su participación en el proyecto. Por ejemplo, el tiempo en el que el equipo desarrolla una cantidad de US.
9. La atención continua a la excelencia técnica y al buen diseño mejora la agilidad: esto se relaciona al valor 2, y manifiesta que la calidad del producto no es un aspecto negociable. Es de suma importancia que a medida que el software se desarrolla, se atiendan las necesidades primordiales del cliente y se responda a ellas, entregando al cliente código de calidad.
10. La simplicidad es esencial: ligado a los valores 2 y 3, este principio apunta a que, como equipo de desarrollo, nos centremos en aquellas necesidades que realmente aporten un valor al cliente, y no perder el tiempo en detalles o aspectos que no se traducen en satisfacer requerimientos de los usuarios, ni hacer desarrollos de más que no estén especificados como necesidad del cliente. Es decir, no hacer cosas que el cliente no pidió.

11. Las mejores arquitecturas, diseños y requerimientos surgen de equipos autoorganizados: ligado al valor 1, este principio expresa que los equipos de desarrollo ágiles a diferencia de organizaciones jerárquicas tradicionales, donde la dirección/gerencia o los jefes son quienes imparten órdenes, y todos deben acatarlas, bajo esta filosofía los integrantes del equipo deben ser proactivos, aportando soluciones, conceptos, herramientas, etc. que sean útiles al equipo. Este principio está sustentado por el empirismo, el cual establece que el conocimiento surge dentro del equipo de trabajo. El equipo toma las decisiones en el momento que ellos creen adecuados.

12. A intervalos regulares, el equipo evalúa su desempeño y ajusta la manera de trabajar: ligado a los valores 1 y 4, este principio se basa en los dos pilares de los procesos empíricos (inspección y adaptación), los cuales permiten a los equipos que cada determinados períodos de tiempo evalúen distintos aspectos concernientes a su desempeño, tales como variaciones de personal en el equipo, tecnologías que no han funcionado como se esperaba, cambio de ideas de usuarios, competencia laboral, riesgos, nuevas tecnologías en uso y reflexionar sobre oportunidades de mejoras en general.

Un equipo ágil debe inspeccionar su forma de desempeño, para adaptar aquellos aspectos que sean necesarios corregir, mejorar o incorporar al equipo, de manera que el comportamiento del equipo como un todo mejore, y de esa forma se pueda responder a las necesidades cambiantes de los clientes.

Filosofía Ágil

El objetivo primordial es entregar software de forma frecuente en un entorno cambiante. Estas metodologías se utilizan en entornos con gran variabilidad de requerimientos, involucrando al cliente en el proceso de desarrollo para conseguir una rápida retroalimentación, ya que estos encuentran imposible predecir cómo un sistema afectará sus prácticas operacionales o qué cambios habrá en el entorno (mercado o políticas de negocio) que pueden dejar el sistema completamente obsoleto.

El desarrollo ágil adopta el ciclo de vida iterativo-incremental, donde el concepto del empirismo subyace al proceso, el software no se desarrolla como una sola unidad, sino como una serie de incrementos y cada uno de ellos incluye una nueva funcionalidad del sistema. Esta es una de las diferencias con el PUD o el RUP, que, si bien utilizan un modelo de proceso iterativo - incremental, son procesos definidos.

Usualmente los incrementos se crean cada 2 o 3 semanas y se ponen a disposición del cliente. El cliente se encuentra inmerso en el proceso, ofreciendo una rápida retroalimentación que permiten una inspección y adaptación sobre los requerimientos cambiantes. Por otro lado, se reduce la cantidad de documentación con el uso de comunicaciones frecuentes e informales con el cliente en lugar de reuniones formales con documentos escritos.

Martin Fowler define al enfoque ágil como “un compromiso entre nada de proceso y demasiado proceso”.

Los métodos ágiles son adaptables en lugar de predictivos y están orientados a la gente en lugar del proceso.

Nace buscando un equilibrio entre hacer Software siguiendo rigurosamente un proceso donde a la larga termina siendo más importante cumplir con el proceso que entregar software de calidad y, yéndonos al otro extremo, hacer software a la ligera.

La filosofía ágil se materializa en el manifiesto ágil. Éste está basado en que la forma de trabajo va a ser sustentada en los conceptos de los procesos empíricos, bajo 3 pilares fundamentales, la transparencia, la inspección y la adaptación.

Pilares del empirismo

TRANSPARENCIA: la transparencia nos permite crecer como equipos y transformar el conocimiento que cada miembro del equipo tiene implícito en conocimiento explícito, es decir, que pase a formar parte del conocimiento de todo el

equipo. Esto ayuda a que los procesos empíricos puedan fluir y generar esa experiencia necesaria para que el equipo evolucione.

Se enfoca en “no es mi código, es nuestro código”. Todo lo que desarrollemos y nos pase dentro del equipo, pasa a ser parte de todos.

Los artefactos que tienen poca transparencia pueden conducir a decisiones que disminuyen el valor y aumentan el riesgo.

La transparencia permite la inspección. La inspección sin transparencia genera engaños y desperdicios.

INSPECCIÓN: La inspección puede realizarse para el producto, procesos, aspectos de personas, prácticas y mejoras continuas. Se realiza por todo el equipo y permite detectar varianzas o problemas potencialmente indeseables, se debe inspeccionar el progreso hacia los objetivos acordados.

Es el momento de aprender cosas nuevas que van surgiendo del trabajo que el equipo va realizando.

La inspección permite la adaptación. La inspección sin la adaptación se considera inútil.

ADAPTACIÓN: Se trata de la mejora continua, es la capacidad de adaptarse en función de los resultados de la inspección. Nos permite hacer ajustes en aspectos del proceso que se desvían fuera de los límites o si el producto es inaceptable.

Los ajustes deben realizarse lo antes posibles para minimizar la desviación normal. El equipo debe adaptarse en el momento que aprenda algo nuevo por medio de la inspección.

¿Cuándo es aplicable ágil?

Primero es importante mencionar que ágil no es una metodología o un proceso, ni tampoco un framework completamente definido, sino que es una ideología con un objetivo definido de valores y principios que guían el desarrollo del producto. Esto se debe a que definen prácticas de cómo hay que organizarse, es decir, cómo debe organizarse el equipo.

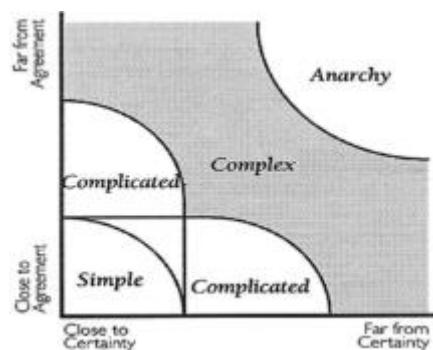
Los frameworks ágiles son distintos marcos de trabajo para implementar los principios y valores ágiles y generalmente son orientados a la gestión de proyectos y no al desarrollo.

La mayoría de los procesos ágiles son prácticas de gestión, no de ingeniería.

En función del gráfico de áreas acerca del entendimiento y certeza que se tiene sobre los problemas, decimos que las **metodologías ágiles** son aplicables en el contexto de lo complejo (dan buenos resultados), en el cual se ubican los nuevos productos sobre los cuales el entendimiento y la certeza que se tiene es parcial.

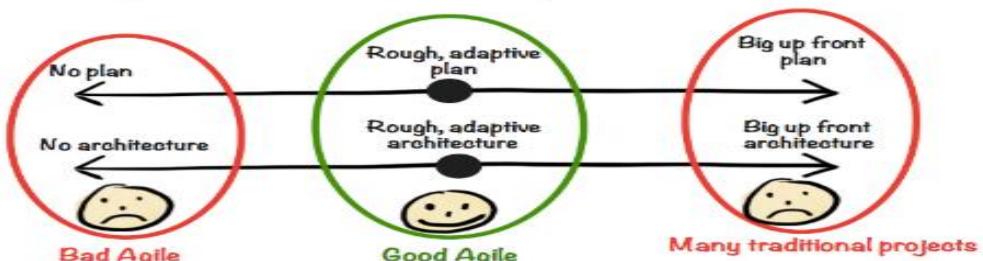
La investigación actúa en el campo de la “anarquía”, donde existe una gran incertidumbre.

Usualmente y no es una regla general, el mantenimiento se encuentra en el campo de lo relativamente simple.



Don't go overboard with Agile!

Ser ágil no es ser indisciplinado, no hay que recaer en lo que la imagen representa.



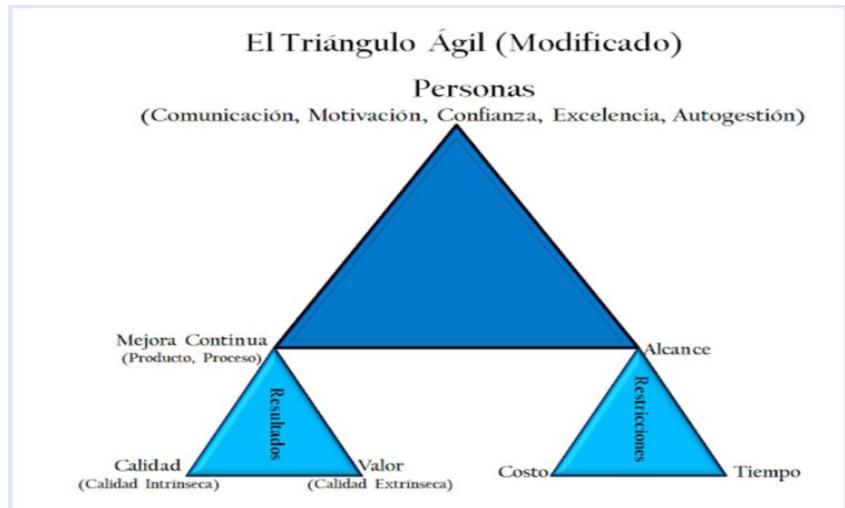
¿Por qué ir a ágil?

- Acelera entregas de productos de **valor para el cliente**.
- Permite adaptarse fácilmente a cambios de prioridades y requerimientos.
- Permite alinear los objetivos organizacionales y de TI.
- Reduce riesgos de proyecto.
- Permite una mejor visibilidad del proyecto.
- Mejora las relaciones en el equipo.
- Reduce costos de proyecto.

El triángulo ágil

El triángulo ágil apunta a que lo que realmente tiene valor que es el software funcionando para el cliente, por lo que se centra más en la calidad de producto, es decir, en la calidad intrínseca (las características de calidad que nosotros le ponemos al producto para que el producto pueda satisfacer al cliente) y la calidad extrínseca (el valor que el producto tiene para el cliente).

El triángulo ágil (modificado) apunta a un producto de software que pone a las personas (que tienen ciertas características como confianza, motivación, etc.) en el lugar más importante ya que las personas son las que hacen el software. La intención de mejora continua que tienen las personas en el proyecto del producto y del proceso.



- **Las personas son lo más importante**, donde se prioriza que haya comunicación, motivación, confianza, excelencia y autogestión. Ya que son éstas las que construyen el software.
- **Debajo encontramos** el pie, en el cual tenemos **la mejora continua** (tanto del producto con las iteraciones como del proceso, cuando habla el equipo entre sí). Es el triángulo de resultados, teniendo como pie a la calidad (la cual es calidad intrínseca, ésta representa a las características que las personas le ponen al producto para que pueda satisfacer al cliente) y al valor (calidad extrínseca, que es el calor que el producto tiene para el cliente).
- **Del otro lado** de la base podemos encontrar **el alcance**, el cual nos da cuáles serían las restricciones de nuestro producto, y encontramos el costo y el tiempo, los cuales suelen ser grandes limitantes a la hora de desarrollar software.

Finalmente se tienen en cuenta las variables que se usan para gestionar el trabajo (alcance- costo-tiempo).

Debemos hacer foco en entregarle al cliente un producto de calidad que se vaya mejorando y evolucionando en forma continua.

Filosofía Lean

Es una filosofía o enfoque más viejo que agile, que nace en Japón ('40) con Toyota, y busca maximizar el valor generado al cliente con el mínimo consumo de recursos, argumentando que todo el esfuerzo que no cree valor, se considera desperdicio. Esto permite reducir los esfuerzos en crear artefactos que no agreguen valor a un producto, concentrando los esfuerzos en los aspectos esenciales.

La filosofía “Lean”, conduce a una visión integrada de la cultura y la estrategia para atender al cliente final con alta calidad, bajo costo y tiempo de entrega, produciendo exactamente lo que el cliente final quiere, cuando lo quiere, dónde lo quiere, a un costo mínimo y precio justo. Siendo el cliente quien determina si el servicio o producto que la empresa entrega tiene valor o no.

Para esto utiliza el método Just in Time, el cual permite el intercambio del producto entre fases del proceso de desarrollo o a la entrega del producto final al cliente justo a tiempo, esto es poco antes de que lo usen y en la cantidad justa y necesaria, “analice cuando lo necesite, no antes”.

7 Principios Lean

Estos principios apuntan al desarrollo de productos en la industria de manufactura. Como el software es una actividad de desarrollo de un producto intangible, a los principios originales de Lean, hubo que adaptarlos al desarrollo de software (matrimonio Poppendieck en “Lean SW Development”).

1. Eliminar desperdicios

Es quizás el más importante. Debemos tener un proceso que sea eficiente y que sume valor en todos los pasos del proceso, todo paso o parte que no contribuye a la generación de valor produce un desperdicio. Todo esto no quiere decir que el proceso es perfecto, porque según todas las filosofías de mejora continua (como esta o agile) marcan que siempre se puede mejorar el proceso.

Lo que se busca evitar:

- Producir funcionalidades en exceso, que luego no se usan (el 80% del valor está en el 20% de las funcionalidades [Pareto]).
- Retrabajo.
- Que los artefactos se vuelvan obsoletos antes de terminarlas.

Este principio se relaciona con los principios agile 3 y 7 (entregar software funcionando frecuentemente y el software funcionando es la principal medida de progreso) y 10 (la simplicidad o maximizar el trabajo no realizado es esencial), porque en el software el trabajo parcialmente hecho y la funcionalidad extra son el desperdicio (en industrias tangibles sería el inventario).

Los 7 desperdicios de Lean

Se pondrán los desperdicios para software y entre corchetes para la industria manufacturera.

- **Características extras [producir productos extras]**

No solo tiene que ver con las funcionalidades que se desarrollan demás, sino también con Up Front Specification (especificar anticipadamente), porque cuando se intenta satisfacer esos requerimientos en etapas posteriores de implementación, la información está obsoleta, incompleta o errónea, debido a que cuando se especificó el requerimiento había incertidumbre o falta de información.

Esto se relaciona con el principio lean 4 (diferir compromisos hasta el último momento responsable).

- **Trabajo a medias [acumulación de stock]**

Esto representa el trabajo que se realiza a medias (US, casos de uso, bugs, etc.) sin terminar, lo cual genera que deba ser retomado posteriormente para su finalización. Esto se relaciona con el

desperdicio de Cambio de Tareas. Por eso en agile se usa una gestión binaria (la unidad de trabajo está lista o no), ya que estimar un porcentaje de avance, es un desperdicio en sí.

- **Proceso extra**

Esto se refiere a los pasos extras que se ejecutan en el proceso de desarrollo que no aportan valor al producto. El objetivo es identificarlos y eliminarlos, para no destinar esfuerzo a actividades que no aportan valor.

Los procesos definidos sobrecargaron mucho a las personas de tareas que no aportan demasiado valor, y como resistencia surgieron los procesos empíricos, que buscan eliminar todos aquellos pasos en los procesos que no aportan valor al desarrollo del producto.

- **Búsqueda de información**

Esto termina siendo un problema importante y difícil de manejar si no se implementa la disciplina SCM, porque no contamos con información suficiente y/o de fácil acceso para tener una trazabilidad sobre los ítems de configuración, lo que dificulta identificar el impacto de los cambios y ralentiza la resolución de errores.

- **Defectos**

Son errores que se trasladan de una etapa a otra etapa posterior en la cual se introdujo. Esto provoca retrabajo, debido a que Testing debe “devolver” la funcionalidad a desarrollo para la resolución de defectos o bugs.

Un antídoto a este problema es la disciplina de Aseguramiento de Calidad, que permite evitar esos errores a lo largo del desarrollo del producto.

- **Esperas [costos de logística]**

Esto es el tiempo de espera que una persona tiene que pasar para poder realizar su trabajo. Ese tiempo de espera puede ser causado porque depende del trabajo de otra persona, porque requiere aprobación de un superior, etc. (por eso la importancia de equipos multidisciplinarios → para evitar tiempos de espera).

- **Cambios de tareas (Multitasking)**

Como el desarrollo de software es un trabajo intelectual, uno de los desperdicios más grandes al hacer este tipo de trabajo, es el cambio de tareas o multitasking. Esto genera una pérdida de tiempo en prepararnos para realizar cada tarea (tiempo de seteo).

Por eso en Kanban se hace uso de limitar el WIP (Work in Progress), porque busca reducir el multitasking. Estos dos últimos desperdicios tienen que ver con el JIT (Just In Time).

2. Amplificar el aprendizaje

Tiene que ver con la transparencia y transformación de conocimiento implícito en explícito, lo cual fortalece al equipo de trabajo.

Crear y mantener una cultura de mejoramiento continuo, donde los individuos intercambien sus experiencias y conocimientos para contribuir con el aprendizaje colectivo. Por otro lado, todo conocimiento que se genere debe compartirse para que sea fácilmente accedido por toda la organización.

Se relaciona con los principios ágiles de (2) recibir requerimientos aún en etapas finales y el principio de (4) personas técnicas y no técnicas trabajando juntos todo el proyecto.

3. Embeber la integridad conceptual

La calidad del producto no se negocia y la simplicidad es un factor clave (principios Agile): el producto tiene que satisfacer las expectativas y necesidades del cliente, esa es la principal prioridad y cada decisión se alinea con esto, no importa si el equipo por momentos se enfoca en otras cosas, no se debe perder de vista que esa es la prioridad principal. Para esto se deben encastrar todas las partes del producto que tengan coherencia y consistencia para hacer al producto más íntegro; donde los requerimientos que cubren la solución se observan como un todo cohesionados armónicamente.

4. Diferir compromisos hasta el último momento responsable

Esto apunta a postergar la toma de decisiones lo suficientemente como para poder tener la mayor información necesaria para tomar esa decisión, pero a la vez, antes de que ese momento sea muy tarde (diferir la decisión irreversible).

Esto se puede reflejar en Agile con el Product Backlog, donde nunca se tiene el 100% de los requerimientos, sino que se va completando a medida que se tiene más información sobre lo que se necesita implementar. Las mejores estrategias de diseño de software están basadas en dejar abiertas opciones de forma tal que las decisiones irreversibles se tomen lo más tarde posible. Se relaciona con los principios ágiles de maximizar el trabajo no hecho y de recibir cambios de requerimientos aun en etapas finales.

5. Dar poder al equipo

Otorgar libertad de acción y poder de decisión al equipo. Para ello, el equipo debe ser multifuncional y autogestionado, y sus miembros deben estar capacitados y motivados. Relacionado con el principio 11 del Manifiesto Ágil (“Las mejores arquitecturas, requisitos y diseños emergen de equipos autoorganizados”).

Esto implica no subordinar por parte de los superiores, debido a que esto anula las capacidades intelectuales, entrenar líderes, delegar decisiones y fomentar buena ética laboral. Pero a la vez, implica un compromiso por parte de los miembros del equipo, para desempeñarse y cumplir con lo pactado de forma completa y correcta.

Este aspecto termina siendo un talón de Aquiles en las filosofías Agile y Lean, ya que en la práctica pocas veces se logran reunir todas estas características.

6. Ver el todo

En Lean se busca tener una visión holística, que permita asociar y comprender el todo, el producto, el valor agregado que hay detrás, el servicio que brinda el complemento de los productos, más allá de los objetivos particulares por áreas o gerencias, porque la idea es que los objetivos particulares de las partes estén alineados con los objetivos globales.

7. Entregar lo antes posible

La idea es entregarle al cliente software funcionando lo más rápido posible y que este producto sea útil para él. Se habla de entregarle software antes de que las necesidades de negocio cambien, porque el ambiente donde se desempeña el mismo cambia.

Para esto Lean propone acotar ciclos de desarrollo (principio 1 de Agile: satisfacer al cliente con entregas tempranas) y entregar rápidamente incrementos pequeños de valor, lo que permite salir pronto al mercado con un producto mínimo que sea valioso (principio ágil de entregas tempranas y frecuentes de software de valor para el cliente).

Relación Lean-Ágil

Lean es previo al Manifiesto Ágil. Algunos principios de Agile heredan de los fundamentos de Lean. Ambos están orientados al cliente, a proveer el máximo valor posible. Para esto, entienden que los individuos de los equipos deben estar motivados, de forma tal que la sinergia de este facilite el desarrollo del proyecto. Esta sinergia, se interpreta como un valor agregado para el cliente. La flexibilidad para adaptarse a los cambios y ofrecer valor al cliente, es también un denominador común de ambos enfoques. Otro aspecto que tienen en común es generar productos de calidad y de mejora continua.

Requerimientos en ambientes lean ágil

Introducción al Desarrollo ágil

Requerimientos en ambientes ágiles – User Stories

Lo más difícil del software es decidir que software queremos construir, ya que las cuestiones tecnológicas no son el impedimento, sino que los proyectos de software fracasan debido a que no se cumplen con las expectativas del cliente: lo que se construye no es lo que el usuario quería o necesitaba. Este problema se advierte en la crisis del software. Probablemente una de estas razones es no asimilar la complejidad que supone hacer software y que la etapa de relevar requerimientos (la cual es social y no técnica) suele ser subestimada.

También, es muy común que haya problemas de comunicación entre los clientes y el equipo. Cuando ninguno de los dos domina la comunicación, el proyecto fracasa. Cuando el cliente o negocio domina la comunicación, es probable que soliciten funcionalidades y fechas de entrega que los desarrolladores no puedan cumplir, o que no entiendan exactamente qué es lo que se requiere. Otro problema es cuando solo los desarrolladores dominan la comunicación, ya que la jerga técnica reemplaza el lenguaje del negocio y los desarrolladores pierden la oportunidad de aprender que es necesario escuchando al cliente.

Lo óptimo es que haya una comunicación de ambas partes, ya que los proyectos suelen fracasar si solo una parte domina la comunicación y asignación de recursos.

Como usuarios, ver “early versions” o entregas tempranas es una ventaja ya que permite que nos surjan nuevas ideas o que nos demos cuenta de que no es exactamente lo que queríamos antes de que el proyecto siga avanzando por un camino equivocado.

Es por esto que no es posible realizar un diagrama PERT perfecto mostrando todo lo que se debe realizar en un proyecto (no se conocen de arranque todas las actividades).

Fundamentación de los requerimientos ágiles

El fondo de la cuestión está en los principios y valores explicitados en el manifiesto ágil, fundamentalmente la entrega frecuente y temprana de software valioso y la posibilidad de recibir cambios de requerimientos aun en etapas finales. Esto último implica una predisposición, una actitud del equipo frente a la posibilidad de cambios. La base de los requerimientos en ambientes ágiles tiene que ver con la posibilidad de comenzar a construir el producto cuando el producto no está completamente definido.

Los requerimientos ágiles se basan en 3 pilares.

- **Usar el valor para construir el producto correcto:** Lo único importante es dejar contento al cliente con software funcionando que le sirva, por lo que, el foco de este enfoque de gestión ágil apunta a construir valor de negocio. Es decir, el software que creamos es para ayudar a alguien más a cumplir con sus objetivos y trabajo, por eso decimos que debemos enfocarnos en construir el producto correcto que le dé valor al negocio. Hay que construir un producto que le haga la vida más fácil a las personas que lo utilizan.
- **Determinar que es “sólo lo suficiente”:** hace referencia a que los requerimientos deben ser encontrados de a poco, no buscar todos los requerimientos desde un inicio y quedarnos solo con eso. Este pilar postula que se tiene que empezar con lo mínimo necesario para generar realimentación y alimentar el empirismo y a partir de allí, se avanza continuamente.
- **Usar historias y modelos para mostrar que construir:** la parte de desarrollo de requerimientos está muy enfocada a la idea de construir junto con el cliente, respetando el principio ágil de “técnicos y no técnicos trabajando juntos”. Se busca trabajar con un referente que esté del lado del negocio, que tenga el conocimiento del producto y que trabaje junto con el equipo.

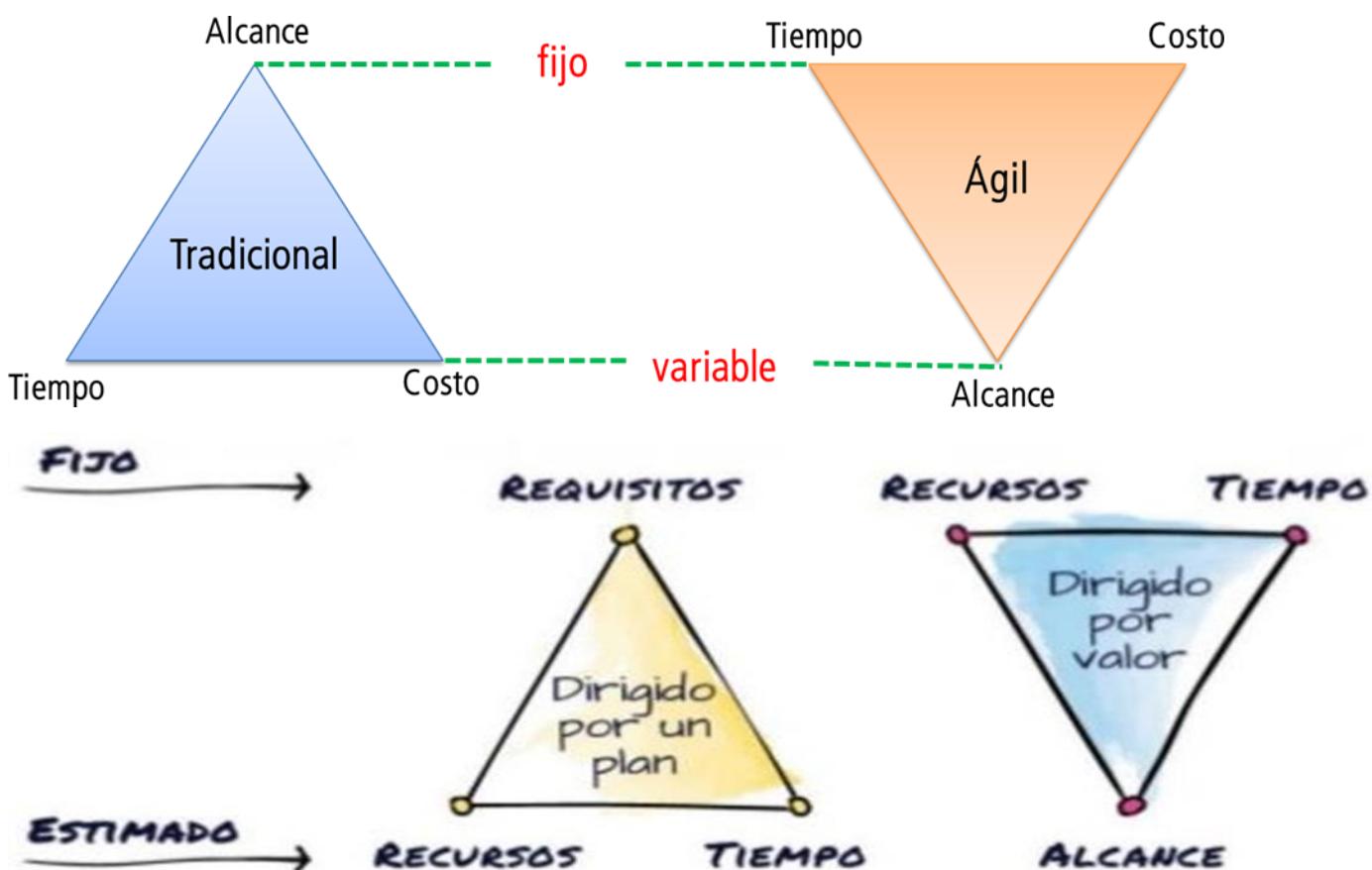
La etapa de capturar los requerimientos se transforma en una etapa de trabajo muy integrado entre el cliente y nosotros. Por eso el mejor medio de comunicación es la comunicación cara a cara porque la idea es armar equipos de trabajo en los que con técnicas de descubrimiento (las User Stories) se pueda lograr crear la visión de qué producto se quiere construir y a partir de ahí, avanzar.

El enfoque de requerimientos ágiles debe estar adherido al manifiesto ágil, es decir, a cumplir con los valores y con los principios que el manifiesto plantea.

Recordemos que la ingeniería de requerimientos tiene 2 procesos: por un lado, **el desarrollo de requerimientos** y por otro lado **la gestión de los requerimientos**. La gestión está más vinculada con mantener los requerimientos íntegros, bien identificados, bien trazados y que si cambian los requerimientos se actualicen todos los lugares donde se tengan que actualizar esa definición de requerimientos. Esto ocurre todo el tiempo mientras estamos trabajando con un producto de software. La parte de desarrollo de requerimientos, que es la parte que tiene que ver con identificar, especificar y validar los requerimientos.

La triple restricción – enfoque ágil

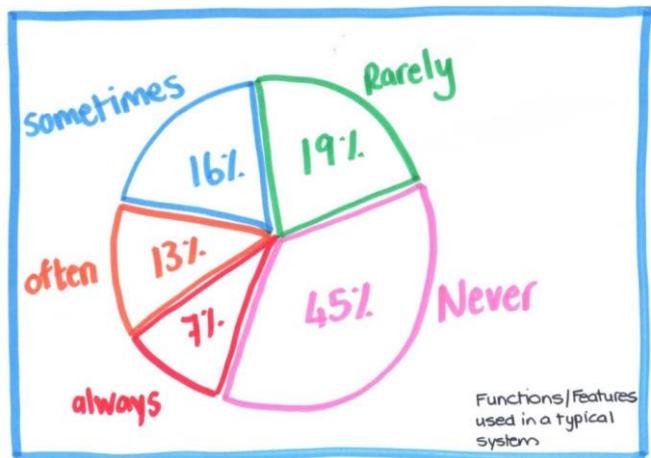
En el enfoque ágil, se trata de aceptar la realidad en la cual el alcance (requerimientos) del producto puede cambiar a lo largo del desarrollo, y no es posible definirlos al comienzo de este. Entonces, el enfoque trata de dejar el tiempo y los recursos fijos, teniendo en cuenta “cuánto software” le puedo entregar en tanto tiempo y con tal equipo de desarrollo.



Por otro lado, en el enfoque tradicional, el alcance se fija y en función de eso derivan los recursos (costo) y el tiempo. El cliente desea controlar el alcance, ya que este comprende lo que él realmente necesita. Luego el cliente pregunta por el tiempo y el costo, los cuales son derivados por el equipo. Por esta razón es tan costosa incorporar cambios de requerimientos, ya que se estaría modificando el alcance y como consecuencia los recursos y el tiempo.

BRUF – Big Requirements Up Front

La imagen adjuntada expresa la frecuencia con la que los usuarios utilizan determinadas funcionalidades de un software. A lo que se apunta con este concepto de BRUF es a notar que, en una primera instancia de un proyecto de software, desarrollar sólo unas pocas funcionalidades de gran valor e importancia para el cliente, puede cubrir gran parte de sus necesidades, a medida que se desarrolla el resto de los requerimientos.



Además, es prácticamente imposible conocer al 100% todas las necesidades de los usuarios, ya que ni ellos saben concretamente qué necesitan, ni cómo explicar de forma abarcadora y completa sus necesidades. Por otro lado, esas necesidades que tienen los usuarios al momento de ser relevadas, con el tiempo probablemente irán cambiando, con lo cual, la priorización es un factor fundamental en esa etapa, donde el equipo en conjunto con el Product Owner, deberán ser capaces de decidir cuáles son aquellos requerimientos indispensables para el negocio, de manera que, si se implementan esas necesidades, se puede entregar una nueva versión del producto que sea de valor para el cliente.

Product Backlog

La gestión ágil busca contrarrestar la situación planteada con anterioridad donde gran parte del software no se termina utilizando. El Product Backlog es una lista priorizada de características y requerimientos del software. Es el Product Owner quien se encarga de definir esta lista ya que es él quien tiene en claro cuáles son sus necesidades y requerimientos principales. El Product Owner se va a encargar de decidir qué requerimientos necesita primero según el valor que éstos tengan para el negocio.

En contraposición con el enfoque tradicional, donde teníamos un cliente que no estaba dispuesto a comprometerse y dejaba la priorización de los requerimientos al equipo que hace el software, se terminaba construyendo un producto que no dejaba satisfecho al cliente. En ágil es el cliente el encargado de esta priorización, lo que lleva a que se construya el producto deseado.

Con esta situación se permite la compensación de no tener documentos de requerimientos formales escritos, detallando todos los puntos. Ágil ofrece una compensación al afirmar que la mejor comunicación es el cara a cara y un cliente participe en el equipo.

Los requisitos cambiantes son una ventaja competitiva si puede actuar sobre ellos



Copyright 2004 Scott W. Ambler

Requerimientos Just in Time

La gestión ágil de requerimientos sostiene los **requerimientos justo a tiempo**, asociados a los principios de eliminar el desperdicio y diferir compromisos, dentro de la filosofía Lean.

Se empieza con una visión del producto y con algunos requerimientos identificados. A partir de esta base, se comienza a construir el producto y se incorpora al cliente para obtener retroalimentación. **El Product Backlog nunca está al 100%**

En el enfoque tradicional se aplica el “Up Front Specification”, mediante el cual se especifica todos los requerimientos antes de comenzar el desarrollo. Esto implica que un error en la captura de requerimientos tiene un alto impacto. En cambio, los requerimientos en ambientes ágiles al ser consensuado continuamente con el cliente, el impacto que se tiene es menor.

El producto no va a estar especificado al 100% desde el principio ni muchísimo menos, sino que se van a ir encontrando requerimientos y describiendo los requerimientos que hacen falta conforme haga falta. “ni antes, ni después”.

Fuente de requerimientos

Los requerimientos ágiles, basados en el principio de que el mejor medio de comunicación es el cara a cara, promueven que los requerimientos del software se obtienen conversando cara a cara con el cliente.

En los ambientes tradicionales, los requerimientos se encuentran especificados en un documento de especificación de requerimientos (ERS) y el cliente luego lo lee para validarlos (sería lo que en el gráfico dice “papel”).

Los agilistas no escriben una ERS, pero lo compensan con la disponibilidad del cliente para resolver las dudas. Esto permite reducir la cantidad de documentación formal. Lalicitación de requerimientos es un proceso que requiere de conversar de forma personal, analizando lo que los clientes dicen y los gestos que manifiestan.

El desarrollo ágil no implica la no documentación o la no especificación de requerimientos, sino que prioriza documentar y especificar aquellos que sean de valor para el cliente, antes que hacer toda una ERS errónea.



Momento de la captura de requerimientos

En el enfoque tradicional, los requerimientos son definidos al principio del proyecto. El caso particular del PUD, que es un proceso definido con un modelo de proceso iterativo e incremental, el flujo de trabajo de requerimientos tiene un gran peso en la fase inicial y de elaboración.

En el enfoque ágil, los requerimientos son relevados continuamente durante todo el proyecto. El inicio de la construcción del software se da en un contexto en el cual los requerimientos no están completamente definidos, sin embargo, se tiene una visión.

Contenedor y persistencia de los requerimientos

El enfoque tradicional, la ERS contiene los requerimientos a lo largo del ciclo de vida del producto todo el tiempo. En caso de que surjan cambios, es necesario actualizar la ERS. En definitiva, es el documento de más alto nivel que define las características del sistema.

En el enfoque ágil, se utiliza al Product Backlog como un contenedor transitorio de los requerimientos, ya que una vez implementados estos se “persisten” en el producto funcionando. El detalle de valor queda persistente en los casos de prueba que se construyen en el Testing.

Tipos de requerimientos

La parte baja de la pirámide, corresponde a los requerimientos del Dominio de la solución (esto no le interesa al cliente, es cómo el equipo transforma sus necesidades en código ejecutable). Lo que está por encima, es decir, Requerimientos de Usuario y Requerimientos de Negocio, corresponden al Dominio del Problema, y es lo que al cliente sí le interesa.

A nivel de la gestión ágil de requerimientos se trabaja con:

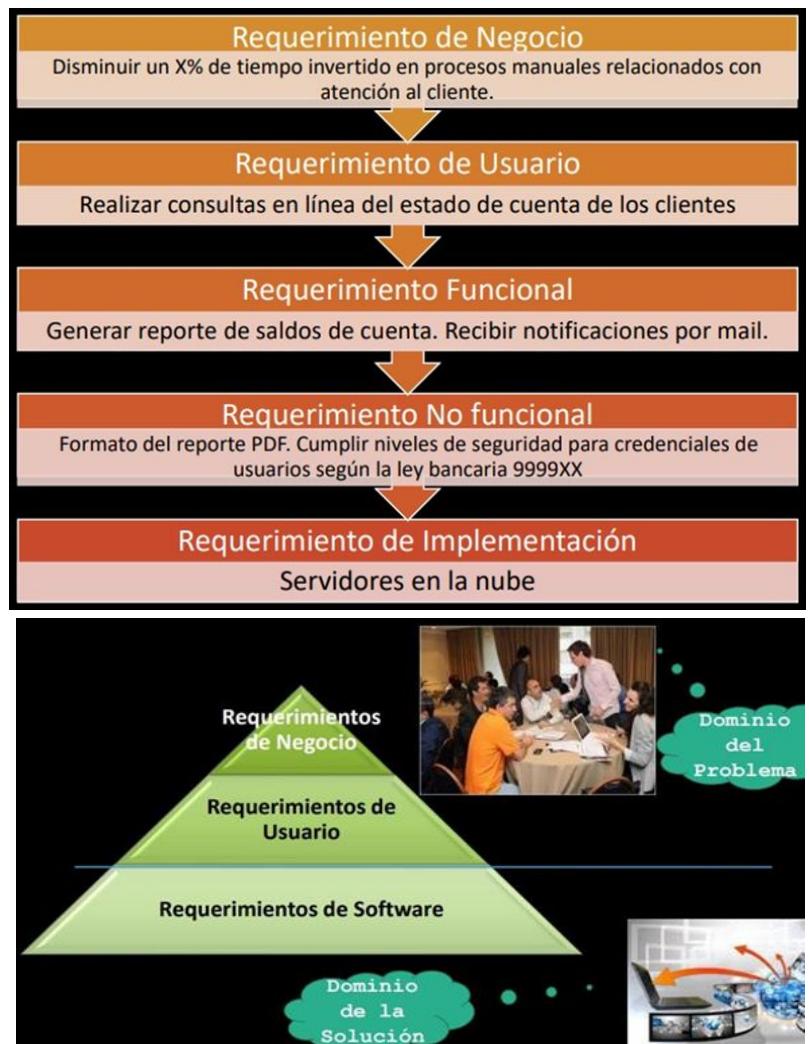
- Requerimiento de negocio
- Requerimiento de usuario

Las User Stories son requerimientos de usuario alineados a un requerimiento de negocio.

Al usuario le interesa el requerimiento de negocio. El valor de negocio es el requerimiento de negocio, pero el requerimiento de usuario es el medio por el cual vamos a lograr ese valor de negocio.

Las User Stories no sirven para especificar requerimientos de software, sirven para identificar requerimientos de usuario.

El foco de la gestión ágil de requerimientos está puesto sobre el usuario y sobre sus necesidades y por eso en enfoques como scrum quién debe trabajar para identificar las User Stories es el Product Owner. Ya que es él quien se concentra en las necesidades de usuarios que van a satisfacer las necesidades de negocio.



¿Qué debemos asumir?

- Los cambios son la única constante. Todo el tiempo va a haber cambios en los requerimientos.
- Stakeholders: no son todos los que están. Debemos saber quiénes son los involucrados y los involucrados son todos los que tiene algo que decir respecto del producto. El PO es el representante de todas estas personas y el que se comunica con todas ellas para luego ser él mismo el que se comunica con el equipo que hace el software.
- Siempre se cumple eso de que: “El usuario dice lo que quiere cuando recibe lo que pidió”.
- No hay técnicas ni herramientas que sirvan para todos los casos. Debemos detectar para cada situación cuál es la herramienta más adecuada.
- Lo importante no es entregar una salida, un requerimiento, lo importante es entregar, un resultado, una solución de “valor”.

Principios ágiles relacionados con requerimientos ágiles



1- LA PRIORIDAD ES SATISFACER AL CLIENTE A TRAVÉS DE RELEASES TEMPRANOS Y FRECUENTES (2 SEMANAS A UN MES)



2 -RECIBIR CAMBIOS DE REQUERIMIENTOS, AUN EN ETAPAS FINALES



4 - TÉCNICOS Y NO TÉCNICOS TRABAJANDO JUNTOS TODO EL PROYECTO



6 - EL MEDIO DE COMUNICACIÓN POR EXCELENCIA ES CARA A CARA



11 - LAS MEJORES ARQUITECTURAS, DISEÑOS Y REQUERIMIENTOS EMERGEN DE EQUIPOS AUTOORGANIZADOS

User Stories

Son una técnica para trabajar requerimientos de usuario en ambientes ágiles. La US contiene una descripción corta de una funcionalidad que se espera del producto valuada por un usuario del sistema.

Funciona como un recordatorio para el equipo de desarrollo y el Product Owner de que tienen que conversar acerca de la necesidad en términos de negocio para la construcción de una característica del producto que aporte valor. En otras palabras, es un mecanismo para diferir una conversación.

Es un requerimiento a nivel de usuario, no al nivel de sistema, por lo que se encuentra a un nivel de abstracción más elevado. Es un ítem de planificación que no acompaña al producto durante todo el ciclo de vida. Para eso está el manual de usuario. Las US son escritas por el Product Owner.

Internamente, las User Stories se consideran verticales dentro del producto, ya que pueden incluir aspectos desde diseños de interfaces hasta diseños de tablas en la base de datos.

Se dice que son multipropósito, debido a que cumplen diferentes funciones en el desarrollo de un software:

- Representan una necesidad de usuario, de forma no detallada.
- Conforman una descripción del producto.
- Son utilizadas como ítems de planificación.
- Se utilizan como recordatorios de futuras conversaciones acerca del producto que se desea construir.
- Junto con el código ejecutable, permiten documentar el producto.

La problemática que motivó en la definición de esta técnica es la dificultad de comunicar lo que hará el sistema a todas las personas afectadas. Cada una tiene un interés respecto a esta información.

La definición de los requerimientos es lo que más afecta la calidad del sistema final y es lo más complejo de definir con detalle debido a la incertidumbre y cambio constante que sufren, que se da por el problema de comunicación. La dificultad se basa en qué se escribe y cuándo.

La elicitation tradicional no da buenos resultados, ya que todas las decisiones se toman al principio del proyecto cuando menos información se tiene. Por lo que se estableció que esta toma de decisiones se traslada a lo largo del proyecto, a medida que la incertidumbre baja y la información sube, buscando que se obtenga más a menor tiempo posible. Aquí surgen las User Stories.

Partes de la User Story

Está compuesta por tres partes:

1. **Tarjeta:** es una descripción de la historia, utilizada para planificar y como recordatorio. Tiene una sintaxis particular que es la siguiente:

<<Frase Verbal>>

Como <<nombre del rol>> yo puedo <<actividad>> de forma tal que <<valor de negocio>>

- Nombre del rol: representa quien está realizando la acción o quien recibe el valor de la actividad.
- Actividad: representa la acción que realizará el sistema.
- Valor de negocio: es la más importante y representa el por qué es necesaria la actividad. Esto justifica la razón de que el Product Owner escriba las US. Permite priorizar el desarrollo.
- Frase verbal: no es obligatoria, es una forma corta de referenciar la US. Representa la funcionalidad que se expresa en la User Story.
- Criterios de aceptación: son diferentes criterios sobre la actividad que se necesita implementar, que permiten definir los límites de la U.S. para que el equipo tenga una mejor visión de esta. Además, facilitan las tareas de desarrolladores y testers para probar la funcionalidad.

Se deben redactar independientes a la implementación (alto nivel de abstracción).

- Pruebas de aceptación: acompañan a la tarjeta en la parte trasera, y es donde se capturan todos los detalles (que se manifiestan en la conversación) de la User, permitiendo determinar cuándo una historia está completa. Esto servirá a los desarrolladores para probar si la implementación ha sido realizada de forma correcta y completa. Las pruebas pueden agregarse o quitarse en cualquier momento.
2. **Conversación:** Es la parte más importante de la US, ya que explicita la comunicación entre el Product Owner y el Equipo de desarrollo que permite compensar la falta de especificación y detalle. El acuerdo por sobre la negociación del contrato y el principio de técnicos y no técnicos trabajando juntos durante todo el proyecto están asociadas a esta parte de la US.

El Product Owner es parte del equipo, por lo cual se espera responsabilidad, compromiso y confianza de su parte para hacerse cargo y llegar a un acuerdo común con el equipo sobre lo definido acerca de la funcionalidad.

También es posible dejar la conversación persistente en grabaciones, minutas o softwares de gestión que posibiliten la conversación. De todas maneras, la conversación se materializa de forma distribuida en la tarjeta y en las pruebas de aceptación.

En simples palabras, son discusiones acerca de la historia que sirven para desarrollar los detalles de la historia.

3. **Confirmación:** Definición de un acuerdo entre el Product Owner y el Equipo para demostrar que la característica del producto realmente se implementó. Conforma a la definición de las pruebas de aceptación. El PO decide si las pruebas de aceptación son válidas. Pruebas que se usan para determinar cuándo una US está completa.

Ventajas

- Prioriza la comunicación verbal por sobre la escrita (recordar la conversación es la parte más importante).
- Son comprensibles tanto para el cliente/usuario (Product Owner) como para el equipo de desarrollo (recordar que se escriben en términos de negocio).
- Tienen el tamaño justo para permitir planificar.

- Permiten el desarrollo iterativo al dividir todas las funcionalidades del producto en US.
- Permiten desestimar los detalles, hasta que se logre una mejor comprensión sobre cuál es la necesidad concreta de los usuarios.

Desventajas

- En proyectos grandes, es difícil entender la relación entre U.S (son independientes entre sí).
- No son apropiadas para equipos demasiado grandes (por el tema de priorizar la conversación verbal, por sobre la escrita).

Product Backlog y las User Stories

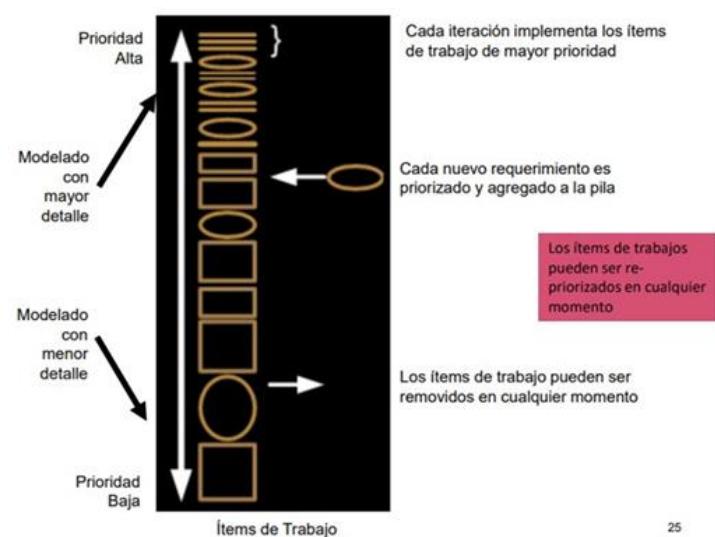
Es una pila de PBIs (Product Backlog Items) con sus correspondientes estimaciones de tamaño, en un formato determinado. En la parte más alta de la pila, se ubican las U.S. de mayor prioridad (que son aquellas U.S. con granularidad fina), y las de menor prioridad (normalmente aquellas User de granularidad gruesa, sobre las cuales no se tiene mucho detalle ya que no se consideran indispensables) al fondo. En cualquier momento se puede cambiar esa priorización, según cómo lo indique el Product Owner, quien es el encargado de realizar el proceso de priorización.

En cualquier momento del proyecto se van agregando nuevas U.S. y actividades a realizar, según se vayan detectando nuevos requerimientos.

Para quitar una U.S. del Product Backlog, se planifica una iteración y se decide, según la velocidad de trabajo del equipo [Story Points / iteración], cuántas U.S. de la pila (las de mayor prioridad que están arriba) se quitarán para implementarlas en la próxima iteración.

Debido a estas dos situaciones (agregar y quitar ítems), nunca encontraremos el 100% de los requerimientos de usuario en el Product Backlog.

El Product Owner Prioriza las historias en el Product Backlog



25

Porciones verticales

Las US son porciones verticales, es decir, para poder agregar valor al cliente debemos tener un poco de interfaz, un poco de lógica de negocio y otro poco de base de datos.

Debemos diseñar el pedacito de cada cosa que nos hace falta para que una determinada característica funcione.

Relación con la arquitectura en capas

La User es una porción vertical, la cual abarca todas las capas de arquitectura: capa de datos, lógica de negocio y capa de presentación. Esto permite entregar una característica terminada al cliente.

Tamaño

Es recomendable tener una gran cantidad de historias con tamaños relativamente pequeños, que tener historias muy grandes.

El tamaño de las U.S. se mide en Story Points.

Story 1	Story 2
GUI	
Business Logic	
Database	

Hay 3 aspectos que determinan el tamaño de una U.S.:

- Complejidad: hace referencia a una dificultad inherente a la funcionalidad, que no permiten dividirla en U.S. más pequeñas. Por ejemplo, algoritmos complejos, cálculos, gran cantidad de campos, muchas validaciones, demasiados filtros, etc.
- Esfuerzo: es el tiempo en horas de trabajo que requerirá la implementación.
- Incertidumbre
 - Técnica: cuando se tiene dudas en el dominio de la solución, por ejemplo, el uso de una cierta tecnología no muy conocida por el equipo.
 - De negocio: cuando hay incertidumbre respecto a cómo interactuará el usuario con el sistema, lo cual es causado por falta de conocimiento del dominio del negocio.

Story Points

Son unidades de medida relativas (no importa el valor en sí, sino su comparación), las cuales miden el tamaño de un producto, NO de una medida basada en el tiempo (por ejemplo, horas hombre). Esto se debe a que estimar basándonos en el tiempo, conduce a muchos errores.

Lo relativo surge a partir de que se comprobó que las personas no saben estimar en términos absolutos, sino que les resulta útil la comparación de tamaños.

Niveles de abstracción

1. **User Story:** cumple con el criterio de Listo (DoR) para ingresar a una iteración.

2. **Épica:** es una User muy grande. El tamaño es relativo a si entra en una iteración o no, ya que las User se empiezan y se terminan en la iteración. Si la User no entra en la iteración es considerado una épica. Tienen la misma sintaxis que la User.

Una épica se suele identificar también cuando no cumple con el modelo de calidad de INVEST. La épica debe ser dividida en US más pequeñas.

Sirven para aquellos requerimientos que no son prioridades en las primeras iteraciones, y basta con tener sólo una aproximación general de las funcionalidades requeridas.

Llegado el momento de su implementación, se deberán dividir en historias de usuario más pequeñas, para poder planificar su implementación en sucesivas iteraciones.

3. Temas: son un conjunto de User Stories agrupadas, debido a que conceptualmente están relacionadas. Se las agrupa para tratarlas como una entidad simple a la hora de estimar o planificar.

Ambos (Épicas y Temas) permiten reducir los esfuerzos del equipo a la hora de estimar.



Modelado de Roles

Se utilizan las **tarjetas de rol de usuario**, aquí se anotan las características en general que va a tener el usuario.

El tema crítico de la funcionalidad está relacionado con quien va a utilizar dicha funcionalidad. Por esta razón existe una tendencia por ocuparse de quien va a usar el sistema, especificando los roles de usuario donde definimos las características del usuario respecto al producto y en relación con su perfil genérico.

Técnicas adicionales

1. Personas: esta técnica busca ser más específica acerca de la descripción de la persona concreta que va a utilizar el sistema. Este análisis es exhaustivo y por lo tanto complejo. Si los roles son clases, las personas serían los objetos.
2. Personajes extremos: que personas en términos extremos podrían llegar a utilizar el software.
3. Proxies (Usuarios Representantes): es un representante de todos los usuarios. Debe tener capacidad de tomar decisiones desde el negocio.

Criterios de aceptación

Es una nota no obligatoria que define un acuerdo respecto a cómo se debe comportar el software para que el PO acepte la implementación, es recomendable que este debido a que sino los desarrolladores no tendrían un parámetro para definir si la misma fue cumplimentada o no. Lo fundamental es que esté escrito en términos objetivos y concretos.

El criterio de aceptación es información concreta que tiene que servirnos a nosotros para saber si lo que implementamos es correcto o no. Debemos saber si el PO nos va a aceptar esta característica implementada o no, porque la va a aceptar si respeta los criterios de aceptación.

- Definen límites para una User Story (US)
- Ayudan a que los PO respondan lo que necesitan para que la US provea valor (requerimientos funcionales mínimos).
- Ayudan a que el equipo tenga una visión compartida de la US.
- Ayudan a desarrolladores y testers a derivar las pruebas.
- Ayudan a los desarrolladores a saber cuándo parar de agregar funcionalidad en una US.

¿Cuáles son los criterios de aceptación buenos?

- Definen una intención, no una solución. Ejemplo: El usuario debe elegir al menos una cuenta para operar.
- Son independientes de la implementación.
- Relativamente de alto nivel, no es necesario que se escriba cada detalle.

¿Y los detalles, a dónde van?

Detalles como:

- El encabezado de la columna se nombra “Saldo”
- El formato del saldo es 999.999.999,99
- Debería usarse una lista desplegable en lugar de un Checkbox.

Estos detalles que son el resultado de las conversaciones con el PO y el equipo puede capturarlos en dos lugares:

- Documentación interna de los equipos

- Pruebas de aceptación automatizadas.

Cada equipo decide dónde guardar el detalle y cómo lo va a mantener.

Pruebas de aceptación

Son declaraciones de intención de que hay que probar. El conocimiento del negocio está inmerso en las pruebas de aceptación. Se prueban escenarios exitosos y escenarios que fallen. En las pruebas de aceptación se encuentran las más importantes, ya que **no es posible escribir todas las posibles pruebas de aceptación**.

El Product Owner acepta la User como implementada si todas las pruebas de aceptación se cumplen.

Expresan detalles resultantes de las conversaciones entre clientes y desarrolladores; suelen usarse para completar detalles de la US. Es importante mencionar que las mismas deben escribirse antes que la programación empiece, y que se recomienda que las escriba el cliente.

DoR (Definition of Ready) – Definición de Listo

Es una medida de calidad que **determina si la User Story está en condiciones de entrar a una iteración de desarrollo**. Para que la US pueda ser implementada, **mínimamente debe satisfacer el INVEST Model**. Entre el equipo de desarrollo y el PO pueden definir características o condiciones extras al INVEST para que una US se considere lista para entrar a la iteración de desarrollo.

INVEST Model

Es un modelo de calidad que nos ayuda a verificar si la User cuenta con las características de la calidad mínima para satisfacer la definición de Ready y poder ingresar en una iteración de desarrollo.

- **Independiente:** la User es calendarizable e implementable en cualquier orden. Esto implica que no existe una dependencia con otra User para desarrollarla o para mostrarla al Product Owner. El PO tiene la libertad de priorizarlas en el Product Backlog como él deseé.

- **Negociable:** Las US no son contratos estrictos, sino más bien pequeñas descripciones de funcionalidades donde se describen las necesidades de los usuarios, y los detalles se dejan para negociar en conversaciones futuras entre clientes y el equipo de desarrollo. Esto implica que no deben detallarse todos los aspectos de forma exhaustiva, sino más bien, servir a modo de recordatorio para negociar aquellos aspectos en un futuro cuando se deba implementar.

El cliente debe expresar lo que necesita, es decir que está centrada en el “qué” no el “cómo”. El cómo es decisión del equipo y debe ser negociable con el Product Owner.

La US debe estar escrita en términos de qué necesita el usuario y no de cómo lo vamos a implementar.

- **Valuable:** Una US debe ser valorable para los clientes o usuarios (el para qué de la descripción). Es decir, las U.S. deben aportar valor de negocio a quiénes estará destinado el producto, no para el equipo de desarrollo. Esto implica dejar de lado en US, por ejemplo, las tecnologías utilizadas para el desarrollo del producto.

- **Estimable:** la US debe contener la cantidad de información suficiente para estimar el tamaño, la complejidad y el esfuerzo necesario para realizarlo. Sin poder estimarla no es posible determinar si se puede finalizar en una iteración por lo tanto no puede ser construida (No puedo saber cuánto tardaría).

Hay tres razones por las cuales una US puede no ser estimable:

1. El equipo tiene incertidumbre acerca del dominio.
 - a. Deberán discutir con el Product Owner para esclarecer todas las dudas.
2. El equipo tiene incertidumbre técnica, es decir, sobre el dominio de la solución.
 - a. Deberá destinarse uno o más miembros del equipo a uno o más Spikes.
3. La US es demasiado grande.

- a. En este caso, deberá dividirse en US más pequeñas.
- **Small (Pequeña):** la US debe ser lo suficientemente pequeña como para ser finalizada en una iteración. Se debe comenzar y terminar una característica en una iteración, no debe haber trabajos a medias. El tamaño es relativo. ya que depende del equipo, de la experiencia que tiene, de su capacidad de trabajo y también depende de la duración de la iteración. Ágil aplica gestión binaria, es decir, si se puede completar la US en la iteración, es pequeña. No aplica porcentaje de avances.
- **Testable:** la US debe cumplir con las pruebas de aceptación para poder probar si la misma fue o no implementada (se muestra que está hecha y demostrable).

DoD (Definition of Done) – Definición de Hecho

Determina si la US está terminada. Son los criterios que establecen cuándo una User Story ha sido implementada de forma correcta y completa, de forma que puede ser mostrada al Product Owner, para que éste decida si se pasa a producción o no. En el momento que una U.S. cumple con todo lo pactado en esta Definición de Hecho, se considera que esa US está lista para ser desplegada y aportar valor al cliente.

Esto permite crear transparencia, debido a que proporciona a todo el equipo una comprensión exhaustiva de qué trabajo se realizó, para lograr ese incremento.

En la práctica consta de un Checklist de condiciones que deben cumplirse. Puede ocurrir que una organización tenga un estándar de DoD, entonces los miembros del equipo deberán adaptarse a ese estándar. O bien, que la organización no posea un estándar, con lo cual el equipo deberá confeccionar y crear una Definición de Hecho adecuada para el producto.

Spikes

Son un tipo especial de US que **se producen por la incertidumbre** que la misma presenta, la cual imposibilita que pueda ser estimada y por lo tanto no cumple con la definición de listo. Una vez resuelta la incertidumbre, la Spike se convierte en una o más US. Es una característica más del producto a la cual el equipo le tiene que dedicar tiempo para:

1. Familiarizarse con una nueva tecnología o dominio.
2. Investigar y prototipar para ganar confianza frente a:
 - a. Riesgos tecnológicos.
 - b. Riesgos funcionales, donde no está claro cómo debe reaccionar el sistema para satisfacer las necesidades del usuario.

Se usan para quitar el riesgo e incertidumbre de una US y otra faceta del proyecto. Se clasifican en técnicas y funcionales y pueden utilizarse para:

1. Inversión básica para familiarizar al equipo con una nueva tecnología o dominio.
2. Analizar el comportamiento de una historia compleja y poder dividirla en piezas manejables.
3. Ganar confianza frente a riesgos tecnológicos, investigando o prototipando.
4. Ganar confianza frente a riesgos funcionales, donde no está claro cómo el sistema debe resolver la interacción con el usuario para alcanzar el beneficio esperado.

Spikes técnicas

Utilizadas para evaluar diferentes enfoques técnicos y tecnológicos en el dominio de la solución. Están asociados a la implementación de la funcionalidad. Nueva tecnología.

Cualquier situación en la que el equipo necesite una comprensión más fiable antes de comprometerse a una nueva funcionalidad en un tiempo fijo. Dependen de los técnicos (nosotros).

Spikes funcionales

Utilizada cuando hay cierta incertidumbre respecto de cómo el usuario interactúa con el sistema, usualmente son mejor evaluadas con prototipos para obtener retroalimentación de los usuarios involucrados.

Dependen de definiciones del negocio, del PO.

Lineamientos para Spikes

- Estimables, demostrables, y aceptables.
- La excepción, no la regla.
 - Toda historia tiene incertidumbre y riesgos.
 - El objetivo del equipo es aprender a aceptar y resolver cierta incertidumbre en cada iteración.
 - Los spikes deben dejarse para incógnitas más críticas y grandes.
 - Utilizar spikes como última opción.
- Implementar la spike en una iteración separada de las historias resultantes.
 - Salvo que el spike sea pequeño y sencillo y sea probable encontrar una solución rápida en cuyo caso, spike e historia pueden incluirse en la misma iteración.



DIFERIR EL ANÁLISIS
DETALLADO TAN TARDE
COMO SEA POSIBLE, LO QUE
ES JUSTO ANTES DE QUE EL
TRABAJO COMIENCE.



LAS USER STORIES NO SON
REQUERIMIENTOS DE
SOFTWARE, NO NECESITAN
SER DESCRIPCIONES
EXHAUSTIVAS DE LA
FUNCIONALIDAD DEL
SISTEMA.

Investment Themes (Temas de Inversión)

Representan un conjunto de iniciativas o propuestas de valor que conducen la inversión de la empresa en sistemas, productos, aplicaciones y servicios con el objetivo de lograr una diferenciación en el mercado y/o ventajas competitivas.

Los THEMES son una combinación de inversión en:

1. Inversión en ofertas de productos existentes, mejoras, soporte y mantenimiento.

2. Inversión en nuevos productos y servicios que mejorarán los beneficios y/o ganarán porciones de mercado en el período actual o al corto plazo.
3. Inversión a futuro en productos y servicios avanzados.
4. Inversión hoy, pero que no dará beneficios hasta dentro de unos años.
5. Inversión en reducción (estrategia de retiro) para ofertas existentes que están cerca del final de su vida útil.

Estimación en ambientes ágiles

Estimar es el proceso de obtener una aproximación sobre una medida con el objetivo de predecir la completitud y gestionar los riesgos en el contexto de un proyecto. Se relaciona con los objetivos del negocio, compromisos y control.

Según McConnell una estimación en el contexto del software es una predicción de cuánto tiempo durará o costará un proyecto. Es considerada una de las actividades más complejas en el desarrollo de software.

Las estimaciones no son planes ni compromisos, sino que sirven de base para planificar, pero el plan final no tiene por qué coincidir con lo estimado.

Errores en las estimaciones

1. Información imprecisa acerca del software a estimar o acerca de la capacidad para realizar el proyecto.
2. Demasiado caos en el proyecto (mal definido el proyecto).
3. Imprecisión generada por el proceso de estimación.
4. Una de las fuentes más común es omitir actividades necesarias para la estimación del proyecto tales como, requerimientos faltantes, licencias, reuniones, revisiones, etc.
5. Se olvidan del detalle que planear no es estimar y estimar no es planear, sino que las estimaciones son la base de los planes, pero los planes no tienen que ser lo mismo que lo estimado.
6. A mayor diferencia entre lo planeado y lo estimado, mayor el riesgo.
7. Las estimaciones no son compromisos.

¿Por qué estimamos?

Existen dos razones principales por las cuales se llevan adelante las estimaciones:

- Para predecir completitud.
- Para administrar riesgos.

Antes de que el proyecto comience, el líder del proyecto y el equipo de software deben estimar el **trabajo** que habrá de realizarse, los **recursos** que se requerirán y el **tiempo** que transcurrirá desde el principio hasta el final. Las estimaciones requieren:

- Experiencia.
- Acceso a buena información histórica (métricas).
- El valor para comprometerse con predicciones cuantitativas cuando la información cualitativa es todo lo que existe.

Siempre se desea saber muy al inicio del proyecto cuál será el costo, el tiempo que llevará, entre otros aspectos y como aún no se conoce con precisión cuál será el alcance del producto resulta complejo predecir con exactitud estos aspectos, es por ello que a medida que avanza el proyecto, las estimaciones son más certeras, dado que se cuenta con

mayor información sobre la cual realizar las estimaciones. (Es por ello que se suele decir que típicamente la primera estimación difiere hasta un 400%).

Al inicio del desarrollo hay una estimación errónea de 2 a 4 veces más, esto se debe a dos aspectos:

1. Al inicio del proyecto tenemos poca información y demasiada incertidumbre, lo que aumenta el riesgo de generar estimaciones erróneas.
2. Las estimaciones son de naturaleza optimista, por eso las estimaciones deben ser recalculadas conforme se avanza con el proyecto.
3. Se suele decir: “el que no sabe estimar, estima muchas veces, y el que no sabe planificar, planifica muchas veces”.

Proceso de estimación

Para ver cómo y qué se estima en el ambiente tradicional ver el punto de estimación en el título “[Plan de proyecto](#)” en la unidad 1 de este apunte.

Métodos utilizados para estimar

Se recomienda utilizar diferentes métodos de estimación y luego contrastar (comparar). Todos tienen un “Factor de Ajuste” que permite que el método cierre (calibraciones).

1. Basados en la experiencia

- a. **Datos históricos:** consiste en recolectar datos básicos de otros proyectos para ir generando una base de conocimientos que sean de utilidad para futuras estimaciones, con esto se busca una alternativa en la cual el conocimiento de cada individuo se transfiere a la organización y no estamos atados a una única persona. Se deben utilizar para no tener problemas entre desarrolladores y clientes.
Esto no va acorde a la gestión ágil, que dice que la experiencia no se puede extrapolar a otros equipos.
 - i. **Industry Data:** datos de otras organizaciones que aportan al mercado y desarrollan productos con algún grado de semejanza y que permite una comparación básica.
 - ii. **Historical Data:** datos de la organización de proyectos que se desarrollaron y ya se cerraron.
 - iii. **Project Data:** datos del proyecto, pero de etapas anteriores a la que se está estimando.
- b. **Juicio experto:** es uno de los más utilizados, el problema se presenta cuando se utiliza como única técnica de estimación el juicio experto puro. Existen dos técnicas para aplicarlo:
 - i. **Juicio experto puro:** un experto estudia las especificaciones y realiza una estimación. Esta estimación se basa fundamentalmente en la experiencia del experto. (el problema es que, si el experto se va de la organización, esta pierde su capacidad de estimación).
Se puede utilizar el “método de optimista, pesimista, habitual”, en el que se le pregunta al experto sobre 3 valores distintos y se aplica una fórmula para calcular la estimación.
 - ii. **Juicio Wideband Delphi:** es similar al anterior pero grupal, un grupo de personas se reúne con el objetivo de converger a una estimación común, tanto en esfuerzo como en duración. Hay un coordinador que va a reunir las estimaciones individuales de cada uno de los expertos cuando les dieron las especificaciones, se opina sobre las estimaciones ajenas de manera anónima, se vuelve a discutir, revisan, las vuelven a enviar al coordinador y se repite hasta que todos estén de acuerdo de forma razonable. Este método es tomado como base para el Poker Planning.

2. Analogía

Se comparan los factores a estimar de forma relativa con respecto a las estimaciones de otros proyectos similares. Es un método que tiene mucho error dado que el conocimiento en un proyecto no es extrapolable a otro. Es decir, consiste en tomar a un proyecto e ir comparando factores, por ejemplo:

- Tamaño: es menor o mayor al proyecto anterior.

- Complejidad: más complejo de lo usual.
- Usuarios: si hay más usuarios habrá más complicaciones, como la concurrencia o los datos a almacenar.

3. Otros

- Basados exclusivamente en los recursos:** consiste en analizar el personal que se tiene y de cuánto tiempo se dispone del mismo, haciendo las estimaciones exclusivamente en función de este factor. En la realización: "El trabajo se expande hasta consumir todos los recursos disponibles, independientemente si se alcanzan o no los objetivos."
- Basados en la capacidad del cliente o exclusivamente en el mercado:** ponen el foco en el cliente y realizan las estimaciones en función de su capacidad de pago. En este método no se evalúa al producto que se está intentando vender, sino a quien se lo vende. A veces se cobra menos para que nos elijan antes que a la competencia.
- Basados en los componentes del producto o en el proceso de desarrollo:** se pueden aplicar 2 enfoques:
 - Bottom-up:** se descompone el producto en unidades lo más pequeñas posible. Se estima cada unidad y luego se hace la estimación total.
 - Top-down:** se descompone el producto en grandes bloques y luego se estima cada uno de los componentes.
- Métodos algorítmicos:** se ejecutan algoritmos que tienen como parámetros de entrada medidas cuantitativas de tamaño, complejidad y conocimiento de dominio. Cada variable tendrá asociada un factor multiplicativo y en función de esto se obtiene como resultado un valor de esfuerzo. Es importante destacar que, a pesar de ser un cálculo matemático, puede no ser certero igual.

Problemas de estimaciones tradicionales

- Estimar demasiado pronto.
- Resistencia a las reestimaciones.
- Estimaciones que buscan precisión.
- Estimaciones hechas por gente distinta a la que hace el trabajo.
- Estimaciones que pretenden ser absolutas (inflexibles).

Estimaciones en ambientes ágiles

Son presunciones numéricas asociadas a una probabilidad de que cierto evento ocurra. La filosofía ágil plantea que quien estima, es quien debe hacer el trabajo, con lo cual es el equipo de desarrollo quien realiza las estimaciones, de forma colectiva (menos el Product Owner, no participa en estimaciones). Al ser de esta manera, la estimación se ajusta mejor, debido a que se generan debates entre los miembros del equipo, lo cual puede abrir puntos de vista no detectados por otros, haciendo que ese proceso de estimar se vuelva lo más preciso posible (a diferencia de las metodologías tradicionales, donde se estima individualmente y se depende sólo de ese miembro).

Para estimar de esta forma, es fundamental que los miembros del equipo tengan una actitud en escucha, donde todos puedan opinar libremente y no haya influencias que "tapen" la opinión de otros.

Frente a la posibilidad de estimar mal y generar enojo por parte del cliente, tenemos dos reacciones posibles:

Como la estimación es un trabajo a riesgo, se subestima, o sea, simplificando sin considerar todo el trabajo que hay que hacer, por miedo a que si el valor es más alto el cliente lo rechace.

Sobreestimar, cuando me preguntan por ejemplo cuantas horas me va a llevar un trabajo, y, sabiendo que me va a llevar 2 horas, digo 4 por las dudas pase algo en el medio, inflando muchísimo la complejidad del producto.

Las estimaciones se tienden a confundir con compromisos o planificación. Uno planifica basándose en la estimación, pero no son lo mismo.

No es una instancia única, ya que a medida que se avance el proyecto, más información vamos a tener y más eficiente van a ser nuestras estimaciones.

Teniendo en cuenta esto, hay que aceptar que la estimación no es fácil, de hecho, es lo más difícil luego de lalicitación.

El gran beneficio del proceso de estimación es justamente “hacerlas”. Al estimar, se enriquece el trabajo del equipo, lo cual mejorará futuros procesos de estimaciones. Además, este proceso ayuda a determinar la factibilidad del trabajo planificado en etapas tempranas.

En el enfoque ágil, el esfuerzo destinado para estimar es bajo, pero amparándose en el concepto de que, al destinar poco esfuerzo en la estimación, se logra acercarse bastante a la realidad, la cual no sigue una relación lineal: si destinamos demasiado esfuerzo a este proceso, no obtendremos una precisión casi perfecta. Es por esto, que, en este enfoque, el tiempo de estimación se define y asigna con anterioridad para no utilizar más esfuerzo del necesario.

Hay que recordar que el remedio, debe ser más barato que la enfermedad: el objetivo es estar en alguna parte de la izquierda del gráfico, donde con poco esfuerzo, se alcanzan grandes valores de certeza en la estimación. “La precisión es cara”.

Existe una serie de consideraciones importantes:

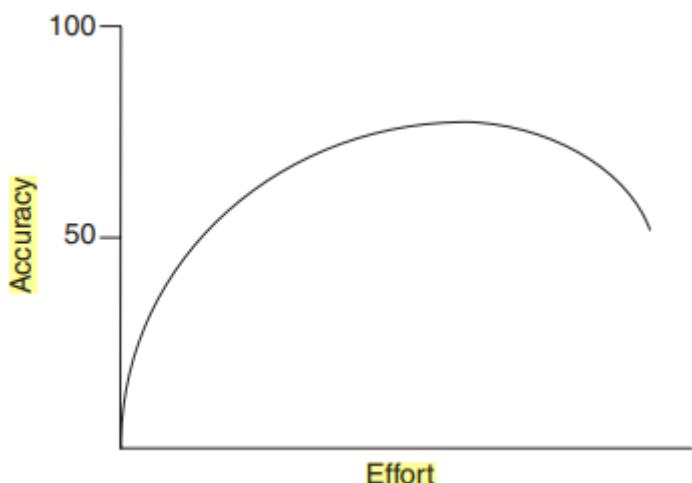
- Son medidas relativas no absolutas.
- No es una medida basada en el tiempo.
- Las personas no saben estimar en términos absolutos, somos buenos comparando.
- Comparar es generalmente más rápido.
- Se obtiene una mejor dinámica grupal y pensamiento de equipo más que individual.
- Se emplea mejor el tiempo de análisis de las Stories.

Hay una fuerte propuesta de que estime la misma persona que va a realizar el trabajo. Se debe tener la mente abierta para escuchar lo que el resto del grupo tiene para decir y no cerrarse a nada. Las estimaciones son un proceso y uno tiene que hacer foco en este proceso.

Lo ideal es comenzar con una estimación que se pueda revisar todas las veces que sea necesario. Recordando el principio de que “la mejor métrica de progreso es el software funcionando”, debemos entender que todo lo definido en cuanto a las estimaciones tiene que ver con el producto.

No se puede comparar una User Story con otra porque hay distintas dimensiones y ahí es donde entra el Sory Point, que hace de unidad homogeneizadora que permite comparar.

Para las estimaciones se deben tener en cuenta tres elementos a la hora de estimar: complejidad, esfuerzo e incertidumbre.



Estas tres variables son las que conforman al Story Point y de esta manera podemos comparar las US.

¿Qué se estima?

Lo que se estima en ambientes ágiles, es solo el tamaño de las US (o Features) usando como unidad de medida los Story Points, y teniendo en cuenta 3 aspectos de estas: incertidumbre, complejidad y esfuerzo. Como la complejidad de las US tiende a incrementarse exponencialmente, se utilizan comúnmente series numéricas de esta índole, siendo una de las más comunes, la serie de Fibonacci.

Al ser estimaciones de tamaños relativos, (es decir, se estima realizando comparaciones de los tamaños de las US) lo importante en este proceso no es el valor que se asigna a cada US en particular, sino más bien que las estimaciones sean consistentes en el conjunto. Es decir, para un equipo, una US de 8 puntos puede no significar lo mismo que para otro equipo, pero lo que importa es que en un mismo equipo, una US de 5 puntos, siempre sea más chica que una de 8.

Es importante tener en claro que no se estima tiempo, ya que la duración de los Sprints es fija (concepto de Timebox). Por esto, es que no se recomienda estimar el tamaño de las US en horas, para no confundir con el esfuerzo.

Tamaño vs. Esfuerzo

Como el esfuerzo (horas de trabajo) es el factor más significativo del costo de un proyecto, tenemos una gran tendencia a confundir tamaño con esfuerzo y esfuerzo con calendario. La complejidad es independiente de quien haga el proyecto, y el esfuerzo depende específicamente de la persona que lo va a llevar a cabo, de su capacitación, conlleva horas lineales (horas únicamente de trabajo, sin tener en cuenta almuerzo, baño, leer emails, etc. Para una jornada de trabajo de 8 horas, usualmente solo 5.5 o 6 son lineales). Por otro lado, no se debe confundir esfuerzo con calendario, ya que yo puedo decir que puedo tener un trabajo para el viernes, pero en el medio pueden suceder millones de cosas que evitan que cumpla mi objetivo. Un trabajo puede requerir un esfuerzo de x horas, pero no se puede asegurar que para tal día va a estar terminado.

Velocidad

Es una métrica del progreso de trabajo de un equipo. Se calcula (no se estima) sumando el número de Story Points asignados al estimar una US, que el equipo completa durante cada iteración, lo cual implica que la US se ha desarrollado y testeado por completo (DoD), y además el Product Owner la ha aceptado.

Si la US se completó parcialmente, no se cuentan sus Story Points en esa iteración (no existe el concepto de casi-completo, es una gestión binaria, es decir, se completó o no se completó la US). Esto genera que la velocidad del equipo baje en esa iteración, pero en la siguiente, aumentará, debido a que en unos pocos días probablemente se complete la US pendiente, y se pueda continuar con otras US planificadas en la iteración.

Esto implica que el equipo no se enfoque en la velocidad de cada iteración, sino más bien, en el promedio de velocidades a lo largo de sucesivas iteraciones. Esto brindará una mayor certeza acerca de la medida de progreso de trabajo del equipo.

Esta métrica permite retroalimentar el proceso de estimación, ya que un equipo maduro y con mucho trayecto, puede permitirse estimar cuántos puntos de historia puede desarrollar en la siguiente iteración a través del cálculo de velocidad.

Reestimación

Estimar nuevamente una US es aconsejable sólo cuando la opinión acerca del tamaño relativo de una US ha cambiado, debido a que en el proceso de estimación se han detectado otras U.S. con tamaños más grandes o pequeños. (esto tiene que ver con mantener la consistencia de tamaños a la hora de estimar).

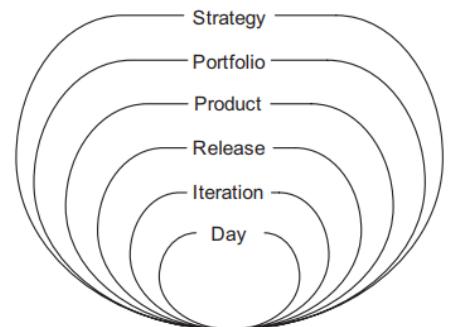
No se debería reestimar porque no se detecta un progreso como el esperado (velocidad del equipo), sino que se debe dejar que la velocidad a lo largo de sucesivas iteraciones equilibre esas inexactitudes.

Planificación

En ambientes ágiles, es necesario que quien planifica el proyecto de desarrollo de software, sea capaz de analizar el contexto del proyecto cada cierto período y hacer aquellos ajustes que sean necesarios, para encauzar los esfuerzos de manera correcta, y poder lograr los objetivos planteados.

“Planning is everything. Plans are nothing” → esto apunta a que lo importante de planificar, es el proceso en sí. No el hecho de realizar una documentación exhaustiva que respete determinados templates. Sí debe quedar registrado de alguna manera, idealmente bajo templates, pero sólo para servir de ayuda de memoria. Lo importante es el proceso.

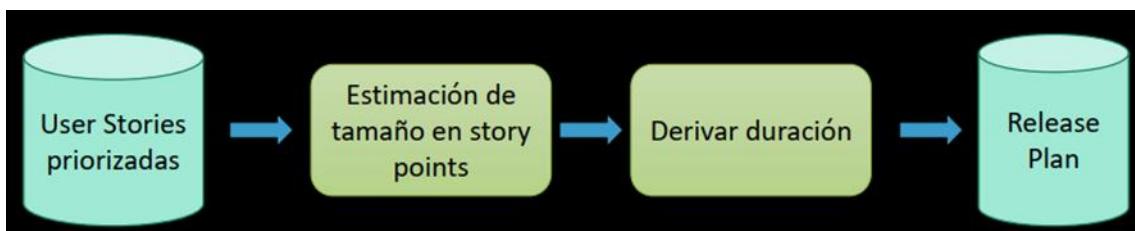
Los equipos ágiles afrontan esta situación, planificando al menos en 3 niveles:



Planificación de Release

Este proceso se da al inicio del proyecto, donde se analiza y determina qué US, épicas o Temas serán desarrollados en una nueva entrega (Release) del producto. Este plan se va actualizando mientras el proyecto transcurre (usualmente al comienzo de cada iteración), de manera que siempre refleja las expectativas actuales sobre qué se incluye en la entrega.

Una vez que se priorizan las US a implementar, se estima su tamaño con Story points, y teniendo en cuenta la velocidad de trabajo del equipo, se deriva la duración de la Release o del proyecto (la cantidad de sprints necesarios para obtener el release).



Planificación de Iteración

Este proceso se da al comienzo de cada iteración, donde el PO identifica aquellas US de mayor prioridad que deberán ser implementadas en la iteración, basándose en la iteración anterior ya finalizada. Esto se hace teniendo en cuenta el tiempo (que es una variable constante por ser metodología ágil) y la velocidad de trabajo del equipo (el cual se mantiene constante también).

Durante esta planificación, se discute sobre las tareas que serán necesarias para transformar una US en código funcional y testeado.

Planificación del día

Este proceso permite coordinar y sincronizar el trabajo del equipo, normalmente en reuniones diarias. Allí sólo se discuten las tareas a realizar durante el día, haciendo foco en las actividades que deberán ser desarrolladas para lograr cumplir con una tarea específica.

Tamaño

Dado que el tamaño es lo que se compara, lo definimos como la **medida de cuán compleja y grande es una US**, además de cuánto trabajo es requerido para hacer o completar una Feature/US. También el tamaño es función de la incertidumbre de dicha US, definida la incertidumbre como la falta de información.

El tamaño no es lo mismo que el esfuerzo (el “quien” define esta diferencia), ya que este último indica las horas humano lineal requeridas para el desarrollo de la US. Por lo tanto, **una US de mismo tamaño puede implicar esfuerzos diferentes en función de la persona que se esté evaluando**, debido a que el esfuerzo varía dependiendo de las habilidades, conocimientos, experiencia, familiaridad con el dominio, etc.

Formas de estimar el tamaño

Existen diferentes escalas para estimar el tamaño y siempre son por comparación, entre ellas:

1. Número del 1 al 10.
2. Tales de remeras: S, M, L: suele ser utilizado como escala de estimación para los ítems del Product Backlog.
3. Serie exponencial de base 2: 2, 4, 8, 16, 32, 64, etc.
4. Serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc.

Una vez definida la escala, no se cambia. En caso de cambiarla, también se debe cambiar el patrón de medición.

Story Point

Es una unidad de estimación que **especifica la complejidad, incertidumbre y esfuerzo** propio del equipo respecto a una US en términos relativos **respecto a otra US**. Es lo que le da la idea del “peso” de cada US, decide cuán grande/compleja es. Por lo general esa complejidad tiende a incrementarse exponencialmente.

- **Complejidad:** que tan compleja es implementar esa US. Relacionada a la cantidad de partes y relaciones entre las mismas posee la US.
- **Esfuerzo:** es importante que estime la persona que va a hacer el trabajo, ya que el esfuerzo es distinto entre dos personas. Son las horas persona lineales que se requieren para terminar la US.
- **Incertidumbre:** es cuánta información falta para ser más adecuados en la estimación. Existe una incertidumbre técnica que es más sobre si se sabe cómo llevar a cabo lo explicitado con respecto a la tecnología a implementar y ese tipo de cosas. Y tenemos incertidumbre del negocio, si entendemos que es lo que el cliente pide y cómo llevarlo a cabo.

¿Para qué se estima?

1. En un primer momento para determinar cuántas Users se puede comprometer el equipo de trabajo a terminar en una iteración.
2. Al finalizar la iteración, contamos los puntos de historia correspondiente a las Users aceptadas por el Product Owner y obtenemos la velocidad del sprint, la cual es una de las métricas fundamentales en ambientes ágiles.

Sin embargo, es necesario aclarar que la velocidad no es una estimación sino una medición propiamente dicha. Si un equipo alcanza una velocidad sostenible en el tiempo, es decir que mantiene la velocidad en las diferentes iteraciones, permite que el equipo obtenga previsibilidad, la cual ayuda a estimar en las siguientes iteraciones.

¿Cómo se estima la duración del proyecto?

Si la estimación se realiza utilizando Story Points para medir la complejidad relativa de las User Stories, para determinar la duración de un proyecto se realiza la derivación tomando el número total de story points de sus US y dividiéndolas por la velocidad del equipo.

Poker Planning o Poker Estimation

Es una técnica de estimación en ambientes ágiles publicada por Mike Cohn. Resulta de la conjunción de diferentes métodos como el de juicio experto, analogía y desagregación, basado en que 4 ojos ven más que 2. En contraposición al método de juicio experto tradicional, asume que las personas que desarrollan el producto deben ser aquellas que estimen. En otras palabras, el equipo estima su propio trabajo y los miembros opinan sobre lo mismo, donde se mezcla la experiencia y el conocimiento de cada uno con la posibilidad de compartir con el resto del grupo. Un detalle es que el Product Owner participa en la planificación, pero no realiza estimaciones, suele ser el moderador, pero no es necesario que sea así siempre.

Escala

Dado que la complejidad en el software incrementa de forma exponencial, se decide adoptar la serie de Fibonacci como escala para la determinación de los puntos de historia.

1. **Valor 1 o 1/2:** son los valores más bajos y se aplican a una funcionalidad pequeña.
2. **Valor 2- 3:** funcionalidad pequeña a mediana.
3. **Valor 5:** funcionalidad media.
4. **Valor 8: funcionalidad** grande, donde nos preguntamos si es posible dividir.
1. **Valor 13 o más:** necesariamente hay que dividir ya que no ingresa en el sprint.

Procedimiento

2. Se define una lista de actividades, módulos, casos de uso, Users stories, etc.
3. Se acuerda cuál de las anteriores es la que tiene **menor grado de incertidumbre y se la define como canónica** asignándole 1 o 2 story point (es la más simple). Esta User es utilizada como referencia para comparar. Los equipos avanzados podrían elegir canónicas de 3 SP, con el objetivo de tener un margen para hacia arriba y hacia abajo. Por esta razón, también él ½ se incluye en la serie.
4. Cada miembro elige una carta con el valor de la estimación del tamaño de la U.S. (comparado con la U.S. canónica y teniendo en cuenta los 3 aspectos nombrados) y se espera a que todos decidan.
5. Utilizando **juicio experto**, cada integrante del equipo de desarrollo expone su valor de estimación de forma simultánea, por medio de las cartas, dándolas vuelta a las que estaban boca abajo. En caso de que la estimación difiera, el mayor y el menor estimador justifican sus estimaciones para conocer sus opiniones. Luego el resto del medio aportan sus opiniones. Siempre la escucha es activa, para poder aprender.
6. En caso de que la primera vuelta existiese diferencias considerables, se realiza nuevamente la ejecución del paso 4.

7. Si las estimaciones no convergen, entonces se llega a una estimación intermedia o promedio.

Consideraciones

1. Para realizar una estimación **se debe tener en cuenta el criterio de Done** y no solo la tarea de programación, es decir todo lo que hay que hacer para terminar una US y poder lidiar con el problema de las actividades omitidas.
2. El tiempo para realizar las estimaciones, como cualquier otra actividad en los ambientes ágiles, es time boxing.
3. Las US por estimar son las menos posibles, en orden con el principio de diferir compromisos.

Ventajas

- Permite opiniones multidisciplinarias de todo el equipo, lo cual enriquece la estimación.
- Al generar debates y discusiones, permite lograr una mejor comprensión del dominio en los miembros del equipo.
- Estudios demuestran que reunir estimaciones individuales conducen a mejores resultados, que estimar de forma grupal.
- Es una actividad que resulta divertida para el equipo.

Estimaciones en Lean

En Lean, el cual es más extremo que las metodologías ágiles, asume que las estimaciones son opcionales y en algunas ocasiones pueden convertirse en un desperdicio dado que tienen una alta probabilidad de no coincidir con la realidad, dado que cualquier variable que se modifica, modificará a la estimación también.

Estimaciones en proyectos pequeños

Para estimar proyectos pequeños lo mejor es utilizar datos históricos propios de la organización, dado que en estos casos los datos históricos de la industria no se adaptan correctamente a estas situaciones. Las estimaciones en proyectos pequeños dependen en gran medida de las capacidades de los individuos que hacen el trabajo. El SEI propone el Personal Software Process para proyectos unipersonales o 2 personas, conocidos como "mini proyectos".

Diferencias con las estimaciones en ambientes tradicionales

1. Respecto a las tres dimensiones

En Agile se tiene fijo el tiempo (lo que dura la iteración) y los recursos afectados para trabajar, dejando variable el alcance. Por lo tanto, al estimar en estos ambientes se responde a cuánto se puede comprometer el equipo, qué cantidad de las características de producto se puede construir en la iteración. Se entiende, además, que en estos ambientes el producto no está completamente definido, sino que se tiene una visión de este.

En los ambientes tradicionales, el alcance es lo que está fijo y el producto se encuentra definido. A partir de esto, se deriva el resto de las variables. Por esta razón en los procesos definidos se estima primero el tamaño, ya que es la base para estimar el resto.

2. Respecto a la precisión

Las estimaciones en los ambientes tradicionales buscan ser más precisas, debido a que son las bases para luego planificar. Dado que la precisión es realmente cara, la precisión en los ambientes ágiles y en correspondencia con la filosofía Lean, es considerada un desperdicio.

3. Principal diferencia

Las estimaciones ágiles son relativas, a diferencia de las estimaciones en ambientes tradicionales que son absolutas. En Agile, las características del producto son estimadas utilizando una medida de tamaño relativa dado que las personas no somos buenas realizando estimaciones absolutas, sino que es más fácil y rápido comparar medidas. Por otra parte, la práctica de estimar mediante comparaciones permite obtener una mejor dinámica grupal y pensamiento de equipo por sobre el individuo.

4. Quien estima

En las metodologías de estimación tradicional, las estimaciones las realiza el **Líder del proyecto** o en ocasiones las realiza un **experto**, por lo que su estimación no es representativa de la cantidad de trabajo que requiere una persona del equipo para terminar una funcionalidad del producto. Esta es una de las razones principales de los desvíos en los proyectos de software tradicional. En los ambientes ágiles y en concordancia con los valores y principios explicitados en el manifiesto ágil, es el **equipo** el que realiza la estimación. Lo óptimo es que cada uno estime su propio trabajo y de esta manera, es posible disminuir la probabilidad de desviaciones.

5. Confusión con la planificación

En los ambientes tradicionales existe un gran error que es asumir las estimaciones como compromisos, como si fueran planificaciones. Esto implica que las estimaciones se transforman en promesas.

Mientras que en agile, asumen que las estimaciones están inmersas en el campo de las probabilidades y por lo tanto no implican un compromiso. Una falla en la estimación no implica extender el tiempo, ya que en ágiles trabajamos con el tiempo fijo, por lo tanto, lo que se ajusta es el alcance.

6. Momentos para estimar

En el enfoque tradicional se estima al comienzo del proyecto, luego de la especificación de requerimientos y luego del diseño (debido que ahí es cuando puedo saber cuál es el tamaño y de ahí derivo todas las demás estimaciones). Realmente se estima cada vez que se modifica el alcance, ya que es la variable que se mantiene fija y de la cual se deriva el resto.

Las estimaciones a lo largo del proyecto se van haciendo cada vez más certeras. Al comienzo, pueden diferir hasta un 400%, esto se debe a:

1. Al comienzo tenemos poca información y alta incertidumbre, por lo que aumenta el riesgo de estimar erróneamente.
2. Las estimaciones son demasiado optimistas.
3. Decimos que aquel que no saber estimar implica estimar muchas veces.

En ágiles se estima al comienzo de cada sprint.

Frameworks de SCRUM a nivel de equipo y escala

SCRUM

Scrum es un marco ligero que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptables para problemas complejos.

En pocas palabras, Scrum requiere un Scrum Master para fomentar un entorno donde:

1. Un propietario del producto (Product Owner) ordena el trabajo de un problema complejo en un Product Backlog.

2. El equipo de Scrum convierte una selección del trabajo en un Incremento de valor durante un Sprint.
3. El equipo de Scrum y sus partes interesadas (stakeholders) inspeccionan los resultados y realizan los ajustes necesarios para el próximo Sprint.
4. *Repetir*

Scrum es simple. Pruébalo tal cual y determine si su filosofía, teoría y estructura ayudan a alcanzar metas y crear valor. El marco de Scrum es deliberadamente incompleto, solo define las partes necesarias para implementar la teoría de Scrum. Scrum se basa en la inteligencia colectiva de las personas que lo utilizan. En lugar de proporcionar a las personas instrucciones detalladas, las reglas de Scrum guían sus relaciones e interacciones.

En el marco se pueden emplear diversos procesos, técnicas y métodos. Scrum envuelve las prácticas existentes o las hace innecesarias. Scrum hace visible la eficacia relativa de la gestión actual, el entorno y las técnicas de trabajo, de modo que se pueden realizar mejoras.

Scrum se basa en el empirismo y el pensamiento Lean. El empirismo afirma que el conocimiento proviene de la experiencia y la toma de decisiones basadas en lo que se observa. El pensamiento Lean reduce los desperdicios y se centra en lo esencial.

Scrum emplea un enfoque iterativo e incremental para optimizar la previsibilidad y controlar el riesgo.

Scrum involucra a grupos de personas que colectivamente tienen todas las habilidades y experiencia para hacer el trabajo y compartir o adquirir tales habilidades según sea necesario.

Scrum combina cuatro eventos formales para la inspección y adaptación dentro de un evento contenedor, el Sprint. Estos eventos funcionan porque implementan los pilares empíricos de Scrum:
transparencia, inspección y adaptación.

Pilares de Scrum

Transparencia: El proceso y el trabajo emergentes deben ser visibles para aquellos que realizan el trabajo, así como para los que reciben el trabajo. Con Scrum, las decisiones importantes se basan en el estado percibido de sus tres artefactos formales. Los artefactos que tienen poca transparencia pueden conducir a decisiones que disminuyen el valor y aumentan el riesgo.

La transparencia permite la inspección. La inspección sin transparencia genera engaños y desperdicios.

Inspección: Los artefactos de Scrum y el progreso hacia objetivos acordados deben ser inspeccionados con frecuencia y diligentemente para detectar varianzas o problemas potencialmente indeseables. Para ayudar con la inspección, Scrum proporciona cadencia en forma de sus cinco eventos.

La inspección permite la adaptación. La inspección sin adaptación se considera inútil. Los eventos de Scrum están diseñados para provocar cambios.

Adaptación: Si algún aspecto de un proceso se desvía fuera de los límites aceptables o si el producto resultante es inaceptable, el proceso que se está aplicando o los materiales que se producen deben ajustarse. El ajuste debe realizarse lo antes posible para minimizar la desviación adicional.

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.

Valores de Scrum

El uso exitoso de Scrum depende de que las personas sean más competentes en vivir cinco valores:

Compromiso, Enfoque, Apertura, Respeto y Coraje

El equipo de Scrum se compromete a lograr sus objetivos y apoyarse mutuamente. Su enfoque principal es el trabajo

del Sprint para hacer el mejor progreso posible hacia estos objetivos. El equipo de Scrum y sus partes interesadas están abiertos sobre el trabajo y los desafíos. Los miembros del equipo de Scrum se respetan mutuamente para ser personas capaces e independientes, y son respetados como tales por las personas con las que trabajan. Los miembros del equipo de Scrum tienen el valor de hacer lo correcto y de trabajar en problemas complejos. Estos valores dan dirección al equipo de Scrum con respecto a su trabajo, acciones y comportamiento.

Las decisiones que se toman, las medidas tomadas y la forma en que se utiliza Scrum deben reforzar estos valores, no disminuirlos o socavarlos. Los miembros del equipo de Scrum aprenden y exploran los valores mientras trabajan con los eventos y artefactos de Scrum. Cuando estos valores son asimilados por el equipo de Scrum y las personas con las que trabajan, los pilares empíricos de Scrum de transparencia, inspección y adaptación cobran vida construyendo confianza.

El equipo Scrum (Scrum Team)

La unidad fundamental de Scrum es un pequeño equipo de personas, un equipo Scrum. El equipo Scrum consta de un Scrum Master, un propietario de producto (Product Owner) y desarrolladores. Dentro de un equipo de Scrum, no hay sub-equipos ni jerarquías. Es una unidad cohesionada de profesionales enfocada en un objetivo a la vez, el objetivo del Producto.

Los equipos de Scrum son multifuncionales, lo que significa que los miembros tienen todas las habilidades necesarias para crear valor en cada Sprint. También son autogestionados, lo que significa que internamente deciden quién hace qué, cuándo y cómo.

El equipo de Scrum es lo suficientemente pequeño como para permanecer ágil y lo suficientemente grande como para completar un trabajo significativo dentro de un Sprint, por lo general 10 o menos personas. En general, hemos descubierto que los equipos más pequeños se comunican mejor y son más productivos. Si los equipos de Scrum se vuelven demasiado grandes, se debe considerar la posibilidad de reorganizarse en varios equipos Scrum cohesionados, cada uno centrado en el mismo producto. Por lo tanto, deben compartir el mismo objetivo de producto, trabajo pendiente del producto (Product Backlog) y propietario del producto (Product Owner).

El equipo Scrum es responsable de todas las actividades relacionadas con los productos, desde la colaboración, verificación, mantenimiento, operación, experimentación, investigación y desarrollo, y cualquier otra cosa que pueda ser necesaria. Están estructurados y empoderados por la organización para gestionar su propio trabajo.

Trabajar en Sprints a un ritmo sostenible mejora el enfoque y la consistencia del equipo de Scrum.

Todo el equipo de Scrum es responsable de crear un incremento valioso y útil en cada Sprint. Scrum define tres responsabilidades específicas dentro del equipo de Scrum: los desarrolladores, el propietario del producto (Product Owner) y el Scrum Master.

Desarrolladores

Los desarrolladores son las personas del equipo Scrum que se comprometen a crear cualquier aspecto de un Incremento útil (funcional) en cada Sprint.

Las habilidades específicas que necesitan los desarrolladores son a menudo amplias y variarán con el dominio del trabajo. Sin embargo, los desarrolladores siempre son 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 Sprint;
- Responsabilizarse mutuamente como profesionales.

Propietario del producto (Product Owner)

El Propietario del Producto es responsable de maximizar el valor del producto resultante del trabajo del equipo de Scrum. La forma en que esto se hace esto puede variar ampliamente entre organizaciones, equipos Scrum e individuos.

El Propietario del Producto también es responsable de la gestión eficaz de la pila del producto (Product Backlog), que incluye:

- Desarrollar y comunicar explícitamente el Objetivo del Producto;
- Creación y comunicación clara de elementos de trabajo pendiente del producto;
- Pedido de artículos de trabajo pendiente del producto;
- Asegurarse de que el trabajo pendiente del producto sea transparente, visible y comprendido.

El Propietario del Producto puede hacer el trabajo anterior o puede delegar la responsabilidad a otros. En cualquier caso, el propietario del producto sigue siendo responsable.

Para que los Propietarios de Productos tengan éxito, toda la organización debe respetar sus decisiones. Estas decisiones son visibles en el contenido y el orden del trabajo pendiente del producto, y a través del Incremento inspeccionable en la revisión de Sprint.

El Propietario del Producto es una persona, no un comité. El Propietario del Producto puede representar las necesidades de muchas partes interesadas en el trabajo pendiente del producto. Aquellos que deseen cambiar el trabajo pendiente del producto pueden hacerlo tratando de negociar con criterio con el Product Owner.

Scrum Master

El Scrum Master es responsable de establecer Scrum tal como se define en la Guía de Scrum. Lo consigue ayudando a todos a comprender la teoría y la práctica de Scrum, tanto dentro del Equipo como en toda la organización.

El Scrum Master es responsable de la efectividad del Scrum Team. Lo logra al permitir que el equipo Scrum mejore sus prácticas, dentro del marco de Scrum.

Los Scrum Masters son verdaderos líderes que sirven al equipo Scrum y a toda la organización.

El Scrum Master sirve al equipo de Scrum de varias maneras, incluyendo:

- Capacitar a los miembros del equipo en autogestión y multifuncionalidad;
- Ayudar al equipo de Scrum a centrarse en la creación de incrementos de alto valor que cumplan con la definición de hecho;
- Promover la eliminación de los impedimentos para el progreso del equipo Scrum;
- Asegurar de que todos los eventos de Scrum se lleven a cabo, sean positivos, productivos y que se respete el tiempo establecido (time-box) para cada uno de ellos.

El Scrum Master sirve al Propietario del Producto (Product Owner) de varias maneras, incluyendo:

- Ayudar a encontrar técnicas para una definición eficaz de los objetivos del producto y la gestión de los retrasos en el producto;
- Ayudar al equipo de Scrum a comprender la necesidad de elementos de trabajo pendiente de productos claros y concisos;
- Ayudar a establecer la planificación empírica de productos para un entorno complejo;
- Facilitar la colaboración de las partes interesadas según sea solicitado o necesario.

El Scrum Master sirve a la organización de varias maneras, incluyendo:

- Liderar, capacitar y mentorizar a la organización en su adopción de Scrum;
- Planificar y asesorar sobre la implementación de Scrum dentro de la organización;
- Ayudar a las personas y a las partes interesadas a comprender y promulgar un enfoque empírico para el trabajo complejo;
- Eliminar las barreras entre las partes interesadas y los equipos de Scrum.

Eventos de Scrum

El Sprint es un contenedor para todos los eventos. Cada evento en Scrum es una oportunidad formal para inspeccionar y adaptar los artefactos de Scrum. Estos eventos están diseñados específicamente para permitir la transparencia necesaria. Si no se realizan los eventos según lo prescrito, se pierden oportunidades para inspeccionar y adaptarse. Los eventos se utilizan en Scrum para crear regularidad y minimizar la necesidad de reuniones no definidas en Scrum. De manera óptima, todos los eventos se llevan a cabo al mismo tiempo y lugar para reducir la complejidad.

El Sprint

Los sprints son el latido del corazón de Scrum, donde las ideas se convierten en valor.

Son eventos de longitud fija de un mes o menos para crear consistencia. Un nuevo Sprint comienza inmediatamente después de la conclusión del Sprint anterior.

Todo el trabajo necesario para alcanzar el objetivo del producto, incluyendo la Planificación (Sprint Planning), Daily Scrums, Revisión del Sprint (Sprint Review) y la Retrospectiva (Sprint Retrospective), ocurren dentro del Sprints.

Durante el Sprint:

- No se hacen cambios que pongan en peligro el Objetivo Sprint;
- La calidad no disminuye;
- El trabajo pendiente del producto se refina según sea necesario;
- El alcance se puede clarificar y renegociar con el Propietario del Producto a medida que se aprende más.

Los Sprints permiten la previsibilidad al garantizar la inspección y adaptación del progreso hacia un objetivo del Producto, como mínimo, una vez al mes en el calendario. Cuando el horizonte de un Sprint es demasiado largo, el Objetivo de Sprint puede volverse obsoleto, la complejidad puede aumentar y el riesgo puede aumentar. Los Sprints más cortos se pueden emplear para generar más ciclos de aprendizaje y limitar el riesgo de coste y esfuerzo a un período de tiempo más pequeño. Cada Sprint puede considerarse un proyecto corto.

Existen varias prácticas para pronosticar el progreso, como gráficos de burn-downs, burn-ups, o flujos acumulativos. Si bien han demostrado ser útiles, estos no sustituyen la importancia del empirismo. En entornos complejos, se desconoce lo que sucederá. Solo lo que ya ha sucedido se puede utilizar para la toma de decisiones con vistas a futuro.

Un Sprint podría ser cancelado si el Objetivo del Sprint se vuelve obsoleto. Solo el Propietario del Producto tiene la autoridad para cancelar el Sprint.

1. Planificación de Sprint (Sprint Planning)

El Sprint Planning inicia el Sprint estableciendo el trabajo que se realizará para el mismo. Este plan resultante es creado por el trabajo colaborativo de todo el equipo de Scrum.

El propietario del producto (Product Owner) se asegura de que los asistentes estén preparados para discutir los elementos de trabajo pendiente de producto más importantes y cómo se asignan al objetivo del producto. El equipo de Scrum también puede invitar a otras personas a asistir a la planificación del Sprint

para proporcionar asesoramiento.

La planificación del Sprint aborda los siguientes temas:

Tema Uno: ¿Por qué este Sprint es valioso?

El Propietario del Producto (Product Owner) propone cómo el producto podría aumentar su valor y utilidad en el Sprint actual. A continuación, todo el equipo de Scrum colabora para definir un objetivo de Sprint que comunique por qué el Sprint es valioso para las partes interesadas. El Objetivo Sprint debe finalizarse antes del final de la Planificación de Sprint.

Tema dos: ¿Qué se puede hacer este Sprint?

A través del debate con el propietario del producto (Product Owner), los desarrolladores seleccionan los elementos del Product Backlog para incluir en el Sprint actual. El equipo de Scrum puede refinar estos elementos durante este proceso, lo que aumenta la comprensión y confianza.

Seleccionar cuánto se puede completar dentro de un Sprint puede ser un desafío. Sin embargo, cuanto más sepan los desarrolladores sobre su rendimiento pasado, su capacidad futura y su definición de hecho, más seguro estarán en sus pronósticos de Sprint.

Tema Tres: ¿Cómo se realizará el trabajo elegido?

Para cada elemento de trabajo pendiente de producto (Product Backlog item) seleccionado, los desarrolladores planifican el trabajo necesario para crear un incremento que cumpla con la definición de hecho. Esto se hace normalmente mediante la descomposición de elementos de trabajo pendiente (Product Backlog item) del producto en elementos de trabajo más pequeños que se puedan realizar en un día o menos. La forma de hacerlo es según la discreción de los propios desarrolladores. Nadie más les dice cómo convertir los elementos de trabajo pendiente del producto en incrementos de valor.

El objetivo de Sprint (Sprint Goal), los elementos de trabajo pendiente de producto seleccionados para el Sprint, más el plan para entregarlos se conocen conjuntamente como el trabajo pendiente de Sprint (Sprint Backlog).

El Sprint Planning tiene una duración máxima de ocho horas para un Sprint de un mes. Para sprints más cortos, el evento suele ser más corto.

2. Scrum Diario (Daily Scrum)

El propósito del Daily Scrum es inspeccionar el progreso hacia el Objetivo Sprint y adaptar el Sprint Backlog según sea necesario, ajustando el próximo trabajo planeado.

El Daily Scrum es un evento de 15 minutos (máximo) para los desarrolladores del equipo de Scrum. Para reducir la complejidad, se lleva a cabo al mismo tiempo y lugar todos los días laborables del Sprint. Si el propietario del producto o el Scrum Master están trabajando activamente en los elementos del Trabajo pendiente de Sprint, participan como desarrolladores.

Los desarrolladores pueden seleccionar cualquier estructura y técnicas que deseen, siempre y cuando su Scrum diario se centre en el progreso hacia el objetivo de Sprint y produzca un plan accionable para el día siguiente de trabajo. Esto crea enfoque y mejora la autogestión.

Los Scrums diarios (Daily Scrum) mejoran la comunicación, identifican impedimentos, promueven una rápida para la toma de decisiones, y en consecuencia, eliminan la necesidad de otras reuniones.

El Daily Scrum no es la única vez que los desarrolladores pueden ajustar su plan. Frecuentemente se reúnen durante todo el día para debatir de forma más detallada sobre la adaptación o replanificación del resto del trabajo del Sprint

3. Revisión del Sprint (Sprint Review)

El propósito de la revisión del Sprint es inspeccionar el resultado del Sprint y determinar futuras adaptaciones. El equipo de Scrum presenta los resultados de su trabajo a las partes interesadas clave y se discute el progreso hacia el Objetivo de Producto.

Durante el evento, el equipo de Scrum y las partes interesadas revisan lo que se logró en el Sprint y lo que ha cambiado en su entorno. En base a esta información, los asistentes colaboran en qué hacer a continuación. El trabajo pendiente del producto también se puede ajustar para satisfacer nuevas oportunidades. Sprint Review es una sesión de trabajo y el equipo de Scrum debe evitar limitarla a que se convierta en una simple presentación.

La revisión de Sprint es el penúltimo evento del Sprint y se utiliza en un plazo máximo de cuatro horas para un Sprint de un mes. Para sprints más cortos, el evento suele ser más corto.

4. La retrospectiva del Sprint (Sprint Retrospective)

El propósito de la retrospectiva Sprint es planificar formas de aumentar la calidad y la eficacia.

El equipo de Scrum inspecciona cómo fue el último Sprint con respecto a individuos, interacciones, procesos, herramientas y su definición de Hecho. Los elementos inspeccionados a menudo varían según el dominio del trabajo. Las suposiciones que los desviaron se identifican y se exploran sus orígenes. El equipo de Scrum analiza qué fue bien durante el Sprint, qué problemas encontró y cómo esos problemas fueron (o no fueron) resueltos.

El equipo de Scrum identifica los cambios más útiles para mejorar su eficacia. Las mejoras más impactantes se abordan lo antes posible. Incluso se pueden agregar al Sprint Backlog para el próximo Sprint.

La retrospectiva Sprint concluye el Sprint. Se utiliza un intervalo de tiempo de hasta un máximo de tres horas para un Sprint de un mes. Para sprints más cortos, el evento suele ser más corto.

5. Refinamiento del Product Backlog

Artefactos de Scrum

Los artefactos de Scrum representan trabajo o valor. Están diseñados para maximizar la transparencia de la información clave. Por lo tanto, cada uno de los que los inspecciona tienen la misma base para la adaptación. Cada artefacto contiene un compromiso para garantizar que proporciona información que mejora la transparencia y el enfoque con el que se puede medir el progreso:

- Para el trabajo pendiente del producto es el objetivo del producto.
- Para el Sprint Backlog es el Sprint Goal.
- Para el Incremento es la Definición de Hecho.

Estos compromisos existen para reforzar el empirismo y los valores de Scrum para el equipo de Scrum y sus partes interesadas.

1. Pila del producto (Product Backlog)

El trabajo pendiente del producto es una lista emergente y ordenada de lo que se necesita para mejorar el producto. Es la única fuente de trabajo emprendida por el equipo Scrum.

Los elementos de trabajo pendiente de producto que puede ser hecho por el equipo de Scrum dentro de un Sprint se consideran listos para su selección en un evento de planificación de Sprint. Por lo general adquieren este grado de transparencia después de las actividades de refinación. El refinamiento de Backlog del producto es el acto de descomponer y definir aún más los elementos de trabajo pendiente del producto en artículos más pequeños y precisos. Esta es una actividad en curso para agregar detalles, como una descripción, un pedido y un tamaño. Los atributos a menudo varían con el dominio del trabajo.

Los desarrolladores que realizarán el trabajo son responsables del tamaño. El Propietario del Producto (Product Owner) puede influir en los desarrolladores ayudándoles a entender y seleccionar mejores alternativas.

Compromiso: Objetivo del producto (Product Goal)

El objetivo del producto (Product Goal) describe un estado futuro del producto que puede servir como objetivo para el equipo Scrum contra el cual planificar. El objetivo del producto se encuentra en el trabajo pendiente del producto (Product Backlog). El resto del trabajo pendiente del producto surge para definir "qué" cumplirá el objetivo del producto.

Un producto es un vehículo para entregar valor. Tiene un límite claro, partes interesadas conocidas, usuarios o clientes bien definidos. Un producto podría ser un servicio, un producto físico o algo más abstracto.

El objetivo del producto es el objetivo a largo plazo para el equipo Scrum. Deben cumplir (o abandonar) un objetivo antes de asumir el siguiente.

2. La pila del Sprint (Sprint Backlog)

El Trabajo pendiente de Sprint se compone del objetivo sprint (por qué), el conjunto de elementos de trabajo pendiente de producto seleccionados para el Sprint (qué), así como un plan accionable para entregar el incremento (cómo).

El Trabajo pendiente de Sprint es un plan por y para los desarrolladores. Es una imagen muy visible y en tiempo real del trabajo que los desarrolladores planean realizar durante el Sprint para lograr el Objetivo Sprint. Por lo tanto, el Sprint Backlog se actualiza a lo largo del Sprint a medida que se aprende más.

Debe tener suficientes detalles para que puedan inspeccionar su progreso en el Scrum Diario.

Compromiso: Sprint Goal

El Sprint Goal es el único objetivo para el Sprint. Aunque el objetivo de Sprint es un compromiso de los desarrolladores, proporciona flexibilidad en términos del trabajo exacto necesario para lograrlo. El Objetivo Sprint también crea coherencia y enfoque, animando al equipo de Scrum a trabajar juntos en lugar de en iniciativas separadas.

El objetivo de Sprint se crea durante el evento Sprint Planning y, a continuación, se agrega al Trabajo pendiente de Sprint. A medida que los desarrolladores trabajan durante el Sprint, tienen en cuenta el objetivo de Sprint. Si el trabajo resulta ser diferente de lo que esperaban, colaboran con el propietario del producto para negociar el alcance del Trabajo pendiente de Sprint dentro del Sprint sin afectar al objetivo de Sprint.

3. Incremento (Increment)

Un Incremento es un paso de hormigón hacia el Objetivo del Producto. Cada Incremento es aditivo a todos los Incrementos anteriores y verificado a fondo, asegurando que todos los Incrementos funcionen juntos. Para proporcionar el valor, el incremento debe ser utilizable.

Se pueden crear varios incrementos dentro de un Sprint. La suma de los Incrementos se presenta en la Revisión Sprint apoyando así el empirismo. Sin embargo, un Incremento puede ser entregado a las partes interesadas antes del final del Sprint. La revisión de Sprint nunca debe considerarse una puerta para liberar valor.

El trabajo no se puede considerar parte de un Incremento a menos que cumpla con la Definición de Hecho.

Compromiso: Definición de Hecho (Definition of Done)

La Definición de Hecho es una descripción formal del estado del Incremento cuando cumple con las medidas de calidad requeridas para el producto.

En el momento en que un elemento de trabajo pendiente de producto cumple con la definición de hecho, se crea un incremento.

La definición de Hecho crea transparencia al proporcionar a todos una comprensión compartida de qué trabajo se completó como parte del Incremento. Si un elemento de trabajo pendiente de producto no cumple con la definición de hecho, no se puede liberar, ni siquiera presentar en la revisión de Sprint. En su lugar, vuelve al Trabajo pendiente del producto para su consideración futura.

Si la definición de hecho para un incremento forma parte de los estándares de la organización, todos los equipos de Scrum deben seguirla como mínimo. Si no es un estándar organizativo, el equipo de Scrum debe crear una definición de hecho adecuada para el producto.

Los desarrolladores deben ajustarse a la definición de Hecho. Si hay varios equipos de Scrum trabajando juntos en un producto, deben definir y cumplir mutuamente con la misma definición de hecho.

Timebox

Esto existe para que no se pierda más tiempo del necesario en las distintas tareas y para aprender a negociar en términos del alcance del producto y no del tiempo o los costos (relacionado con la triple restricción).

Si no se llega al final de un sprint, en vez de extender el tiempo, se entrega menos.

De esta manera, nos volvemos más confiables, con un ritmo de trabajo sostenible. Todas las ceremonias son Timebox. El refinamiento del Product Backlog al ser una actividad continua, no tiene definido un tiempo exacto, sino que el tiempo se expresa en términos porcentuales sobre la definición del Sprint.



Definición de Listo (DoR)

El DoR es un criterio que define cuando una US está lo suficientemente bien formulada como para poderse incluir en un Sprint. Por ejemplo, que cierta US debe tener un prototipo, entonces cuando el prototipo esté realizado, el DoR dirá “se ha definido el prototipo para la US”.

El DoD es un criterio que dice cuando una historia está lo suficientemente bien implementada como para entrar al Sprint Review y ser mostrada al PO, entonces él será el encargado de aprobar o no la US y en caso de que esté aprobada decidirá si desea ponerla en producción.

En el DoD se arma un checklist con todo lo que debe cumplir una US para entrar en producción.

Definición de Hecho (DONE)
<input type="checkbox"/> Diseño revisado
<input type="checkbox"/> Código Completo
<input type="checkbox"/> Código refactorizado
<input type="checkbox"/> Código con formato estándar
<input type="checkbox"/> Código Comentado
<input type="checkbox"/> Código en el repositorio
<input type="checkbox"/> Código Inspeccionado
<input type="checkbox"/> Documentación de Usuario actualizada
<input type="checkbox"/> Probado
<input type="checkbox"/> Prueba de unidad hecha
<input type="checkbox"/> Prueba de integración hecha
<input type="checkbox"/> Prueba de sistema hecha
<input type="checkbox"/> Cero defectos conocidos
<input type="checkbox"/> Prueba de Aceptación realizada
<input type="checkbox"/> En los servidores de producción

Capacidad del equipo en un Sprint

La capacidad es una métrica que utiliza SCRUM para determinar cuánta cantidad de trabajo puede comprometer un equipo para un determinado Sprint.

La capacidad se estima y se mide.

Una de las cosas que se hace en el Sprint Planning es determinar la capacidad del equipo, y entonces, teniendo en cuenta la capacidad, se contrasta en cuántas US del Product Backlog se pueden incorporar en el Sprint Backlog.

Para equipos más maduros, la capacidad puede ser estimada en Story Points y para equipos menos maduros se puede estimar en horas ideales.



Niveles de Planificación

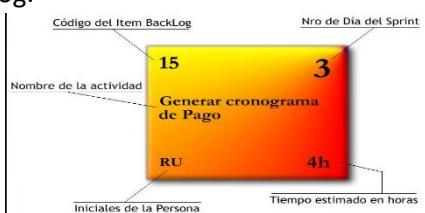
Nivel	Horizonte	Quién	Foco	Entregable
Portfolio	1 año o más	Stakeholders y Product Owners	Administración de un Portfolio de Producto	Backlog de Portfolio
Producto	Arriba de varios meses o más	Product Owner y Stakeholder	Visión y evolución del producto a través del tiempo	Visión de Producto, roadmap y características
Release	3 (o menos) a 9 meses	Equipo Scrum entero y Stakeholders	Balancear continuamente el valor del cliente y la calidad global con las restricciones de alcance, cronograma y presupuesto	Plan de reléase
Iteración	Cada iteración (de 1 semana a 1 mes)	Equipo Scrum entero	Que aspectos entregar en el siguiente Sprint	Objetivo del Sprint y Sprint Backlog
Día	Diaria	Scrum Master y Equipo de desarrollo	Cómo completar lo comprometido	Inspección del progreso

Herramientas de SCRUM

Tarjeta de tarea

Es una tarjeta que cuenta con 5 partes:

1. Código del ítem Backlog: es un identificador del ítem en el Product Backlog.
2. Nombre de la actividad: suele ser una frase verbal que define qué es lo que hay que hacer.
3. Iniciales del responsable de realizar la tarea.
4. Número de día del sprint.
5. Estimación del tiempo para finalizar la tarea.



Tablero

Tablero utilizado por el Equipo de Desarrollo en el cual se colocan las tarjetas de tareas a realizar durante el sprint. Se encuentra dividido en 5 secciones:

1. Story: el nombre de la historia.
2. To Do: aquí se encuentran las tareas por hacer en el Sprint.
3. Work In Process: aquí se encuentran las tareas que se están realizando.
4. To Verify: aquí se encuentran las tareas finalizadas que deben verificarse.
5. Done: aquí se encuentran las tareas terminadas: finalizadas y verificadas.

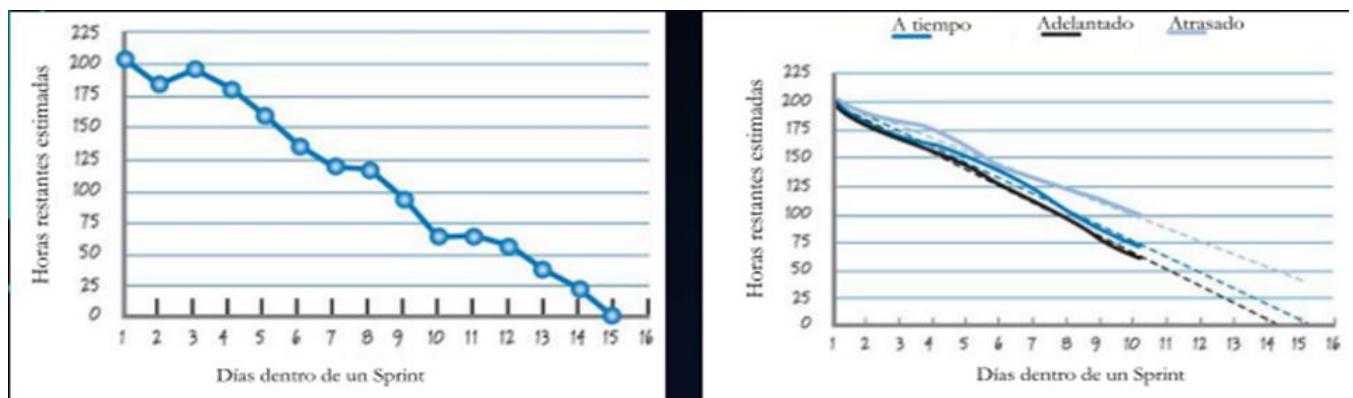
Lo ideal es que el nivel de granularidad de las tareas sea lo más fino posible, de tal forma que se detalle profundamente el avance del sprint, teniendo varias tareas en cada sección del tablero.

Gráficos del Backlog

Gráficos que brindan información acerca del progreso de un Sprint, de una release o del producto. El backlog de trabajo es la cantidad de trabajo que queda por ser realizado (en horas), mientras que la tendencia del backlog compara esta cantidad con el tiempo (medido en días).

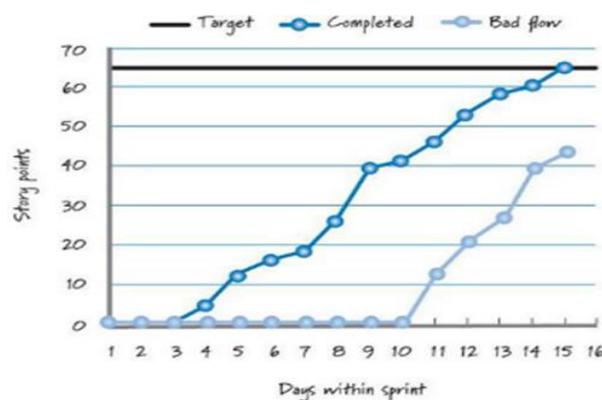
Sprint Burn-Down Chart

Visualiza con una curva descendente cuánto trabajo queda para terminar. En el eje de las abscisas se coloca el tiempo que va desde el inicio del sprint. En el eje de las ordenadas se coloca los puntos de historia a realizar en el sprint.



Sprint Burn-Up Chart

Visualiza con una curva ascendente cuánto trabajo se ha completado a lo largo de la release, es decir que visualiza los puntos de historia terminados en cada sprint.



Combined Burn Chart

Visualiza cuánto trabajo queda para terminar y cuánto trabajo se ha realizado.

Frameworks para escalar SCRUM

Dado que Scrum impone un límite en la cantidad máxima de integrantes del Equipo de Desarrollo, surge la necesidad de escalar este Framework para poder gestionar productos de mayor envergadura, ya que las prácticas agiles se utilizan en ambientes complejos para reducir tal complejidad.

Existen diferentes frameworks para escalar como Nexus, scale scrum, less, SAFe.

Framework Nexus

Nexus es un Framework que consiste en roles, eventos, artefactos y técnicas que vinculan el trabajo de aproximadamente tres a nueve equipos Scrum que trabajan sobre un único Product Backlog común para la construcción de un incremento integrado de producto “Terminado”. Con un solo Product Owner para entregar un único producto.

Nexus surge para lidiar con la complejidad que supone varios Equipos Scrum trabajando sobre un mismo Product Backlog. Esta complejidad está dada por las siguientes dependencias entre equipos:

1. Requerimientos: el alcance de estos puede superponerse. La forma en que se implementan también puede afectar a los demás.
2. Conocimiento del dominio: el conocimiento del sistema de negocio debería mapearse a los Equipos Scrum para minimizar las interrupciones entre los mismos durante el Sprint.

Responsabilidades

- **Nexus Integration Team**: Responsable de asegurar que se produzca un incremento integrado al menos una vez en cada sprint. La integración incluye abordar las restricciones técnicas y no técnicas del equipo multifuncional que pueden impedir la capacidad de un Nexus para entregar constantemente un incremento integrado.
- **Product Owner**: Es responsable de maximizar el valor del producto y el trabajo realizado e integrado por los Scrum Teams en un Nexus, así mismo también es responsable de la gestión eficaz del Product Backlog.
- **Scrum Master**: responsable de asegurar que el marco de trabajo Nexus se entienda y se promulgue como se describe en la Guía de Nexus. Puede ser un Scrum Master en uno o más de los scrums teams en el Nexus.
- **Uno o más miembros del Nexus Integration Team**: a menudo está formado por miembros de los Scrum Teams que ayudan a los Scrum Teams a adoptar herramientas y prácticas que contribuyen a mejorar la capacidad de los Scrum Teams para entregar un Integrated Increment útil y de valor que cumple con la Definición de Terminado.

Eventos

Nexus agrega o extiende los eventos definidos por Scrum. La duración de los eventos Nexus se guía por la duración de los eventos correspondientes en la Guía de Scrum. Tienen definido un bloque de tiempo adicional a sus correspondientes eventos de Scrum.

- **Nexus Sprint:** Los scrums teams producen un solo Integrated Increment (Incremento de integración), es lo mismo que scrum.
- **Refinamiento entre equipos:** El Refinamiento Entre Equipos del trabajo del Product Backlog reduce o elimina las dependencias entre equipos dentro de un Nexus. El Product Backlog debe descomponerse para que las dependencias sean transparentes, se identifiquen entre equipos y se eliminan o se minimicen.
El Refinamiento Entre Equipos del Product Backlog a escala sirve a un propósito dual:
 - Ayuda a los Scrum Teams a prever qué equipo entregará qué elementos del Product Backlog.
 - Identifica dependencias entre estos equipos.

El Refinamiento Entre Equipos es continuo. La frecuencia, duración y participación del Refinamiento Entre Equipos varía para optimizar estos dos propósitos.

- **Nexus Sprint Planning:** El propósito de la Nexus Sprint Planning es coordinar las actividades de todos los Scrum Teams dentro de un Nexus para un solo Sprint. Los representantes apropiados de cada Scrum Team y el Product Owner se reúnen para planificar el Sprint.

Resultados:

- Objetivo de Sprint para cada Scrum team.
- Objetivo de Nexus.
- Un Nexus sprint backlog.
- Un Sprint backlog.

- **Nexus Daily Scrum:**
 - Se discuten problemas de integración y se inspecciona el progreso hacia el objetivo de sprint del nexus.
 - Solo asisten representantes de cada Scrum team.
 - La Daily Scrum de cada Scrum Team complementa la Nexus Daily Scrum creando planes para el día, centrados principalmente en abordar los problemas de integración planteados durante la Nexus Daily Scrum.
 - Los Scrum teams no solo se reúnen o coordinan en el Nexus daily scrum, sino que durante el día, puede existir comunicación entre equipos con mayor detalle sobre adaptar o replanificar el resto del trabajo del sprint.
- **Nexus Sprint Review:** La Nexus Sprint Review se lleva a cabo al final del Sprint para brindar retroalimentación al Integrated Increment terminado que el Nexus ha construido durante el Sprint y determinar adaptaciones futuras.

La nexus sprint review reemplaza a las sprint reviews de cada scrum team.

- **Nexus Sprint Retrospective:** El propósito de la Nexus Sprint Retrospective es planificar formas de mejorar la calidad y la eficacia en todo el Nexus. El Nexus inspecciona cómo fue el último Sprint con respecto a individuos, equipos, interacciones, procesos, herramientas y su Definición de Terminado. Además de las mejoras de cada equipo, las Retrospectivas de los Scrum Teams complementan la Nexus Sprint Retrospective mediante el uso de inteligencia de abajo hacia arriba para centrarse en los problemas que afectan al Nexus en su conjunto.

Artefactos

Representan trabajo o valor y están diseñados para maximizar la transparencia. El Nexus Integration Team trabaja con los Scrum Teams dentro de un Nexus para garantizar que se logre la transparencia en todos los artefactos y que el estado del Integrated Increment se entienda.

Cada artefacto contiene un compromiso, y estos existen para reforzar el empirismo y el valor de Scrum para el Nexus y sus interesados.

- **Product Backlog:** Hay uno solo que contiene la lista de todo lo que el Nexus y sus Scrum Teams necesitan para mejorar el producto. El Product Backlog debe entenderse con tal nivel que permita detectar y minimizar las dependencias. Es responsabilidad del Product Owner.
El compromiso es el Objetivo del Producto, que describe el estado futuro del producto y sirve como un objetivo a largo plazo para el Nexus.
- **Nexus Sprint Backlog:** está compuesto del Objetivo del Sprint de Nexus y elementos del Product Backlog de cada Scrum Team del Nexus. Se usa para resaltar las dependencias y el flujo de trabajo durante el Sprint, y se actualiza durante el mismo a medida que se obtiene más conocimiento. Este debe tener un nivel de detalle tal que permita que Nexus pueda inspeccionar su progreso en la Nexus Daily Scrum.
Su compromiso es el Objetivo de Sprint de Nexus, que es un único objetivo para Nexus. Es la suma de todo el trabajo y los Objetivos de Sprint de los Scrum Teams dentro del Nexus. Se crea en la Nexus Sprint Planning y se agrega al Sprint Backlog de Nexus. Los Scrum Teams lo tienen en cuenta a medida que trabajan. En la Nexus Sprint Review se debe de mostrar la funcionalidad de valor y útil que está terminada para alcanzar el Objetivo del Sprint.
- **Integrated Increment:** es la suma actual de todo el trabajo integrado completado por un Nexus en relación con el Objetivo del Producto. Se inspecciona en la Nexus Sprint Review pero puede entregarse antes de la misma. Debe cumplir con la definición de terminado.
El compromiso de este artefacto es la Definición de Terminado, que define el estado del trabajo integrado cuando cumple con la calidad y las medidas requeridas para el producto. El incremento se termina sólo cuando está integrado, es de valor y utilizable. Es responsabilidad del Nexus Integration team una definición de terminado que se pueda aplicar en cada sprint, y todos los Scrum Team deben definir y adherirse a esta definición. Cada Scrum Team se autogestiona para llegar a este estado, pudiendo aplicar una definición de terminado más estricta pero nunca usar criterios menos rigurosos de lo acordado para el Integrated Increment.

Framework LeSS

LeSS es un marco de trabajo que permite escalar Scrum y se aplica a productos de entre dos a ocho equipos. LeSS propone un solo Product Owner, un Scrum Master que puede servir de uno a tres equipos, gestionando un único Product Backlog. El Sprint es común para todos los equipos. Si se trabaja con más de ocho equipos, se utiliza LeSS Huge.

Principios

- Scrum a gran escala es Scrum;
- Más con LeSS;
- Pensamiento sistémico;
- pensamiento Lean;
- Control empírico de procesos;
- Transparencia;
- Mejora continua hacia la perfección;
- Centrado en el consumidor;
- Enfoque de producto completo;
- Teoría de colas.

Miembros del Equipo

- **Scrum Master:** enseña Scrum y LeSS a la organización y los entrena en su adopción. Domina Scrum y LeSS y usa este conocimiento para guiar a todos a descubrir cómo pueden contribuir mejor a crear el producto más valioso, crea el ambiente para que las personas tengan éxito. El Scrum Master dedica todo su tiempo a cumplir ese rol y un Scrum Master sirve entre uno a tres equipos. Su enfoque es hacia los equipos, el Product Owner, la organización y las prácticas de desarrollo; se enfoca en todo el sistema organizacional no en un solo equipo.
- **Product Owner:** enseña Scrum y LeSS a la organización y los entrena en su adopción. Domina Scrum y LeSS y usa este conocimiento para guiar a todos a descubrir cómo pueden contribuir mejor a crear el producto más valioso, crea el ambiente para que las personas tengan éxito. El Scrum Master dedica todo su tiempo a cumplir ese rol y un Scrum Master sirve entre uno a tres equipos. Su enfoque es hacia los equipos, el Product Owner, la organización y las prácticas de desarrollo; se enfoca en todo el sistema organizacional no en un solo equipo.
- **Equipos:** un equipo en LeSS es el mismo que en Scrum de un solo equipo. El objetivo del equipo en LeSS es agregar ítems del Product Backlog al producto durante el Sprint. Los equipos trabajan en estrecha colaboración con los clientes/usuarios para aclarar la definición de los elementos y con el Product Owner en la priorización. Coordinan e integran su trabajo con el de los demás equipos, para final del Sprint haber producido un solo incremento del producto. Cada equipo tiene la responsabilidad de manejar las relaciones con los demás equipos.
Cada equipo es autogestionado, multifuncional.

Eventos

- **Sprint Planning One:** es atendida por el Product Owner y los equipos o representantes de estos. Juntos seleccionan tentativamente a los ítems que cada equipo trabajará en ese Sprint. Los equipos identifican oportunidades para trabajar juntos y se aclaran las preguntas finales.
- **Sprint Planning Two:** es para que decidan cómo harán los ítems seleccionados. Esto generalmente involucra el diseño y la creación de su Sprint Backlog.
- **Daily Scrum:** cada equipo tiene su propia Daily Scrum y no hay diferencia con las Daily Scrum para un solo equipo. Tiene una duración de 15 minutos. Y los miembros del equipo deben responder a tres preguntas.
 - ¿Qué hice ayer?
 - ¿En qué voy a trabajar hoy?
 - ¿Qué queda por hacer?
- **Sprint Review:** hay un review para el producto y es común para todos los equipos. Es el punto de inspección – adaptación y se realiza al final del Sprint; es la ocasión para todos los equipos de ver el incremento (el enfoque se pone en todo el producto). Durante esta ceremonia, clientes y Stakeholders examinan lo que los equipos construyeron durante el Sprint y se discute sobre cambios y se aportan nuevas ideas. Los equipos deben estar juntos y el PO, clientes7Stakeholders definen la dirección del producto. Es importante que los Stakeholders formen parte y aporten la información necesaria para una inspección y adaptación efectiva.
- **Retrospective:** Al final del Sprint, cada equipo individualmente realiza su propia Sprint Retrospective.
- **Overall Retrospective:** nueva reunión el LeSS. Su propósito es discutir problemas entre equipos organizacionales y sistémicos dentro de la organización. Algunas preguntas que se hacen son: ¿Qué tan bien

trabajan los equipos juntos?; ¿Hay algo que los equipos hagan que puedan compartir?; ¿Están los equipos aprendiendo juntos?

- **Product Backlog Refinement:** es necesario dentro de cada Sprint para refinar los elementos y estar listos para futuros Sprints. Las actividades claves son: dividir elementos grandes; aclarar elementos hasta que estén listos para la implementación y estimar tamaño, valor, riesgos. A esto lo realiza todo el equipo junto y no el PO por separado.
- **LeSS Sprint:** hay un sprint a nivel producto, no un Sprint diferente para cada equipo. Cada equipo comienza y termina el Sprint al mismo tiempo.

Artefactos

- **Incremento de producto potencialmente entregable:** la salida de cada Sprint es un incremento de producto potencialmente entregable. Es la forma de integrar el trabajo de todos los equipos al finalizar el Sprint. Potencialmente enviable es una declaración sobre la calidad del software y no sobre el valor o comerciabilidad del este. Si el producto se puede enviar realmente dependerá del Definition of Done.

Hay un DoD común para todos los equipos.

- Cada equipo puede tener su propia DoD más detallada, pero siempre expandiendo de la común.
- El objetivo es mejorar la DoD para que resulte en su envío de producto en cada Sprint (o incluso con más frecuencia).

- **Product Backlog:** Todos los equipos construyen un solo producto en base a los ítems de un solo Product Backlog. Estos ítems no están preasignados a los equipos. La definición de producto debe ser tan amplia y centrada en el usuario final/cliente como sea práctico. Con el tiempo, la definición de producto podrá expandirse. El Product Backlog de LeSS es el mismo que en un entorno Scrum de un solo equipo.
- **Sprint Backlog:** cada equipo tiene su propio Sprint Backlog. Es la lista de trabajo que el equipo debe realizar para poder completar los ítems del Product Backlog. Por lo tanto, el Sprint Backlog es por equipo y no hay diferencia entre un LeSS Sprint Backlog y un Sprint Backlog.

LeSS Huge

Para más de 8 equipos.

- ¿Qué es igual a LeSS?
 - Un Product Backlog para todos.
 - Un DoD.
 - Un DoR.
 - Un Product Manager.
 - El Sprint es común para todos los equipos.
 - Un incremento potencialmente entregable.
- ¿Qué es diferente a LeSS?
 - Ahora hay un Area Product Manager.
 - Hay un Area Product Owners.
 - Hay un Area Product Backlogs.
 - Hay un Area Product Backlogs.

- Hay un Area Product Vision.
- Hay más de 8 equipos.

Métricas ágiles y en otros enfoques

¿Qué es una métrica?

Una métrica es un número que representa el grado o presencia de un determinado conjunto de atributos respecto de lo que se quiere medir (proceso, producto o proyecto), expresadas de tal forma que permitan una medición objetiva de la realidad. Por ejemplo, la escala: NS, S, B, MB, E no sirve. Lo que sí sirve es cuántos E hay, cuántos S hay, etc. Debe ser posible medir en términos de la definición de esta y del esfuerzo que se debe aplicar para medirla. Es decir que la métrica no es la escala sino la cantidad de cierta escala.

Todas las métricas poseen un costo asociado, debido a que se deben destinar recursos para su planificación, ejecución y análisis, y no siempre es factible disponer de esos recursos en un proyecto. Esto implica un análisis de costo-beneficio para evitar definir métricas que no aporten un valor a la organización/equipo/proyecto/etc. En ese análisis costo-beneficio, también influye la relación precisión-medición, ya que muchas veces no es necesaria tanta precisión. Esto también quiere decir que las métricas no sólo hay que calcularlas y dejarlas, hay que utilizarlas para que justifiquen ese costo. (El paso de “no medir” a “comenzar a medir” no tiene que ser excesivo, porque ese es otro de los factores que termina generando resistencia, por costo excesivo, ser una pérdida de tiempo en algo que después no se mira, etc.)

Las métricas no son para evaluar a la gente. No deben ser utilizadas para castigar o beneficiar a la gente, ya que es posible que las personas comiencen a fingir en el valor de las métricas para lograr objetivos finales, lo cual resulta perjudicial para el análisis del fenómeno que se desea medir.

En el software es complicado tener métricas de algunas características del producto, como por ejemplo la usabilidad del producto, por lo que no se mide directamente al producto, sino que se utilizan conceptos asociados (epifenómenos) a la característica que se desea medir. Por ejemplo, si se quiere medir el uso de una funcionalidad, se puede medir a través de la cantidad de clicks en el botón que permite ejecutar esa funcionalidad.

Por ejemplo, una métrica que dice cantidad de requerimientos tomado en función de la cantidad de requerimientos que deberíamos tomar, no es una métrica válida.

¿Para qué medir?

- Para señalar, controlar y supervisar el desarrollo del proyecto.
- Para predecir al estimar proyectos de software.
- Para evaluar, es decir, para tener una noción de los costos, por ejemplo.
- Para mejorar el proceso, el proyecto o el producto.

Dominio de métricas

Las métricas se dividen en tres dominios, el cual, cada uno posee un foco distinto de medición:

- **Métricas de proceso:** son métricas estratégicas a nivel organizacional, orientadas a mejorar el proceso y tienen como objetivo generar indicadores que permitan mejorar los procesos de software a largo plazo. Son públicas y se despersonalizan las mediciones de personas, áreas o proyectos particulares, para obtener un número aislado y tener una métrica sobre el proceso de la organización en general, como un todo.

Son responsabilidad del Ingeniero de Procesos, quien realiza las mediciones y luego publica los datos para que todos los empleados de la organización puedan acceder a ellos.

Por ejemplo, se toman métricas de cantidad de defectos en todos los productos que se realizaron, se despersonalizan, se promedian a cada una de ellas y se publican a toda la organización sin hablar particularmente de un proyecto o producto, sino en forma general al proceso de la organización.

En este enfoque, se tiene en cuenta que la experiencia de cada proyecto es extrapolable, por eso hacen uso de esta forma de tomar las métricas, desvinculándose de un producto o proyecto particular para hablar del comportamiento de la organización.

Los indicadores de proceso permiten tener una visión profunda de la eficacia de un proceso, determinando qué funciona de acuerdo a lo esperado y qué no.

Proporcionan beneficios significativos a medida que la organización trabaja para mejorar su nivel global de madurez del proceso.

Ejemplos:

- Desviación organizacional de estimaciones.
- Defectos por severidad en productos de la organización.
- Errores previos a releases por proyecto.
- Defectos detectados por usuarios.
- Esfuerzo realizado.
- Tiempos de planificación promedio por proyectos.
- Propagación de errores de fase a fase.

- **Métricas de proyecto:** están enfocadas a los recursos que se dedican al proyecto, como costos, esfuerzos, estimaciones y tiempo. Son responsabilidad del Líder de Proyecto y permiten al equipo adaptar el desarrollo de los proyectos y de las actividades técnicas. Estas métricas son privadas de ese proyecto y solo son visibles para los involucrados en el mismo.

Se utilizan para mejorar la planificación del desarrollo, generando ajustes que eviten retrasos, reduzcan riesgos potenciales y por lo tanto, problemas.

Además, se utilizan para evaluar la calidad de los productos en todo momento y en caso de ser necesario, modificar el enfoque para mejorar la calidad, minimizando defectos, retrabajo y por ende el costo total del proyecto.

Las métricas de proyecto se consolidan con el fin de crear métricas de procesos que sean públicas para la organización de software como un todo.

Ejemplos:

- Eficiencia de estimación del proyecto.
- Costos estimados versus costos reales.
- Esfuerzo/Tiempo por tarea del Ingeniero de Software.
- Errores no cubiertos por hora de revisión.
- Fechas de entregas reales versus programadas.
- Cantidad de cambios y sus características.

- **Métricas de Producto:** están enfocadas en lo que se construye, son responsabilidad del equipo de desarrollo y de Testing y son particulares de ese producto.

Se utilizan con propósitos técnicos y tienen como objetivo generar indicadores en tiempo real de la eficacia del análisis, el diseño, la estructura del código, la efectividad de los casos de prueba y calidad del software a construir.

Se deben controlar los artefactos resultantes del proceso de desarrollo (componentes y modelos) para garantizar:

- Que cumplan con los requerimientos del cliente;
- Que cumplan con los requerimientos de calidad;

- Que estén libre de errores;
- Que se realizaron bajo los procedimientos de calidad.

Ejemplos:

- Cantidad de líneas de código del producto;
- Defectos por severidad de un producto;
- Promedio de métodos por clase;
- Cantidad de métodos de cada clase;
- Cantidad de casos de uso por complejidad.

Métricas en el enfoque tradicional

En el enfoque tradicional se hace énfasis en los 3 dominios arriba mencionado.

Están basadas en la gestión de proyectos definidos, poseen un abanico de métricas mayor a los demás enfoques.

Son una disciplina transversal. Cuando se planifican los proyectos, allí se definen qué métricas se utilizarán, quiénes serán responsables de estas, cómo se calcularán, etc.

No todas las métricas le interesan a todo el mundo, dependiendo del momento y rol es lo que interesa medir. Los perfiles más técnicos por ejemplo apuntan a cuestiones relacionadas con el producto y esfuerzo (porque son ahí donde los desarrolladores asumen compromiso) y en ámbitos organizacionales interesa plata (costo) y tiempo. Por ejemplo:

- Testers: les interesa el esfuerzo y cantidad de defectos encontrados principalmente, porque es lo que más afecta a este rol.
- Líder de proyecto: El enfoque de las métricas estará más relacionado al calendario, los costos, plazos de tiempo a cumplir, etc. por el contrato que se tiene con el cliente, y la relación entre el esfuerzo y el tiempo.

Métricas básicas para un proyecto de software

son las mínimas e imprescindibles que uno tiene que utilizar si tiene la intención de desarrollar una cultura de medición:

- Tamaño del producto: asociada a métrica de producto.
- Esfuerzo: asociada a métrica de proyecto.
- Calendario: asociada a métrica de proyecto.
- Defectos: asociada a métrica de producto.

Estas métricas están relacionadas con la triple restricción en un proyecto (esfuerzo, alcance y tiempo).

Métricas en ambientes ágiles

Las métricas sirven para un equipo y no son extrapolables por lo que dice el agilismo a otros proyectos o equipos. En ágil hay un principio que habla específicamente de métricas (la mejor métrica de progreso es el software funcionando). Este principio apunta a que vamos a medir producto (no proceso ni proyecto). Esto es una reacción a proyectos tradicionales que tenían todas las métricas posibles cubriendo los tres enfoques, pero no teniendo avances sobre el producto (se perdía mucho tiempo y no se progresaba). Solo se debe medir lo que sea necesario y nada más, es decir, lo que agregue valor para el cliente.

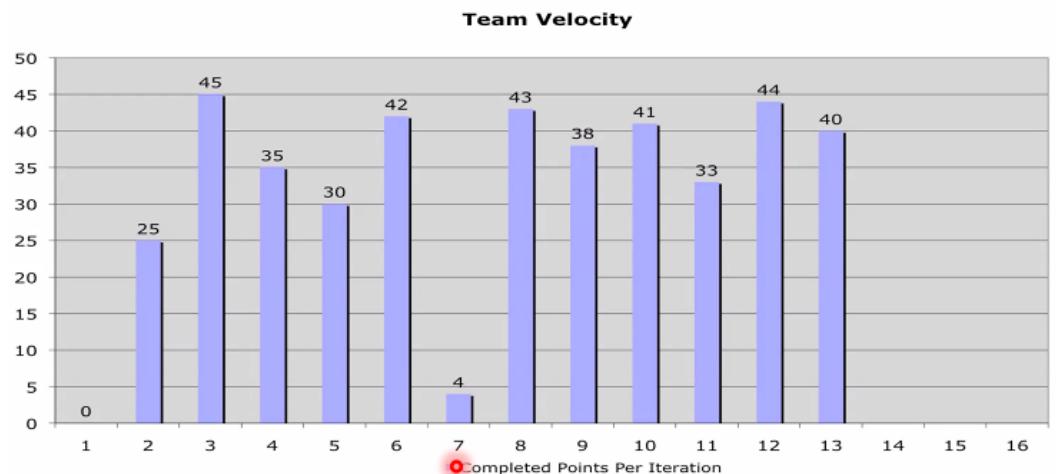
Métrica de velocidad

Es la métrica más importante del enfoque (métrica de producto), y mide la cantidad de puntos de historia que se realizaron en un Sprint y fueron aceptados por el Product Owner. Hay que recordar que no se cuentan las historias parcialmente terminadas, sólo las que están completadas y aceptadas por el PO.

Esta métrica se calcula (no se estima) luego de que el PO. acepte la implementación de una US.

Esta métrica suele representarse con un gráfico de barras para medir la estabilidad del equipo. Estos gráficos son permanentes durante todo el proyecto y son visibles para todo el mundo.

Esta métrica, y sus valores a lo largo de un proyecto ágil, permiten analizar si el equipo posee una estabilidad a lo largo del mismo, lo cual también se relaciona con un principio del manifiesto ágil (desarrollo sostenible).



Métrica de capacidad

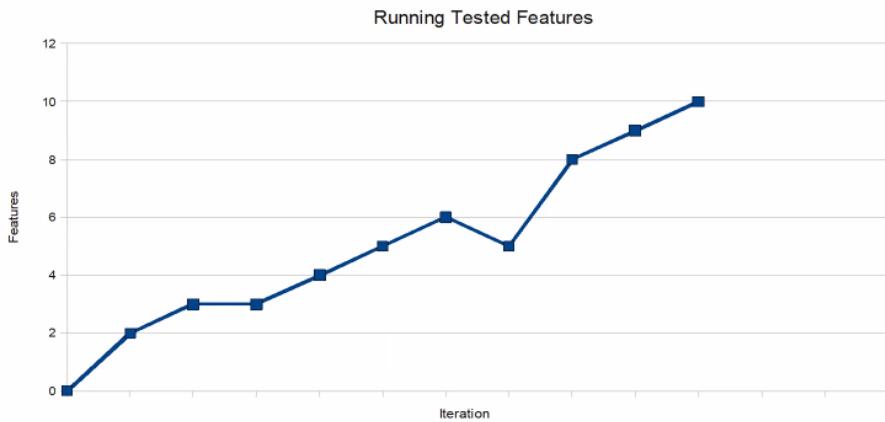
Es una métrica de proyecto que mide el compromiso de un equipo para un determinado sprint, en horas de trabajo ideales. Es por esto que, se utiliza para planificar, ya que permite definir cuántas historias de usuario se van a tomar del Product Backlog para implementar en el próximo sprint.

Se estima al principio de un sprint, por lo que también se la llama velocidad estimada. Se puede estimar en horas ideales o en Story Points.

Running Tested Features (RTF)

Mide la cantidad de features testeadas que están funcionando, es decir, cuantas piezas de producto (historias de usuarios, casos de usos, requerimientos) se terminaron y están en ejecución.

El problema de esta métrica es que no tiene en cuenta la complejidad de las piezas de producto que se implementan. Por ejemplo, si se toma como piezas de producto a historias de usuario, se mide cuántas historias de usuarios están funcionando, pero no se sabe de cuantos puntos de historias tiene cada una de ellas. Es una medición absoluta.

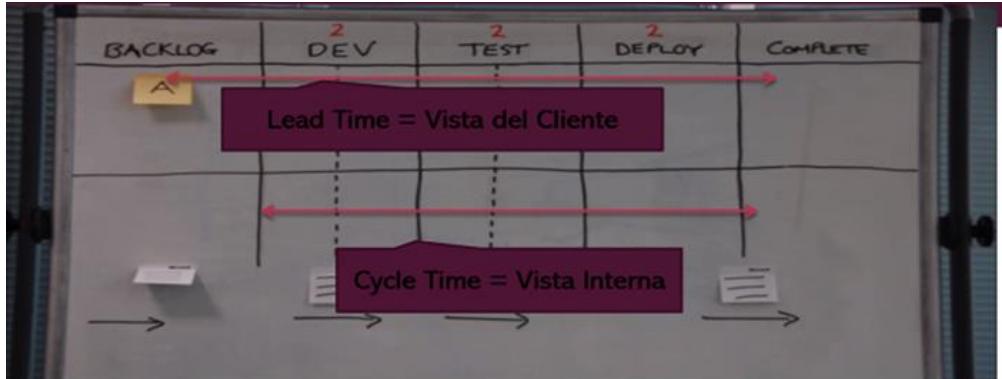


Si la curva es constante (llana) o tiene pendiente negativa, entonces indica la existencia de un problema, ya que las características no incrementan o disminuyen en el tiempo. Esta última situación se presenta cuando una funcionalidad del sistema deja de ser válida.

Las métricas de defectos también son necesarias en todos los ambientes.

Métricas de software en Lean (KANBAN)

El foco de medición de Kanban es el proceso, ya que el sistema de trabajo de Kanban es introducir mejoras en un flujo de trabajo continuo, es decir, no existe un proyecto con principio y fin. Es decir, se mide el comportamiento del proceso en función al tiempo que se demoran las características.



Lead Time o Elapsed Time

Métrica de vista o perspectiva del cliente. Es la más importante para el cliente. Mide desde el momento en que el cliente me pide algo (entra al Backlog) hasta que yo se lo entrego.

Cycle Time

Mide el tiempo desde que el equipo comenzó a trabajar sobre una funcionalidad pedida, hasta que se lo entrega, eliminando el tiempo de espera en el Backlog. Por esto se la considera como una vista interna, que sirve al equipo de trabajo y no es tan relevante para el cliente. Es igual al Lead Time menos el tiempo que estuvo en el Backlog.

Touch Time

Mide los tiempos en los cuales las personas están efectivamente trabajando sobre una unidad de trabajo (columnas de producción), sin tener en cuenta aquellos tiempos de espera (columnas de acumulación).

En cada columna, por ejemplo, en la columna Test de la imagen, se tiene una columna de trabajo y otra de acumulación para poder realizar el concepto de sistema “pull” o de arrastre.

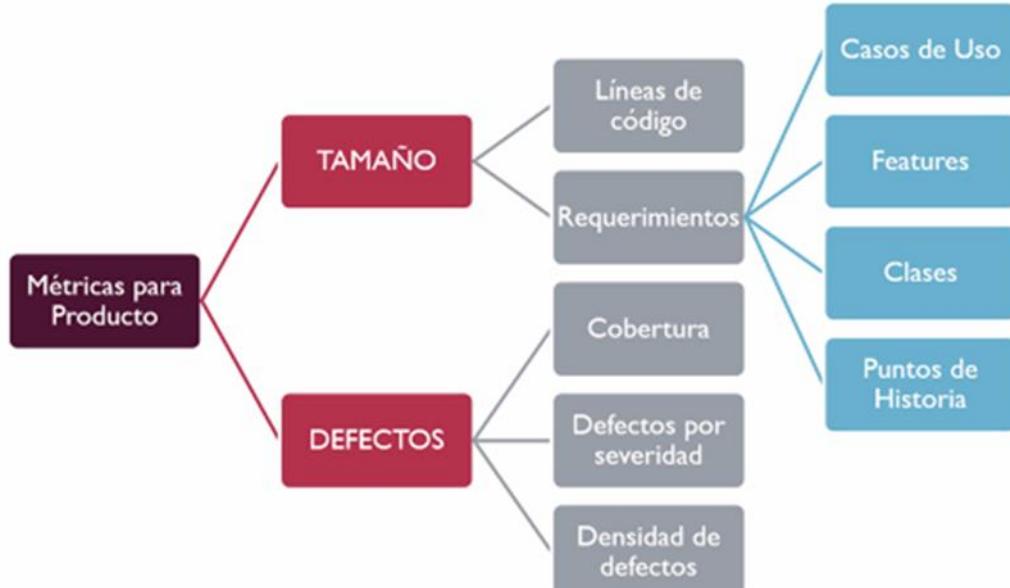
$$\text{Touch Time} \leq \text{Cycle Time} \leq \text{Lead Time}$$

Eficiencia del ciclo de proceso

Esta métrica mide la eficiencia del tiempo para entregar un trabajo, respecto al tiempo destinado a su ejecución. Se calcula como Touch time / Lead time.

Mientras más cercano a 1 sea su valor, más eficiente es el proceso, ya que todo el tiempo se estuvo trabajando sobre alguna funcionalidad, y no hubo muchos momentos de espera en columnas de acumulación.

Métricas de Producto de Software



Hoy en día no es recomendable medir en líneas de código.

Defectos

- Cobertura: tratar de que sea la mayor posible para cubrir el 100% del análisis de los defectos del producto.
- Defectos por severidad: cantidad de defectos por escala de severidad de cada defecto. 5 defectos graves.
- Densidad de defectos: cuantos defectos se encuentran por historia de usuario.

Resumen métricas en cada enfoque

<u>Tradicional</u>	<u>Ágil</u>	<u>Lean</u>
✓ Esfuerzo	✓ Velocidad	✓ Lead time – Elapsed time
✓ Tiempo	✓ Capacidad	✓ Cycle time
✓ Costos	✓ Running Tested Features	✓ Touch time
✓ Riesgos		✓ Eficiencia de proceso

Gestión de productos de software – Planificación de productos – herramientas para definición de productos de software

Un producto de Software es un artefacto que se construye para satisfacer una necesidad específica en base a ciertos requerimientos.

Cuando creamos un producto debemos enfocarnos en aquello que es de valor para el cliente, sabiendo que el 7% de las características del software son siempre usadas. Esto implica, que la mayor parte del valor agregado está concentrado en ese 7%. Alrededor del 80% de las características del software que se desarrolla no se utilizan. El desafío está en encontrar ese 7%.

Las organizaciones que usan el software se enfocan en la parte superficial del software, aquello que consideran significativo y placentero. Usualmente no se fijan si el producto es funcional o confiable. Esa debe ser la base, pero el producto debe proveer una experiencia que cumpla con todas las expectativas del usuario.

En resumen, nosotros creamos productos para:

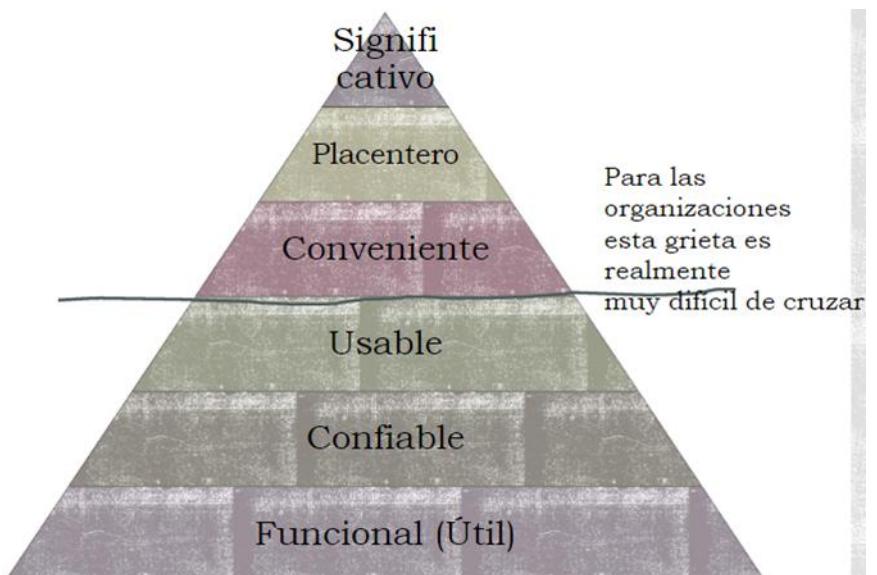
- Satisfacer a los clientes.
- Tener muchos usuarios logueados. Una métrica que se utiliza para saber que tan aceptado es un producto, es por la cantidad de usuarios que lo instalan.
- Para obtener mucho dinero.
- Realizar una gran visión, cambiar el mundo.

Evolución de los productos de software

Cuando vamos a desarrollar un software la mayoría de las veces se empieza a construir un producto sobre la base de lo funcional, que haga lo que tiene que hacer (las tareas que debe realizar), luego si se quiere avanzar más, el software debe ser confiable donde los usuarios no corran peligro usando el producto y los resultados que este producto me da yo pueda depender de ellos sin ningún problema. Luego, la usabilidad tiene que ver con aspectos de la disciplina UX/UI, donde el usuario podrá lograr sus objetivos productivamente usando este producto, estando a gusto con su uso. Tiene que ser productivo, no es lo mismo que la utilidad.

Por ejemplo, si tarda 12 horas en realizar una operación donde el anterior tarda 4 horas no es productivo para mí y no mejora la calidad de vida al usuario.

Existe una línea en la pirámide que separa la usabilidad de la conveniencia. La mayoría de los productos no se ubican por encima de esta línea, debido a que resulta complejo que un software sea conveniente para el desempeño de tareas de ciertas personas (es decir, que el producto nos haga sentir que lo necesitamos porque nos produce un cambio, una facilidad para cumplir con un objetivo), y que estas no sean “esclavos” del software, limitándose a cumplir con las características de funcional, confiable y/o usable. Placentero y significativo para todo el mundo, algo que nos cambie la forma de usar algo o de vivir.



Planificación del producto ágil

En las metodologías ágiles, se tiene una visión del producto materializado en el Product backlog, que contiene todas las características de la funcionalidad que requiere el producto que han sido identificadas hasta el momento. Esta visión se descompone en releases, las cuales son las entregas frecuentes que se hará al cliente. Cada release se

construye a partir de sucesivas iteraciones.

Valor vs desperdicio

La productividad de un Startup no puede medirse en términos de cuánto se construye cada día, por el contrario, se debe medir en términos de construir lo correcto cada día y lo correcto es aquello que agrega valor. Si no construimos valor, entonces estamos invirtiendo esfuerzo en desperdicio.

Mínimo Producto Viable

Es un concepto de Lean Startup que enfatiza el impacto del aprendizaje en el desarrollo de nuevos productos. Se define como **la versión de un nuevo producto que permite a un equipo recopilar la cantidad máxima de aprendizaje validado sobre los clientes con el menor esfuerzo**.

A este concepto le subyace la idea de que ver lo que la gente realmente hace con el producto es mucho más confiable que preguntarle a la gente qué harían.

Una premisa clave detrás de la idea del MVP es que usted produce un producto real que puede ofrecer a los clientes y observar su comportamiento real con el producto o servicio.

Entre sus características se destaca que:

1. Tiene el **valor suficiente para que las personas estén dispuestas a usarlo** o comprarlo.
2. Demuestra beneficios a futuro para retener a los usuarios.
3. Proporciona un ciclo de **retroalimentación** para guiar el desarrollo a futuro.

Los errores comunes son:

1. Confundirlo con MMF o MMP. La MMF (Característica Comercializable Mínima) o el MMP (Producto Comercializable Mínimo) a diferencia del MVP se enfocan en **ganar**, en cambio un MVP se enfoca en el aprendizaje
2. Enfatizar la parte de mínima y dejar de lado la parte de viable. Si el producto no tiene la calidad suficiente, no será posible obtener la retroalimentación del cliente.
3. Hacer del MVP un producto final, es decir, entregar lo que consideran un MVP, y luego no hacer más cambios a ese producto, independientemente de los comentarios que reciban al respecto.

Un MVP permite validar diferentes hipótesis:

1. **Hipótesis del valor:** prueba si el producto realmente está entregando valor a los clientes después de que comienzan a usarlo (ver si el producto es realmente el producto correcto). Una métrica de prueba: tasa de retención (más de la mitad de los usuarios volvieron a utilizar el sistema todos los días, solo un 25% dejó de usarlo).
2. **Hipótesis de crecimiento:** prueba cómo nuevos clientes descubrirán el Producto (ver si más personas desean usar el producto). Una métrica de prueba: tasa de referencia o Net Promoter Score (NPS) (Se corrió la voz acerca de lo que era, no fue siquiera necesario el uso de políticas de marketing o publicidad).



Preparar un MVP



26

- 1) Encontrar un nicho de mercado.
- 2) Roadmap: definición de muy alto nivel de qué características del producto se podrían tener a cierta altura del año. Esto **sirve para los inversores**.
- 3) Investigar la competencia: ver hasta dónde llega, qué hace, qué fortalezas tiene.
- 4) Pre-vender el MVP.
- 5) Testear las distintas suposiciones.
- 6) Asegurarse de que el MVP esté resolviendo el problema correcto.
- 7) Enfocarse solo en las funcionalidades principales.

Design Thinking

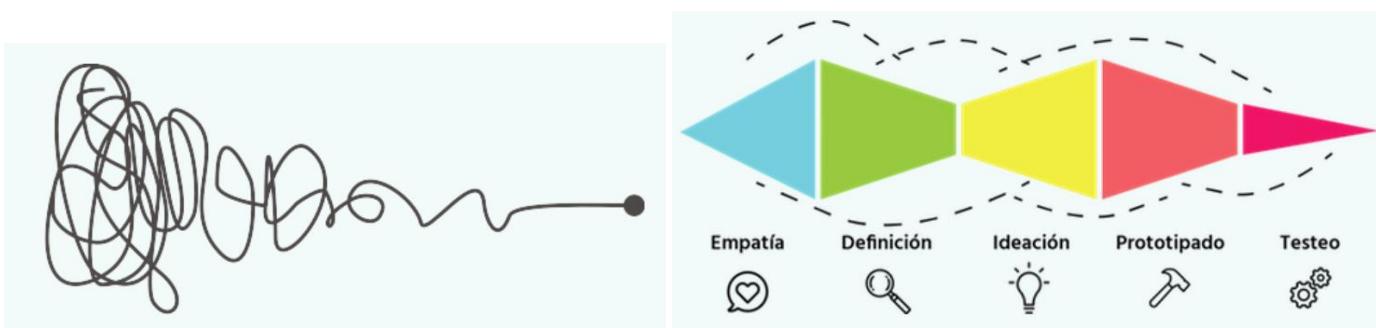
Es una metodología para la creación de servicios digitales centrada en el cumplimiento de las expectativas del usuario mediante un proceso de mejora continua que advierte las verdaderas necesidades de este.

Para llevar a cabo su proceso, utiliza un esquema de **pensamiento divergente y convergente**. Con el primero se busca tener una visión amplia del espectro de soluciones que responden al problema inicial. El segundo consiste en reducir a la mejor idea el conjunto de posibles soluciones, con el objetivo de brindar aquella que es más adecuada para las necesidades del cliente.

El proceso cuenta con 5 pasos y es iterativo (no lineal y se puede volver a los pasos anteriores las veces que sea necesario):

1. Empatizar: implica comprender las necesidades del cliente mediante la utilización del razonamiento intuitivo.
2. Definir: determinar lo valioso para el cliente a partir de lo recopilado durante la primera etapa, lo cual permitirá guiar el trabajo.
3. Idear: articular muchas soluciones posibles. Esto se puede lograr en los equipos multifuncionales, donde cada miembro aporta una mirada diferente para generar muchas ideas para luego filtrar las más viables.
4. Prototipar: consiste en materializar las ideas. Las ideas seleccionadas se transforman en prototipos que permitan visualizar la solución propuesta.

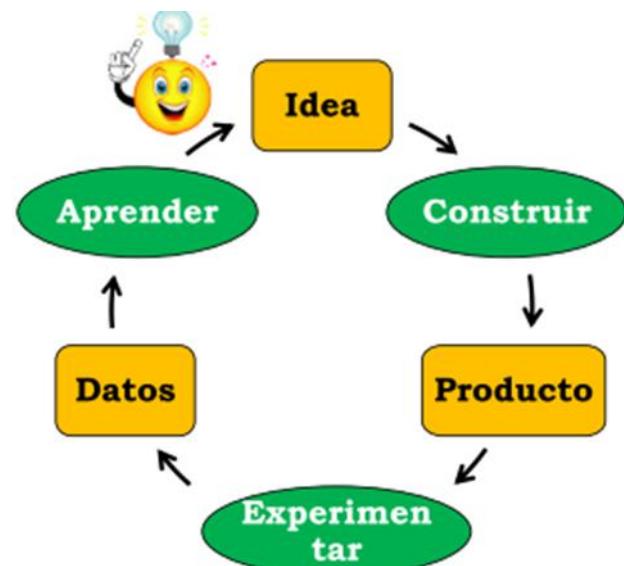
5. Probar: es una de las etapas más importantes del proceso, ya que permite validar los prototipos con los usuarios implicados en la solución, obteniendo retroalimentación y permitiendo iterar.



Lean UX

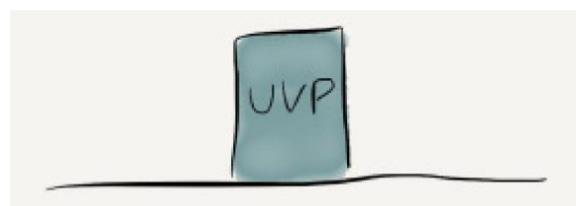
La filosofía Lean UX es un proceso iterativo, basado en el Build-Measure-Learn, es decir: construir, medir, aprender. Se trata de sacar conclusiones basándose en los usuarios reales que usan (o que van a usar) el producto/servicio. Si desde el primer momento el usuario no «aprueba» el producto, es más barato pivotar y volver a iniciar el proceso. Es decir, la metodología Lean UX nos permite «equivocarnos» en estadios muy precoces, cuando aún es muy barato reconducir la idea.

- El éxito no es entregar un producto, el éxito se trata de entregar un producto (o característica de producto) que el cliente usará.
- La forma de hacerlo es alinear los esfuerzos continuamente hacia las necesidades reales de los clientes.
- The Build-Experiment-Learn feedback loop permite descubrir las necesidades del cliente y alinearlas metodológicamente.
- La fase de construir MVP:
 - Ingresar lo más rápido posible un producto mínimo viable.
 - Un MVP varía en complejidad desde pruebas de humo extremadamente simples (no hay productos construidos aún) hasta prototipos tempranos (se ha construido un producto con características faltantes o problemas, vendrían a ser como las versiones beta de las aplicaciones).

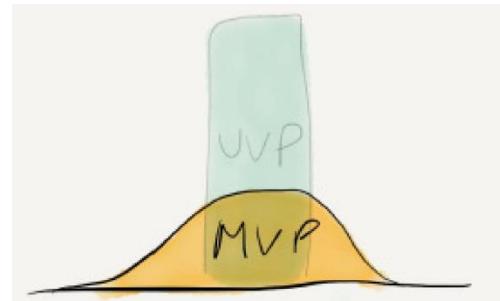


Explicación MVP, MVF, MMF con el dinosaurio Lean/Agile

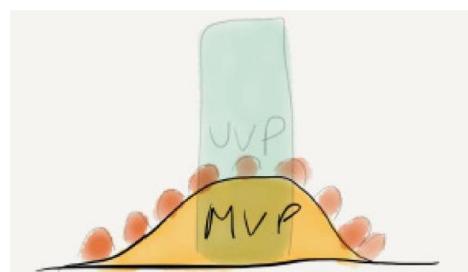
1. Comprender que un producto nuevo tiene una hipótesis de valor único: el Producto será único (UVP).



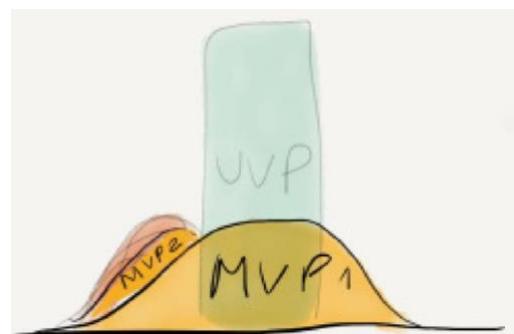
2. El siguiente paso es crear un producto mínimo viable (MVP) para probar su hipótesis. Esto se centra en su propuesta de valor única, pero normalmente también proporciona un poco de funciones de "Tablestakes" solo para asegurarse de que sea "viable" como producto.



3. Tu MVP también es una hipótesis. Podría ser lo suficientemente bueno para encontrar un mercado o no. Se muestra el caso en el que cada cliente potencial con el que interactúas te dice "Esto es genial, pero para poder usarlo necesito X" y X es diferente para cada cliente / usuario. Esto muestra que aún no se encuentra un mercado para el producto.



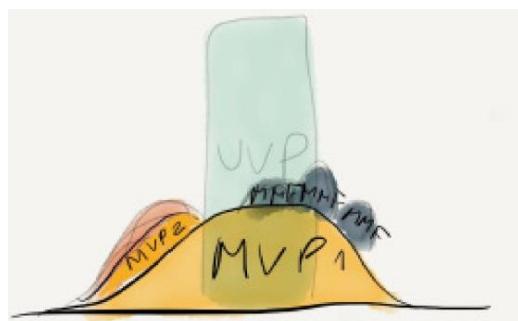
4. Si por el contrario ves cada vez más respuestas apuntando al MISMO X entonces tiene sentido revisar la hipótesis de Cliente/Problema/Solución. Básicamente, estás ejecutando un pivot. Está construyendo MVP2 centrado en la nueva hipótesis basada en el aprendizaje reciente de Desarrollo de clientes generado por el anterior MVP.



5. Supongamos que MVP2 es exitoso y está viendo una atracción real de los primeros usuarios. Si desea aumentar el crecimiento y busca una penetración más profunda de sus primeros usuarios además de atraer nuevos clientes, algunos de ellos más allá de la multitud de usuarios pioneros. En función del feedback recopilado y la investigación de su gestión de productos, tiene un par de áreas que potencialmente pueden traer este crecimiento. Algunos de ellos por cierto amplían su propuesta de valor única y algunos hacen que su producto actual sea más robusto.



6. En el caso de áreas con una fuerte indicación de valor, puede directamente definir un MMF (Características mínimas comercializables). Para encontrar la pieza mínima que pueda empezar a traer crecimiento. El objetivo del MMF es aportar valor. Supone una alta certeza de que existe valor en esta área y que sabemos cuál debe ser el producto para proporcionar este valor. La razón para dividir una característica grande en MMF más pequeños es principalmente el tiempo de comercialización (Time to market) y la capacidad de aportar valor en muchas áreas. Una indicación de que está trabajando en MMF es que cuando al liberarse uno se siente cómodo trabajando en el próximo MMF en esa área.



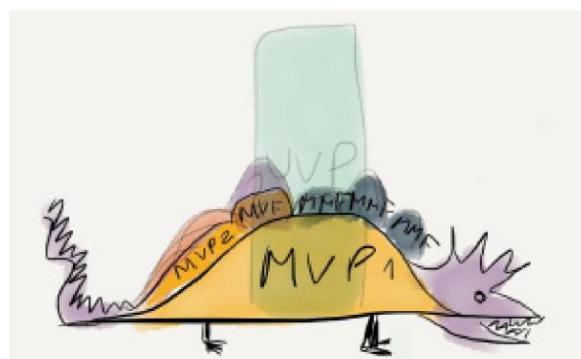
7. Si se desea esperar para ver si el primer MMF apesta ... entonces está de vuelta en la tierra de la hipótesis. Ahora tu hipótesis se centra en una característica en lugar del producto. Tienes un área con alto potencial pero también alta incertidumbre.



8. La forma de afrontarlo es crear una función "pionera": MVF (Característica Mínima Viable). La característica mínima que aún puede ser viable para uso real y aprendizaje de los usuarios reales. Si el MVF resulta exitoso (hit gold), puede desarrollar más MMF en esa área para tomar ventaja (si eso tiene sentido). Si no es así, puede cambiar a otro enfoque hacia esa área de características, o en algún momento buscar una ruta de crecimiento alternativa. Esencialmente, el MVF es una versión mini del MVP.

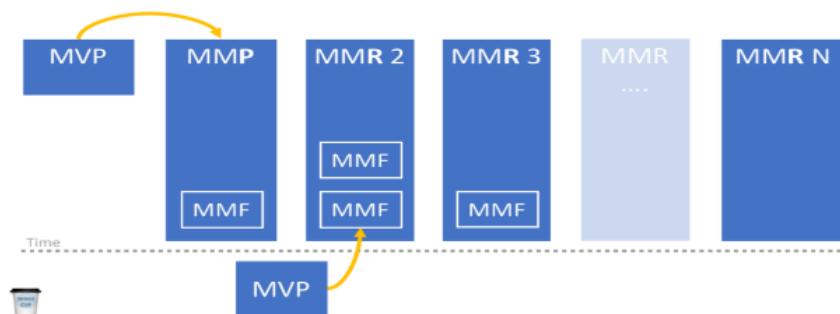


9. El producto se cultiva en mercados inciertos al intentar varios MVP. Cuando se logra ajustar el producto en el mercado de productos se combinan MMF y MVF según el nivel de incertidumbre del negocio / requisitos en las áreas en las que se está enfocando. Si bien los MVP / MMF / MVF son atómicos desde una perspectiva empresarial (no puede implementar y aprender de algo más pequeño) pueden ser bastante grandes desde la perspectiva de la implementación. El dinosaurio carpaccio se obtiene cortando cada una de esas piezas en pequeñas porciones destinadas a reducir el riesgo de ejecución / tecnología (normalmente se denominan User Stories). Esas porciones más pequeñas pueden tener un valor comercial tangible o no. Para otros puede ser importante proporcionar una temprana retroalimentación de decisiones a lo largo del camino.



Productos mínimos para la gestión de productos

1. Producto Mínimo Viable (MVP). Tanto esta como el MVF responden a hipótesis.
2. Característica Mínima Viable (MVF), es una versión mini del MVP.
3. Característica Mínimas del Release (MRF). Son las características mínimas debe tener un producto para salir al mercado. Es una versión del producto que se publica y que el cliente puede usar.
4. Característica Mínima Comercial (MMF).



Relación entre MVP, MMF, MMP, MMR

1. MVP

- a. Versión de un nuevo producto creado con el menor esfuerzo posible.
- b. Dirigido a un subconjunto de clientes potenciales.
- c. Utilizado para obtener aprendizaje validado.
- d. Más cercano a los prototipos que a una versión real funcionando de un producto.
- e. Es un concepto de Lean Startup que enfatiza el impacto del aprendizaje en el desarrollo de nuevos productos.
- f. Una premisa clave detrás de la idea de MVP es que usted produce un producto real que puede ofrecer a los clientes y observar su comportamiento real con el producto o servicio.
- g. Eric Ries: "versión de un nuevo producto que permite a un equipo recopilar la cantidad máxima de aprendizaje validado sobre clientes con el menor esfuerzo". Este aprendizaje validado viene en forma de si sus clientes realmente comprarán su producto.
- h. Ver lo que la gente realmente hace con respecto a un producto es mucho más confiable que preguntarle a la gente qué harían.

2. MMF

- a. Es la pieza más pequeña de funcionalidad que puede ser liberada (es una versión mini del MVP).
- b. Tiene valor tanto para la organización como para los usuarios.
- c. Característica a pequeña escala que se puede construir e implementar rápidamente, utilizando recursos mínimos, para una población objetivo para probar la utilidad y adopción de la característica.
- d. Es parte de un MMR o MMP.

3. MMP

- a. Primer release de un MMR dirigido a primeros usuarios (early adopters).
- b. Focalizado en características clave que satisfarán a este grupo clave.

4. MMR

- a. Release de un producto que tiene el conjunto de características más pequeño posible.
- b. El incremento más pequeño que ofrece un valor nuevo a los usuarios y satisface sus necesidades actuales.
- c. MMP = MMR1.

MVF (Minimal Viable Feature)

- Es una versión mini del MVP.
- Característica a pequeña escala que se puede construir e implementar rápidamente, utilizando recursos mínimos, para una población objetivo para probar la utilidad y adopción de la característica.
- Un MVF debe proporcionar un valor claro a los usuarios y ser fácil de usar.
- MVF requiere recursos mínimos, los estándares de calidad de la industria y la producción deben guiar el diseño y la confiabilidad.

- El grupo de usuarios para un MVF, son los primeros en adoptar, los clientes leales que han compartido conocimientos anteriormente o los miembros de una junta asesora de clientes. Usuarios flexibles y tolerantes.
- Los resultados le ayudarán a tomar decisiones estratégicas sobre productos.

Unidad 3: Gestión del Software como producto

Conceptos introductorios de la gestión de configuración

Introducción

Los sistemas de software siempre cambian durante su desarrollo y uso. Conforme se hacen cambios al software, se crean nuevas versiones del sistema. Los sistemas pueden considerarse como un conjunto de versiones donde cada una de ellas debe mantenerse y gestionarse. Esto es necesario para no perder la trazabilidad de los cambios que se incorporan a cada versión.

Definición de Gestión de Configuración

Es una disciplina de soporte, transversal a todo el proyecto con aplicación en diferentes disciplinas, que surge para enfrentar la crisis de problemas de software, aplicando dirección y monitoreo (administrativo y técnico) a:

- Identificar características técnicas y funcionales de ítems de configuración;
- Documentar características técnicas y funcionales de ítems de configuración;
- Controlar los cambios de esas características identificadas y documentadas;
- Registrar y reportar estos cambios;
- Verificar correspondencia con los requerimientos (trazabilidad).

Su propósito es resolver problemas de diferente índole, a través del establecimiento y el mantenimiento de la integridad del producto de software a lo largo de todo su ciclo de vida.

Algunos de esos problemas son:

- Pérdida de componentes;
- Pérdida de cambios (la versión del componente no es la última);
- Regresión de fallas;
- Doble mantenimiento;
- Superposición de cambios;
- Cambios no validados.

Su propósito es establecer y mantener la integridad de los productos de software a lo largo de su ciclo de vida. Esto implica identificar la configuración en un momento dado, controlar sistemáticamente sus cambios y mantener su integridad y origen.

La integridad es el medio por el cual podemos garantizar que el producto a entregar tiene la calidad correspondiente. Y nos garantiza un nivel mínimo de confiabilidad.

El problema de la calidad es que es subjetiva y se suma a que el software es intangible, por lo tanto, es muy difícil medir si la calidad que responde al cumplimiento de las expectativas del cliente realmente se verifica.

La idea de la integridad del producto es hacer explícita las características o expectativas que tiene el cliente sobre el producto de software.

Decimos que se mantiene la integridad de un producto de software cuando:

- Satisface las necesidades de usuario: hace lo que el usuario espera que haga;

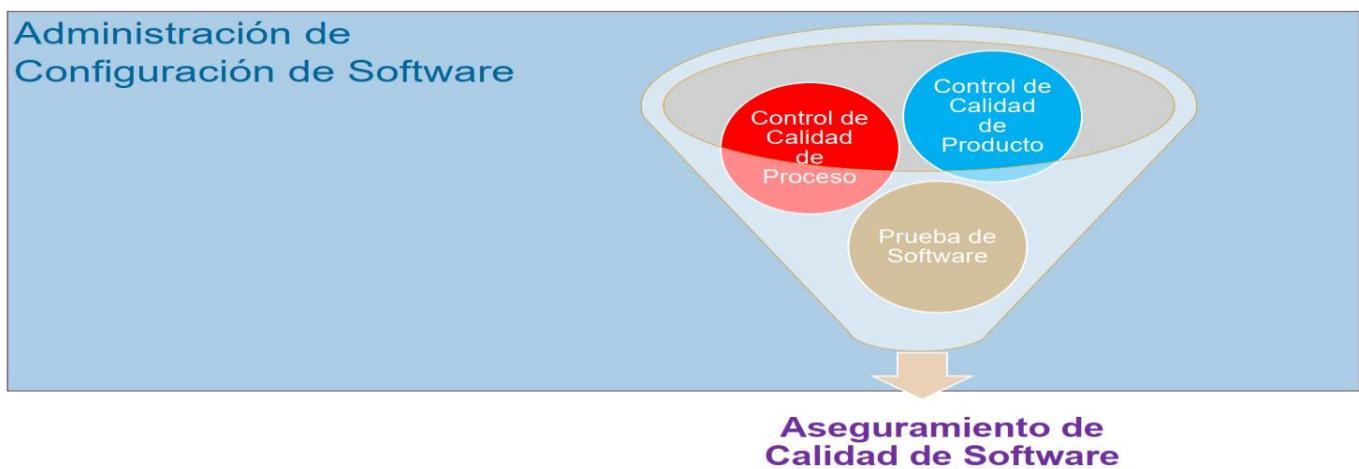
- Permite la rastreabilidad durante todo su ciclo de vida: existen vínculos o conexiones entre los ítems de configuración (los cuales deben ser definidos desde un primer momento del ciclo de vida del producto), que permiten analizar en donde impactará un cambio en un ítem de configuración. De esta forma, se puede determinar cómo impactará cada cambio de requerimientos a través de la trazabilidad. A mayor cantidad de vínculos → mayor información → mayor trazabilidad → mayor costo (analizar la relación costo-beneficio);
- Satisface criterios de performance y RNF;
- Cumple con expectativas de costo: este apartado incluye la satisfacción del equipo de proyecto. No solo el cliente debe estar satisfecho, sino también el desarrollo del producto debe ser redituable.

Mientras el producto existe hay actividad de gestión de configuración, que es responsabilidad de todo el equipo. Sin embargo, existen roles específicos como el rol del Gestor de configuración que tiene tareas adicionales como por ejemplo mantener la herramienta, marcar línea base, etc.

¿Cuál es el problema que se está tratando de solucionar?

El software es muy fácil de modificar, en el sentido de que uno puede entrar al repositorio y con un solo click eliminar meses de trabajo. Entonces lo que buscamos con la gestión de configuración del software es tratar de que estas cosas no pasen, buscando que el producto de software sea íntegro y si fuera el caso de que se van a realizar cambios, podamos tener un control de qué cosas cambiaron en un determinado momento. Es por esta razón que se asocia el concepto de software con el concepto de versión, ya que cada ítem de configuración debe tener su versión.

Y todas las personas se enteren de los cambios.



Conceptos Generales

Ítem de Configuración

Son aquellos artefactos que forman parte del producto o proyecto, y pueden ser almacenados en un repositorio, sin importar su extensión/tipo. Pueden sufrir cambios, y se desea conocer su estado y evolución a lo largo del ciclo de vida (ya sea del producto o proyecto, dependiendo del artefacto). Es decir, es cualquier aspecto asociado al producto de software (requerimientos, diseño, código, datos de prueba, documentos, etc.) que es necesario mantener. Son todos aquellos elementos que componen toda la información producida como parte del proceso de ingeniería de software, como ser programas de computadora (código fuente y ejecutables), documentos que describen los programas (documentos técnicos o de usuario), datos (de programa o externos).

Los ítems, dependiendo de su naturaleza, pueden durar lo que dure el ciclo de vida del proyecto, producto o sprint.

Algunos ejemplos: prototipo de interfaz, manual de usuario, ERS, arquitectura del software, casos de prueba, código fuente, íconos, etc, etc.

Versión

Instancia de un ítem de configuración que difiere de otras instancias de este ítem. Las versiones se identifican únicamente. Controlar una versión refiere a la evolución de un único ítem de configuración.

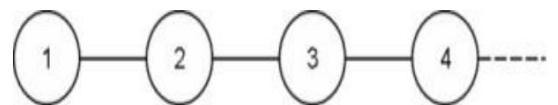
"La gestión de configuración permite a un usuario especificar configuraciones alternativas del sistema de software mediante la selección de las versiones adecuadas"; esto se puede gestionar asociando atributos a cada versión (que pueden ser datos sencillos como un nro. de versión asociado a cada objeto. Cada versión de software es una colección de elementos de configuración (ECS), como ser código fuente, documentos, datos).

La versión es un punto particular en el tiempo de ese ítem de configuración (es un estado).

Una versión se define, desde el punto de vista de la evolución, como la forma particular de un artefacto en un instante o contexto dado.

El control de versiones se refiere a la evolución de un único ítem de configuración (IC), o de cada IC por separado.

La evolución puede representarse gráficamente en forma de grafo.



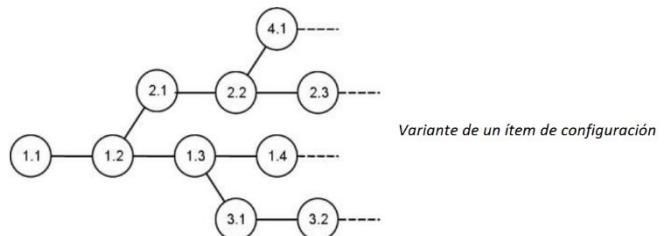
Evolución lineal de un ítem de configuración

Control de versiones

Hace referencia a la evolución de un único ítem de configuración o de cada ítem de configuración por separado.

Variante

Versión de un ítem de configuración (o de la configuración) que evoluciona por separado. Las variantes representan configuraciones alternativas. Las variantes de un ítem de configuración permiten satisfacer requerimientos en conflicto, como por ejemplo la compatibilidad del producto con diferentes hardware o sistemas operativos. Si la diferencia que existe entre las instancias de un mismo ítem de configuración es muy pequeña, no se denomina versión, sino que se la conoce como variante. Está ligado a trabajar con Branches, donde el producto adopta distintas configuraciones, dependiendo de múltiples factores (problemas, SO donde se ejecutará, etc.).



Variante de un ítem de configuración

Configuración

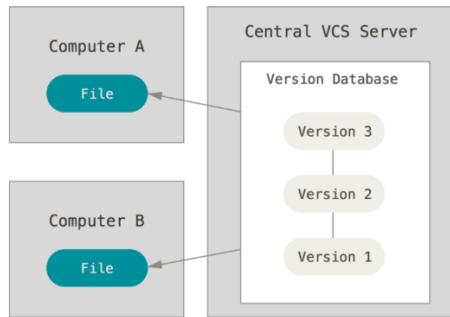
Conjunto de todos los ítems de configuración con su versión específica. Define una foto de los ítems de configuración en un momento de tiempo determinado.

Una configuración es el conjunto de todos los componentes fuentes que son compilados, sus documentos y la información de la estructura que definen una versión del producto a entregar. La configuración de un software es la sumatoria de todos los ítems de configuración que tiene en un momento determinado, equivale a una instantánea o una foto de todos los ítems de configuración con su versión en un momento del tiempo.

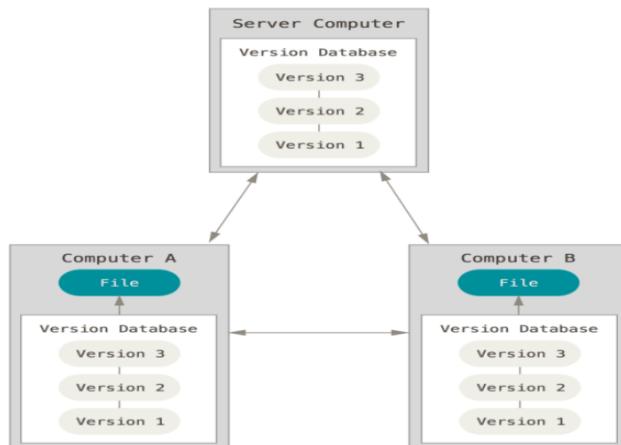
Repositorio

Es un contenedor de ítems de configuración, se encarga de mantener la historia de cada ítem con sus atributos y relaciones, además es usado para hacer evaluaciones de impacto de los cambios propuestos. Puede ser una o varias bases de datos. Tiene una estructura para mantener el orden y la integridad. El que tenga una estructura ayuda a la seguridad, los controles de acceso, las políticas de Backup y todo aquello que se aplica sobre un repositorio. Tenemos 2 tipos de repositorios:

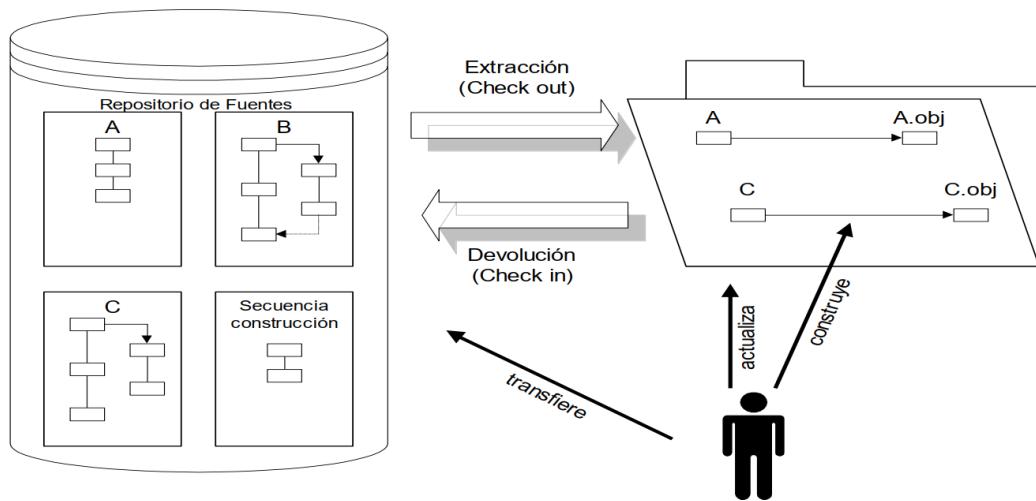
1. **Repositorios Centralizados:** está caracterizado porque un servidor contiene todos los archivos con sus versiones. Los administradores tienen un mayor control sobre el repositorio. Tiene como desventaja qué si falla el servidor, todo lo positivo se cae.



2. **Repositorios Descentralizados:** está caracterizado porque cada cliente tiene una copia exactamente igual del repositorio completo. Tiene como ventaja que se resuelve el problema de los servidores centralizados, debido a que si el servidor falla sólo es cuestión de “copiar y pegar”, además posibilita otros workflows no disponibles en el modelo centralizado. La desventaja que podemos mencionar es que es más complicado llevar un control sobre el mismo.



Funcionamiento de un repositorio



Release

Entrega de un sistema que se libera para su uso a los clientes u otros usuarios de la organización.

Rama (Branch)

Es un conjunto de ítems de configuración con sus correspondientes versiones, que permiten bifurcar el desarrollo de un software, por varios motivos:

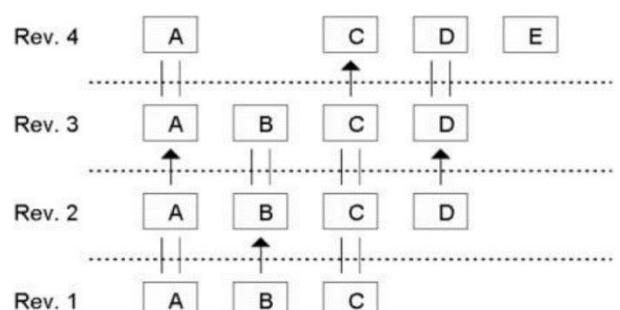
- Experimentación;
- Resolución de errores en el desarrollo;
- Desarrollar un mismo software para diferentes plataformas.

Integración de ramas (Merge)

Se da cuando se integran dos ramas, para fusionar la configuración de los ítems de configuración con sus correspondientes versiones, en cada una de ellas. Una buena práctica es mantener la rama principal como la versión estable, y fusionar los cambios hacia ella. En caso de no integrarse a la Main, las ramas deberían descartarse.

El rol de las líneas base y su administración

Es un conjunto de ítems de configuración que han sido construidos y revisados formalmente, de manera que pueden ser tomados como referencia para demostrar que se ha alcanzado cierto nivel de madurez en ellos, y que sirve como base para desarrollos posteriores. Esto es lo mismo que decir que es una configuración de software que ha sido formalmente revisada y aprobada, que sirve como base para desarrollos futuros.



Este conjunto de ítems debe tener una referencia única, y esto se hace a través de "tags".

Se acuerdan parámetros para determinar qué se considera o qué debe cumplir los ítems de configuración para considerarse como línea base.

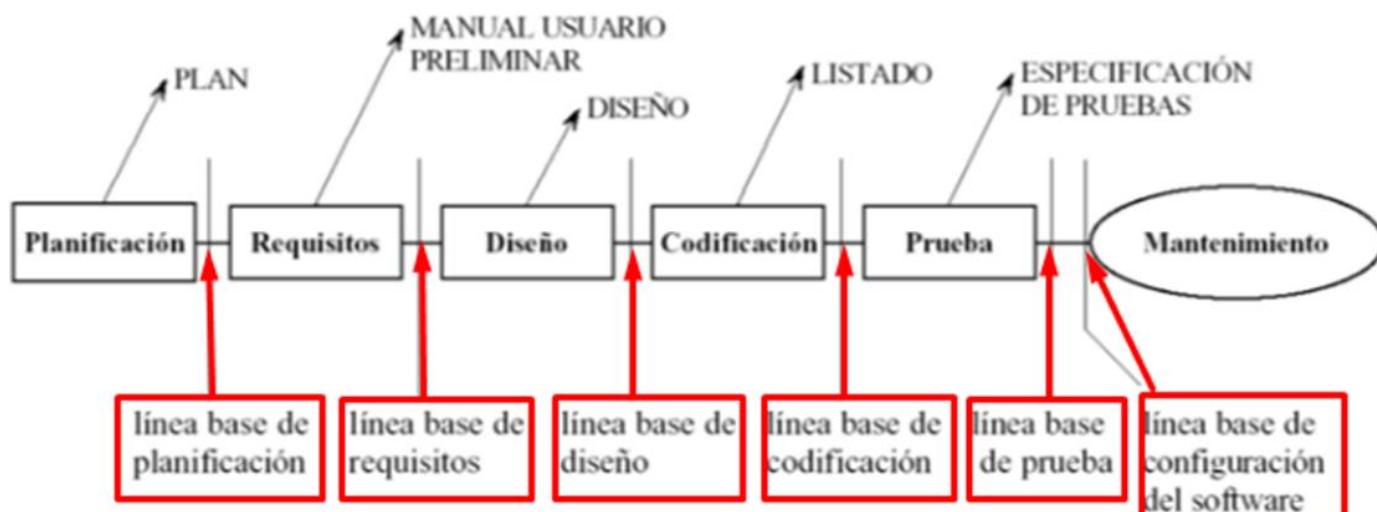
Si se desea cambiar una línea base, existe un protocolo formal de control de cambios, dirigido por el Comité de Control de Cambios, que permite definir el procedimiento a seguir para manejar peticiones de cambios, y en caso de aceptarse, acordar nuevamente los parámetros y comunicar los cambios a todo el equipo, para que tomen la nueva línea base como modelo a seguir.

¿Para qué sirve una línea base?

Fundamentalmente es para tener un punto de referencia, pero también sirve para hacer Rollback o saber qué se pone en producción. Nos permite saber cuál era la última situación estable en un momento de tiempo y cómo se fue evolucionando. Permiten ir atrás en el tiempo y reproducir el entorno de desarrollo en un momento dado del proyecto.

Existen dos tipos de línea base:

1. **Operacionales:** contiene una versión de producto cuyo código es ejecutable, han pasado por un control de calidad definido previamente. La primera línea base operacional corresponde con la primera Release. Es la línea base de productos que han pasado por un control de calidad definido previamente.
2. **De especificación o documentación:** son las primeras líneas base, dado que no cuentan con código. Podría contener el documento de especificación de requerimiento. Son de requerimientos, diseño.



Planificación de la gestión de configuración de software

Este plan se debe confeccionar al inicio de un proyecto, y existen diferentes estándares donde se expresa cómo planificar:

- Reglas de nombrado de los ítems de configuración;
- Herramientas para utilizar en SCM;
- Roles e integrantes del Comité de Control de Cambios;
- Procedimientos formales de cambios;
- Procesos de auditoría;
- Estructura del repositorio;

- SCM para software externo (opcional);
- Cómo se hará el control de cambios;
- Registros que deben mantenerse;
- Tipos de documentos.

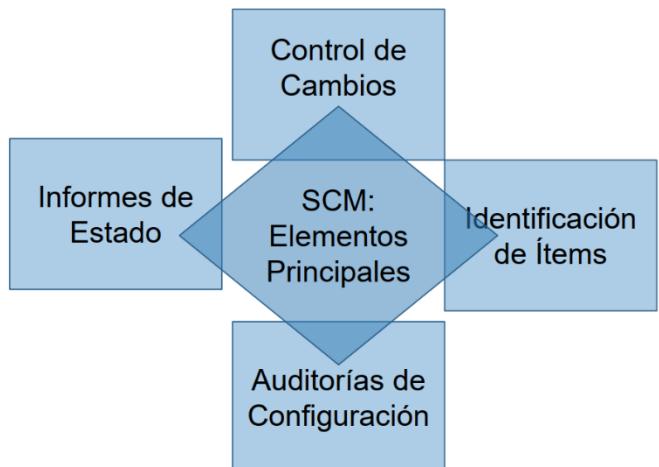
Consideraciones:

- Debe hacerse tempranamente;
- Se deben definir los documentos que se administrarán;
- Todos los productos del proceso deben administrarse.

Actividades relacionadas a la gestión de configuración

Estas actividades/elementos son las cosas que contienen las secciones de un plan de gestión de configuración. Recordemos que la gestión de configuración también se debe planificar.

Todas las actividades se realizan en ambas metodologías, menos las auditorías, ya que el concepto de ser auditado y controlado por alguien externo, no es compatible con una metodología ágil pura.



Identificación de Ítems de Configuración

esto implica una identificación única para cada ítem, donde en el equipo se definirán políticas y reglas de nombrado y versionado para todos ellos. También, se debe definir la estructura del repositorio, y la ubicación de los ítems de configuración dentro de esa estructura.

Además, implica una definición cuidadosa de los componentes de la línea base.

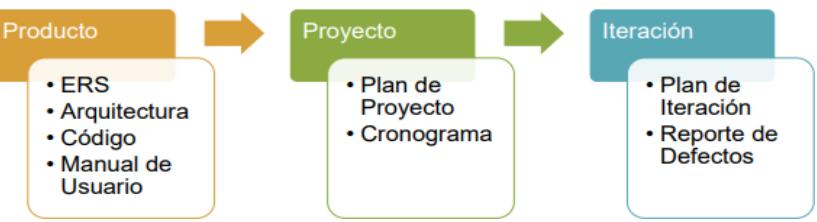
Esta identificación y documentación proveen un camino que une todas las etapas del ciclo de vida del software, lo que permite a los desarrolladores controlar y velar por la integridad del producto, como así también a los clientes evaluar esa integridad.

También se debe tener en cuenta al momento de identificar los ítems, la duración de su integridad, ya que difieren en función del tiempo que es necesario mantenerlos.

Se clasifican en:

1. Ítems de producto: tienen el ciclo de vida más largo, y se mantienen mientras el producto exista. Ejemplo: documento de arquitectura, casos de uso, código, manual de usuario y de parametrización, casos de prueba, una ERS, los casos de prueba, la base de datos.

2. Ítems de proyecto: el plan de proyecto, el listado de defectos encontrados, tienen un ciclo de vida de proyecto. Un plan de iteración, un Burn-Down Chart, se mantiene durante una iteración. La duración impacta también en el esquema de nombrado, plan de proyecto. Ejemplo: Los ítems de configuración a nivel de proyecto, el plan de proyecto y el cronograma.
3. Ítems de iteración: Conocer el ciclo de vida de un ítem permite establecer la nomenclatura de este. Ejemplo: planes de iteración, cronogramas de iteración, reporte de defectos.



Luego de identificar los ítems se debe armar la estructura del repositorio con nombres representativos. Luego se vinculan los ítems de configuración con el lugar físico del repositorio donde se van a almacenar.

Entonces, las actividades involucradas son:

- identificación única de cada ítem de configuración;
- convenciones y reglas de nombrado;
- definición de la Estructura del Repositorio;
- ubicación dentro de la estructura del repositorio.

Control de cambios

Una vez definida la línea base, no es posible cambiarla sin antes pasar por un proceso formal de control de cambios. Es decir que el control se hace sobre los ítems de configuración que pertenecen a la línea base, ya que todos los trabajadores del software tienen a dichos ítems como referencia.

Comité de control de cambios

Este proceso es llevado a cabo por el comité de control de cambios, el cual se reúne para autorizar la creación y cambios sobre la línea base. Forman parte de dicho comité los interesados en evaluar y en enterarse del cambio, decidiendo si lo aceptan o no. El líder del proyecto, el analista, el arquitecto e incluso el cliente pueden constituirlo. Realmente la integración del comité depende de la propuesta del cambio.

Etapas del proceso

Si el comité fijó una línea base luego de la revisión de los ítems que la conforman y llegada a la conclusión de que los mismos se encuentran en estado **estable**, en caso de recibir una propuesta de cambio, los pasos son:

1. Se recibe una **propuesta de cambio** sobre una línea base determinada, no sobre un ítem.
2. El comité de control de cambios realiza un **análisis de impacto** del cambio para evaluar el esfuerzo técnico, el impacto en la gestión de los recursos, los efectos secundarios y el impacto global sobre la funcionalidad y la arquitectura del producto.

3. En caso de que se autorice la propuesta de cambio, se genera una orden de cambio que define lo que se va a realizar, las restricciones a tener en cuenta y los criterios para revisar y auditar.

4. Luego de realizado el cambio, el comité vuelve a intervenir para aprobar la modificación de la línea base y marcarla como línea base nuevamente, es decir que realiza una revisión de partes.

5. Finalmente se notifica a los interesados los cambios realizados sobre la línea base.

Además, el control de cambios permite tener una trazabilidad entre los ítems de configuración, y ante un cambio saber cuáles ítems están afectados por este.

Auditorías de configuración

Son controles autorizados a realizar por el equipo de desarrollo en un momento determinado, donde un auditor externo al equipo (independiente y objetivo) analiza las líneas base, las cuales permiten “congelar” y analizar en un momento determinado cuál es el estado del software, y si se están cumpliendo todas las pautas que plantea esta disciplina de SCM.

En metodologías ágiles, esta es la única actividad de la disciplina SCM que no se soporta por la filosofía.

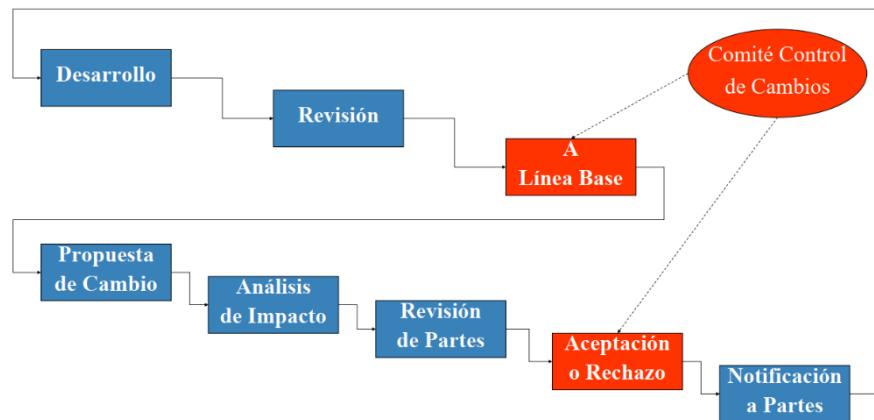
- Es objetiva cuando hay independencia de criterio, por lo que el auditor no tiene ningún condicionamiento respecto de lo auditado.
- Es independiente cuando no depende jerárquica, funcional ni salarialmente del equipo o elemento a auditar.

La auditoría de configuración se hace mientras el producto se está construyendo y su objetivo es que el producto tenga calidad e integridad. Se entiende que lo más barato es prevenir.

La auditoría de configuración complementa a la revisión técnica y consiste en revisar si se están realizando las tareas tales como se planificaron y especificaron en el plan de gestión de configuración. Las auditorías se realizan sobre una línea base. Es decir, la auditoría requiere la existencia de un plan de gestión de configuración y de la línea base a auditar.

Trazabilidad o rastreabilidad de requerimientos

Establecer una correspondencia de relación entre ítems de configuración de diferentes niveles de abstracción en el proceso de desarrollo de software. Una traza puede ser constituida por el requerimiento, sus modelos de análisis y de diseño, la implementación en código y los casos de prueba para validar y verificar el requerimiento. La trazabilidad permite realizar un análisis del impacto sobre el sistema a partir de una solicitud de cambio en el requerimiento. Los modelos de calidad exigen la trazabilidad bidireccional. Desde el requerimiento hacia el componente de código y viceversa.



Procesos a los que sirve

1. Validación: se encarga de asegurar que cada ítem de configuración resuelva el problema apropiado, es decir, lo que el cliente necesita.
2. Verificación: implica asegurar que un producto cumple con los objetivos definidos en la documentación de líneas base. Todas las funciones son llevadas a cabo con éxito y los test cases tengan status "ok" o bien consten como "problemas reportados" en la nota de reléase. Es decir, que se cumpla lo definido para cada ítem de configuración en la documentación de una línea base.

Las auditorías permiten visualizar si se están satisfaciendo los requerimientos y si se ha cumplido la intención de la línea base anterior. Ante un análisis, el auditor puede detectar defectos y ajustar correcciones.

Tipos de auditorías

1. **Auditoría física o de contingencia:** se realiza primero. Se validan los ítems de configuración de la línea base contra el plan de gestión de configuración, asegurando que la documentación que se entregará es consistente con el código desarrollado. Si esto no pasa, la auditoría funcional no se realiza.
 - Controla que el repositorio esté alojado en el lugar donde se dijo que iba a estar, que los IC respeten el esquema de nombrado, que estén guardados donde se dijo que iban a estar guardados.
 - Lo que se audita es una línea base.
 - Hace verificación.
2. **Auditoría funcional de configuración:** audita la trazabilidad de requerimientos, controlando que la funcionalidad y performance real de cada ítem de configuración sean consistentes con la especificación de requerimientos. El propósito final es validar que el código implementa únicamente y todos los requerimientos de la ERS.

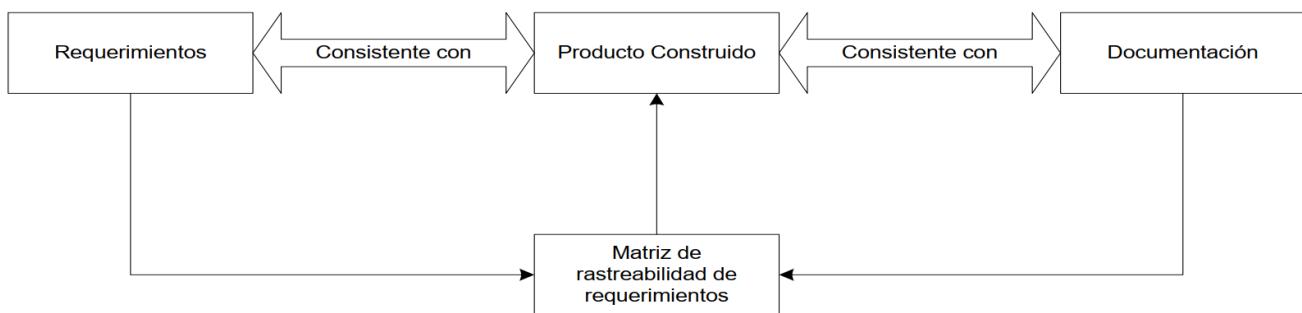
Evaluación independiente de los productos de software, controlando que la funcionalidad y performance reales de cada ítem de configuración sean consistentes con la especificación de requerimientos. Antes de comenzar, pregunta si está disponible el reporte de la auditoría de configuración física. Ya que, si la auditoría física no sale bien, la auditoría funcional no se hace.

 - Hace validación.

La gestión en ambientes ágiles se trabaja con integración continua (CI - Continuous Integration), un servidor donde se prueba automáticamente el código y luego se integra en producción continuamente. Un repositorio, un esquema de nombrado son cosas que si se utiliza en ágiles, mientras que la auditoría y el comité de control de cambios durante el sprint, pueden obviarse.

Auditoría Funcional de Configuración

Auditoría Física de Configuración



Beneficios:

- Aumenta la protección contra cambios innecesarios;
- Mejora de la visibilidad de estado del proyecto y sus componentes;
- Aumenta la auto responsabilidad;
- Disminuye los costos por retrabajo;
- Disminuye el tiempo de desarrollo;
- Aumenta la calidad;
- Suministra viabilidad y rastreabilidad del ciclo de vida del producto de software.

Desventajas:

- Quejas (es burocrático, molesto, se meten con mi trabajo);
- La transición es difícil;
- No hay compromiso en todos los niveles;
- No hay conciencia del problema.

Informe de estado

provee un mecanismo para mantener un registro de cómo evoluciona el sistema, y dónde está ubicado el software, comparado con lo que está publicado en la línea base. Esto permite mantener a todo el equipo informado sobre la última línea base, y que se trabaje en base a su última versión, y no sobre una versión obsoleta.

Estos reportes incluyen todos los cambios que han sido realizados a las líneas base durante el ciclo de vida del software. Normalmente esto consta de grandes unidades de datos, por lo que se utilizan herramientas automatizadas por una computadora.

El objetivo principal es asegurarse que la información de los cambios llegue a todos los involucrados.

El reporte más conocido es el de inventario, el cual provee una copia del contenido del repositorio con la estructura de directorios.

Permite responder a preguntas como:

- a. ¿Cuál es el estado del ítem?
- b. ¿La propuesta de cambio fue aceptada o rechazada por el comité?
- c. ¿Qué versión de ítem implementa un cambio de una propuesta de cambio aceptada?
- d. ¿Cuál es la diferencia entre dos versiones de un mismo ítem?

Gestión de configuración en ambientes agiles

En las metodologías ágiles la gestión de configuración cambia el enfoque, ya que la misma es de utilidad para los miembros del equipo de desarrollo y no viceversa.

En orden con la caracterización de los equipos ágiles, decimos que la gestión de configuración posibilita el seguimiento y la coordinación del desarrollo en lugar de controlar a los desarrolladores.

Los equipos ágiles son autoorganizados, por lo que las Auditorías de configuración no son una actividad propia de la gestión de configuración en los ambientes ágiles. Todos los procesos definidos establecidos en la gestión de configuración del enfoque tradicional son relativizados en agile. Por ejemplo, no existe un comité para el proceso forma de control de cambios.

Podemos decir que va de la mano con el manifiesto ágil porque recordemos que se buscaba estar siempre preparados para el cambio y justamente, la gestión de configuración busca responder a los cambios y mantener la integridad del producto.

La diferencia es que el manifiesto ágil se enfoca en los proyectos, mientras que la gestión de configuración de software se enfoca en el producto. Es decir, trasciende el proyecto.

Mientras el manifiesto ágil apunta a cómo trabajar en el contexto del proyecto de desarrollo y la gestión de configuración es una práctica específica del producto que garantiza la calidad del producto.

Características

1. Dada la naturaleza de los requerimientos ágiles, la SCM responde a los cambios en lugar de evitarlos.
2. La automatización debe ser aplicada donde sea posible. Esto colabora con la entrega frecuente y rápida de software (integración continua).
3. Provee retroalimentación continua sobre calidad, estabilidad e integridad.
4. Las actividades de SCM se encuentran embebidas en las demás tareas para alcanzar el objetivo del sprint.
5. Es responsabilidad de todo el equipo, por lo tanto, requiere capacitación.
6. Adopción de las prácticas continuas.

Continuous Integration

Es una práctica de desarrollo que promueve que los desarrolladores adopten la costumbre de integrar su código a un repositorio compartido varias veces al día. Cada integración de código es luego verificada por una serie de pruebas automatizadas, permitiéndole al equipo detectar problemas de manera temprana. Ya que al integrar el código al repositorio de manera frecuente resulta más fácil detectar y corregir los errores.

La integración continua es asegurar que el software pueda ser desplegado en cualquier momento, es decir, que el código compile y que sea de calidad.

Cada desarrollador en su entorno de trabajo realiza pruebas unitarias (en la medida de lo posible, automatizadas) desarrollando con algo que se llama TDD (desarrollo conducido por pruebas) y cuando terminó ese componente de código y lo probó y sabe que funciona, lo sube a un repositorio de integración.

Así, la versión del producto está en condiciones de ir a las pruebas de aceptación de usuario sin problemas.

Continuous Delivery

Es una disciplina de desarrollo de software en la que el software se construye de tal manera que puede ser liberado en producción en cualquier momento. Suma la automatización de las pruebas de aceptación y entonces el producto ya está listo para desplegarlo a producción.

No implica necesariamente que se libere cada vez que hay un cambio, sólo ocurre si el responsable de negocio, el Product Owner de turno, decide pasar a producción. Es decir que hay un componente «humano» a la hora de tomar la decisión. Pero, en cualquier caso, la versión está lista de inmediato. Esto implica, entre otros, que se prioriza que la versión esté en un estado en el que pueda ser puesta en producción.

La integración continua es prerequisito de la entrega continua. Con esto se refiere a que los artefactos producidos en el servidor de integración continua son puestos en producción con solo un click. Hay que asegurarse de tener un despliegue automatizado, para que cada puesta en producción no lleve demasiado tiempo, lo que hará que las puestas puedan llegar a hacerse más seguidas, generando que lo que se despliegue no sean demasiados cambios y el riesgo

de que algo se rompa disminuya. El software debe estar siempre en un estado de “entregable”, es decir, el software builda, el código se compila y los tests pasan.

Continuous Deployment – Estrategias de Deployments – Canary Deployments – Blue/Green Deployment

Consiste en poner en el ambiente de producción del usuario final el producto. A diferencia de la entrega continua, en el despliegue continuo no existe la intervención humana para desplegar el producto en producción. Para esto, se utilizan pipelines que contienen una serie de pasos que deben ejecutarse en un orden determinado para que la instalación sea satisfactoria.

El propósito de las estrategias es que sea transparente para el usuario que pusiste en producción una nueva versión del producto.

Se recomienda hacer el despliegue a poco tiempo de haber trabajado con los cambios en el código, pues un error en producción un día después de haberlo hecho significa que todavía nos acordamos de lo que hicimos y será más fácil de resolver.

Canary Deployment

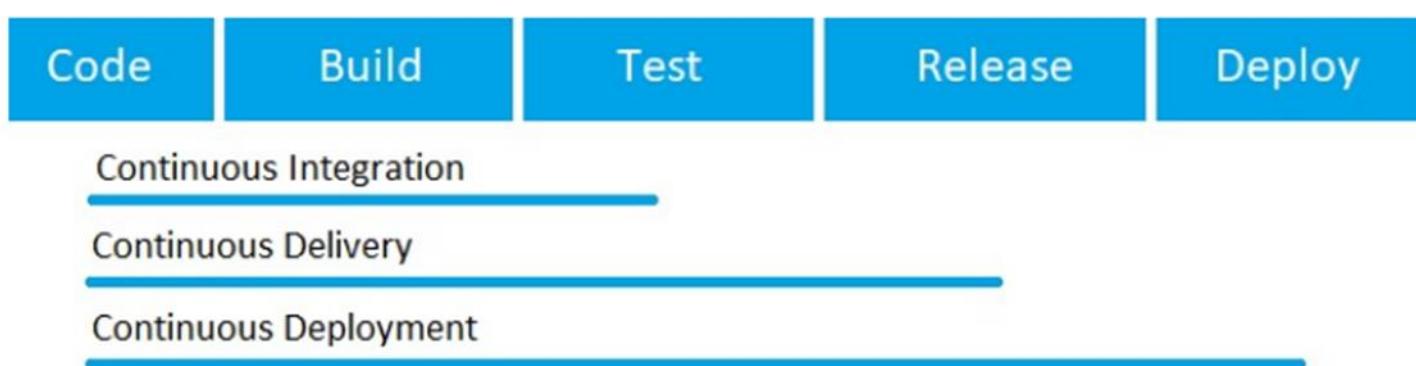
En ingeniería de software, Canary deployment es la práctica de realizar lanzamientos por etapas. Primero implementamos una actualización de software para una pequeña parte de los usuarios, para que puedan probarla y proporcionar comentarios. Una vez aceptado el cambio, la actualización se extiende al resto de usuarios.

Esta estrategia nos muestra cómo los usuarios interactúan con los cambios de la aplicación en el mundo real. Al igual que en Blue-Green Deployment, la estrategia Canary ofrece actualizaciones sin tiempo de inactividad y reverisiones sencillas. A diferencia de Blue-Green, las implementaciones Canary son más fluidas y las fallas tienen un impacto limitado.

Blue-Green Deployment

Es un modelo de lanzamiento de aplicaciones que transfiere gradualmente el tráfico de usuarios de una versión anterior de una aplicación o microservicio a una nueva versión casi idéntica, las cuales se ejecutan en producción.

La versión anterior puede denominarse entorno azul, mientras que la nueva versión puede denominarse entorno verde. Una vez que el tráfico de producción se transfiere por completo de azul a verde, el azul puede quedar en espera en caso de reversión o retirada de producción y se actualiza para convertirse en la plantilla sobre la que se realiza la próxima actualización.



La diferencia entre estas tres prácticas es el nivel de automatización de las pruebas.

Unidad 4: Aseguramiento de calidad de Proceso y de Producto

Conceptos generales sobre calidad

Importancia de trabajar para y con Calidad. Ventajas y Desventajas.

Calidad

Formalmente se puede definir como el cumplimiento de los requerimientos funcionales y de performance explícitamente definidos, de los estándares de desarrollo explícitamente documentados y de las características implícitas esperadas del desarrollo de software profesional.

Todos los aspectos y características de un producto o servicio que se relacionan con la habilidad de alcanzar las necesidades manifiestas o implícitas. Está relacionado con MIS necesidades y MIS expectativas (se relaciona con las necesidades implícitas ya que no son expresadas).

Está directamente relacionada con la persona (es subjetiva a quién la analice), las circunstancias, el contexto, la edad en un momento de tiempo dado (puede para mí algo no tener calidad, pero para otra persona sí).

Gestión de calidad

La gestión de calidad del software para los sistemas de software tiene tres intereses fundamentales:

- A nivel de organización, la gestión de calidad se ocupa de establecer un marco de proceso y estándares de organización que conducirán a software de mejor calidad.
- A nivel del proceso, la gestión de calidad implica la aplicación de procesos específicos de calidad y la verificación de que continúen dichos procesos planeados.
- A nivel del proyecto, la gestión de calidad se ocupa también de establecer un plan de calidad para un proyecto.

Aseguramiento de calidad

Es la definición de procesos y estándares que deben conducir a la obtención de productos de alta calidad y, en el proceso de fabricación, a la introducción de procesos de calidad.

El espíritu del aseguramiento de la calidad del SW es actuar como medida preventiva a diferencia del testing que llega tarde.

El equipo QA debe ser independiente del equipo de desarrollo para que pueda tener una perspectiva objetiva del software.

La planeación de calidad es el proceso de desarrollar un plan de calidad para un proyecto. El plan de calidad debe establecer las cualidades deseadas de software y describir cómo se valorarán. Por lo tanto, define lo que realmente significa software de “alta calidad” para un sistema particular.

Humphrey (1989), en su libro referente a la gestión del software, sugiere un bosquejo de estructura para un plan de calidad. Éste incluye:

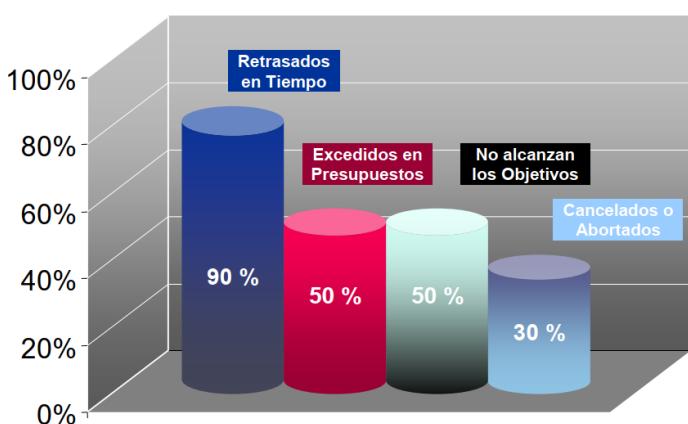
- Introducción del proyecto;
- Planes del producto: fechas de entrega críticas y las responsabilidades para el producto;
- Descripciones de procesos;

- Metas de calidad: Las metas y los planes de calidad para el producto, incluyendo una identificación y justificación de los atributos esenciales de calidad del producto;
- Riesgos y gestión de riesgos.

Aunque los estándares y procesos son importantes, **los administradores de calidad deben enfocarse también a desarrollar una “cultura de calidad”** en la que todo responsable del desarrollo del software se comprometa a lograr un alto nivel de calidad del producto. Deben exhortar a los equipos a asumir la responsabilidad de la calidad de su trabajo y desarrollar nuevos enfoques para el mejoramiento de la calidad.

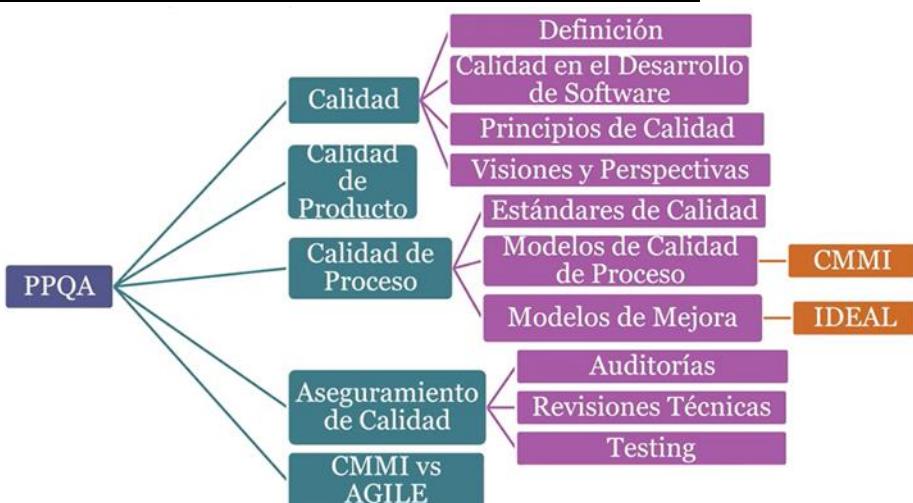
Problemas en la calidad

- Atrasos en las entregas -> Relacionado a calidad del proyecto.
- Requerimientos no claros -> Relacionado a calidad del producto.
- Software que no hace lo que debería -> Relacionado a calidad del producto.
- Costos excedidos -> Relacionado a calidad del proyecto.
- Trabajo fuera de hora -> Relacionado a calidad del proyecto.
- Fenómeno 90-90 (90% hecho 90% restante) -> Relacionado a calidad del proyecto.
- No se aplica SCM. -> Relacionado a calidad del producto.



Un software de calidad no solo debería cumplir con los requerimientos, sino que también debería poder entregarse a tiempo, no exceder costos, etc.

Aseguramiento de calidad de Proceso y de Producto



Calidad en el desarrollo de Software

Los costos excedidos, retrasos en la entrega, falta de cumplimiento de los compromisos, requerimientos no claros hacen que la percepción de nuestro cliente sea mala. El proyecto es el medio que permite alcanzar el producto. Si el medio no tiene calidad, difícilmente se pueda lograr un producto de calidad.

Para que un SW sea de calidad debe satisfacer:

- Las expectativas del cliente;
- Las expectativas del usuario;
- Las necesidades de la gerencia;
- Las necesidades del equipo de desarrollo y mantenimiento;
- Las expectativas de otros interesados.

Principios de Calidad

- La calidad NO se inyecta, debe estar embebida: la calidad es algo que se concibe desde el momento cero, desde requerimientos en adelante. Lo que se hace con el testing es controlarla.
- Es una responsabilidad de todos los involucrados en el proyecto.
- Las personas son la clave para lograrla: se hace mucho hincapié en la capacitación, para brindar herramientas y conocimientos a los involucrados. "Si usted cree que la capacitación es cara entonces pruebe con la ignorancia". Hacer software es una actividad humano-intensiva
- Se necesita sponsor a nivel gerencial, pero se puede empezar por uno.
- Se debe liderar con el ejemplo.
- No se puede controlar lo que no se mide, debemos usar métricas.
- Simplicidad, empezar por lo básico.
- Debe planificarse el aseguramiento de calidad.
- El aumento de las pruebas no aumenta la calidad. La calidad se obtiene y se inyecta a lo largo del proyecto, no al final.
- Debe ser razonable para mi negocio.
- Prevención mejor que corrección: prevenir es menos costoso, ya que corregir demanda un costo muy alto asociado al retrabajo.

En la actualidad, el trabajo con calidad es fundamental en la producción del software ya que:

- Es un aspecto competitivo, que permite sobrevivir en el mercado internacional. Las empresas certifican estándares de calidad (de proceso no de producto) para poder competir en el mismo.
- Retiene a los clientes e incrementa los beneficios. Siempre que hablamos de calidad, en el fondo nos referimos al cumplimiento de las expectativas del cliente en todos los sentidos.
- Determina un equilibrio del costo-efectividad. Trabajar con calidad reduce los costos y el retrabajo, ya que se asume que las fallas serán menores.

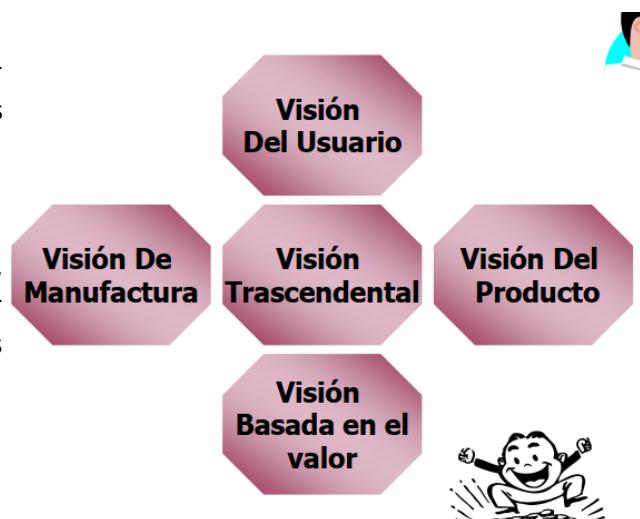
Las metodologías ágiles hablan constantemente de calidad de SW y se asume que los equipos ya están capacitados y orientados a procesos de calidad.

La disciplina de Gestión de la Administración de Configuración nos da el contexto de base para poder trabajar con calidad.

Calidad para Quién – Visiones de Calidad

A la calidad se la puede analizar desde distintas perspectivas o visiones:

- **Visión del usuario:** esto tiene que ver con las expectativas que tiene el usuario para con el producto, y si este las satisface. Esta es quizás la más complicada de establecer, porque está en la cabeza de los usuarios, lo cual es una razón más por la que el principio de comunicación constante es crucial, para ir validando en todo momento si estamos en el camino correcto. El agilismo trata de incluir la visión de calidad del usuario a través del rol de Product Owner, el cual es ocupado por una persona con capacidad de decisión del cliente.
- **Visión del producto:** esto se asocia al nivel de satisfacción de los requerimientos particulares de cada producto.
- **Visión del proceso (manufactura):** si el proceso de desarrollo utilizado es el correcto para el producto que se desea desarrollar, es decir, el proceso aporta valor al producto y no produce desperdicios.
- **Visión del valor:** encontrar un equilibrio en la relación costo-beneficio, para obtener siempre el mayor valor para el cliente posible y, obviamente generar ganancias con el desarrollo del software.
- **Visión Trascendental:** esta visión es un tanto utópica, ya que se asocia a objetivos difíciles de alcanzar. Por ejemplo 0 defectos en el producto, pero son aquellos objetivos que motivan a seguir en busca de la mejora continua.



La calidad es un concepto subjetivo que tiene que ver con, si para uno cumple con las necesidades y expectativas. Las expectativas son implícitas ya que son las cosas que quieras que abarque tu producto o servicio, pero no están dichas en ningún lado. Por ejemplo, que contenga una interfaz agradable el sistema, pero si no ocurren estas expectativas empiezan las disconformidades.

La calidad subjetiva de un sistema de software se basa principalmente en sus características no funcionales. Esto refleja la experiencia práctica del usuario: Si la funcionalidad del software no es lo que se esperaba, entonces los usuarios con frecuencia solos le darán la vuelta para realizar lo que necesitan y encontrarán otras formas de hacer lo que quieren. Sin embargo, si el software no es fiable o es muy lento, entonces es prácticamente imposible que los usuarios logren sus objetivos con el sistema.

Con demasiada frecuencia, la causa real de los problemas en la calidad del software no es una gestión deficiente, procesos inadecuados o capacitación de escasa calidad. Más bien, es el hecho de que las organizaciones deben competir para sobrevivir. Una empresa, puede estimar el esfuerzo requerido o prometer la entrega rápida de un

sistema en plazos de tiempo ajustados, para lograr un acuerdo con el cliente. En consecuencia, al no haber suficiente tiempo para el desarrollo, es probable que el software entregado tenga funcionalidades reducidas o niveles más bajos de fiabilidad o rendimiento, por lo tanto, esto conlleva a que la calidad del software se vea afectada, en ese intento de lograr cerrar un acuerdo de negocio con el cliente que necesita el software.

Calidad en el Software

La gente necesita definir un proceso para llevar a cabo el software, y generalmente en las organizaciones se basa en modelos para tomar de referencia.

Si quiero funcionar con mejora continua tengo modelos para mejorar esos procesos. También tenemos modelos de evaluación que ven el grado de adherencia del proceso al modelo que se tomó de referencia. Por ejemplo, si tomo la ISO 9001, si llamo a una auditoría de ISO esa auditoría debería decirte todas las diferencias que se tienen en ese proceso definido respecto de lo que deberías tener.

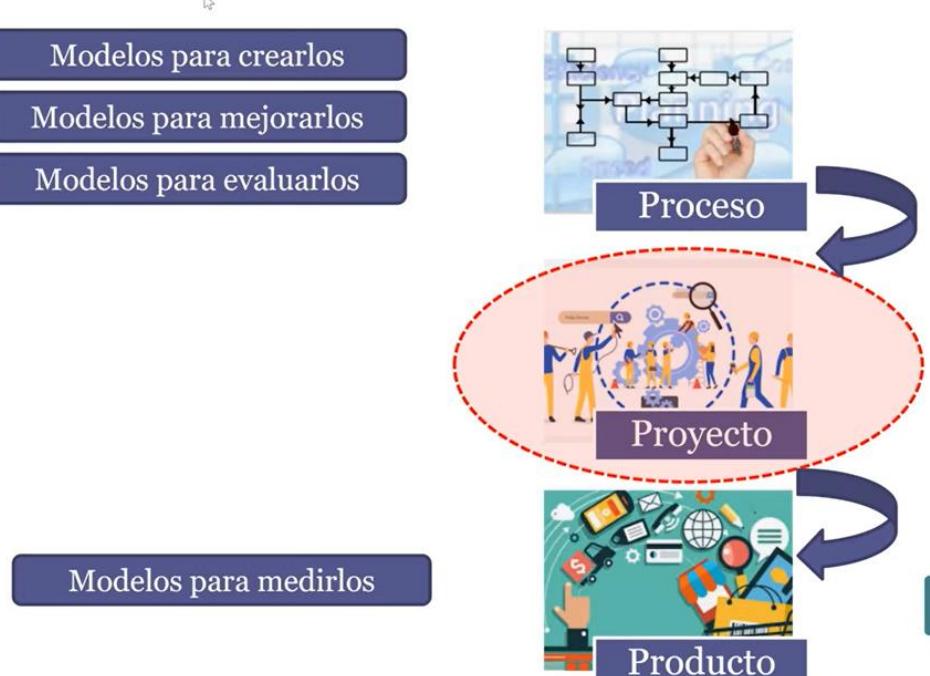
Los procesos se instancian en los proyectos. Los procesos son solo una indicación de cómo hacer las cosas, pero es el proyecto que al ejecutarse se insertan actividades para ir regulando si lo que estoy haciendo es lo que se había pensado.

Tenemos las revisiones técnicas que se hacen entre pares, no se somete a la persona a evaluación sino el producto. Se puede hacer sobre cualquier artefacto que queramos: requerimientos, etc. Luego tenemos las auditorías que son realizadas por personas externas (los empíricos están muy en contra porque creen que los equipos son capaces de reconocer y corregir por sí mismos, esto puede verse en la retrospectiva).

El producto que tengo que someter a evaluación es el que se va generando en el contexto del proyecto, por lo que tengo que insertar tareas para asegurar la calidad del producto. Ahí es donde

aparecen varias técnicas como revisiones técnicas (realizado entre pares, con el propósito de detectar tempranamente el defecto), auditorías de configuración funcional (la que valida una versión de la línea base), auditorías de configuración física (la que verifica una versión de la línea base), Testing (es para controlar la calidad cuando el producto ya está listo).

Los modelos de calidad dicen: hace tu proceso de manera que tenga calidad para vos (empresa), pero este proceso debe ser compatible con lo que yo te digo (modelo de referencia).



Modelos para medirlos

Calidad del producto

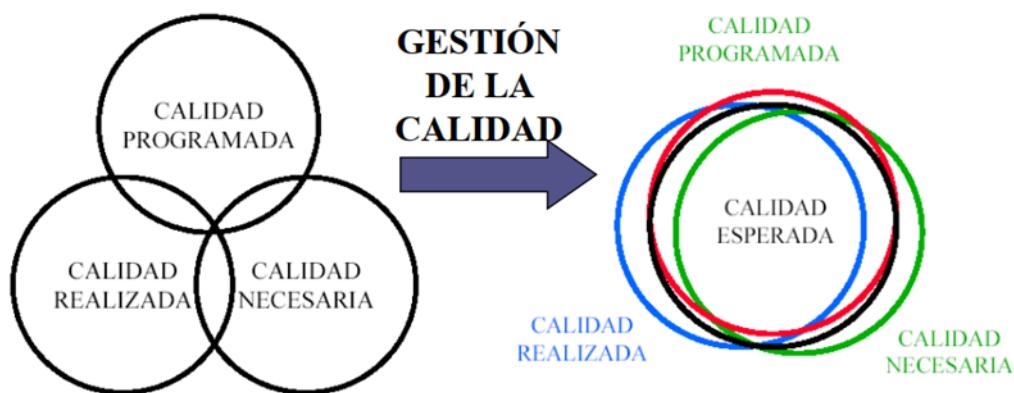
Definimos anteriormente que la calidad es el nivel de cumplimiento o adecuación a las necesidades (explícitas e implícitas) y para el caso del producto estas deberían estar explicitadas en los requerimientos. Es por esto que los

requerimientos son tan importantes, porque si no están o están mal definidos, no tenemos contra qué validar. Esto quiere decir que para que los requerimientos nos sirvan estos tienen que ser medibles (verificables) y objetivos (no ambiguos).

La gestión de calidad en productos son acciones sistemáticas de las organizaciones para lograr calidad, las cuales requieren equilibrio entre:

- Calidad programada: los alcances del producto planificados.
- Calidad realizada: lo que realmente se ha desarrollado del producto.
- Calidad necesaria: mínimas características que el producto debe tener, para que este satisfaga los requerimientos.

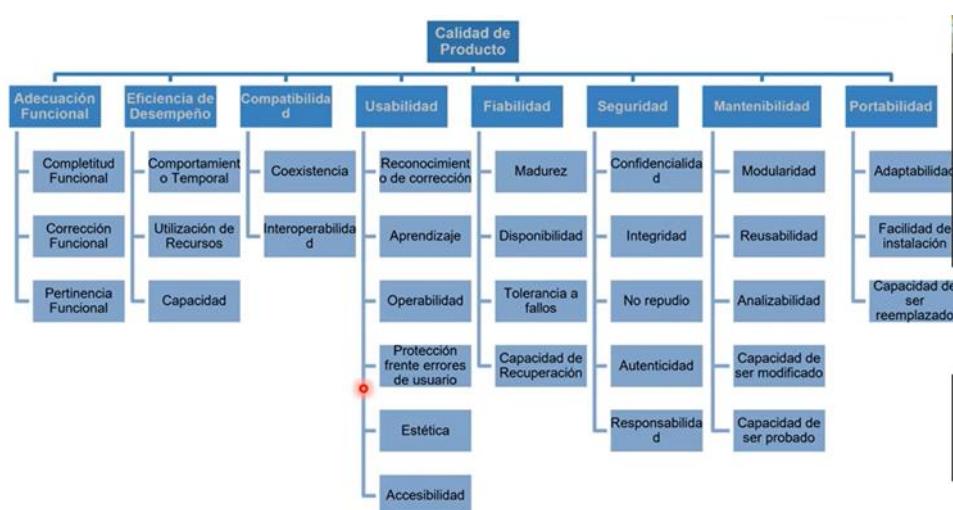
En el equilibrio obtenemos las expectativas de calidad que esperamos del producto y todo lo que esté por fuera de esta es desperdicio o insatisfacción.



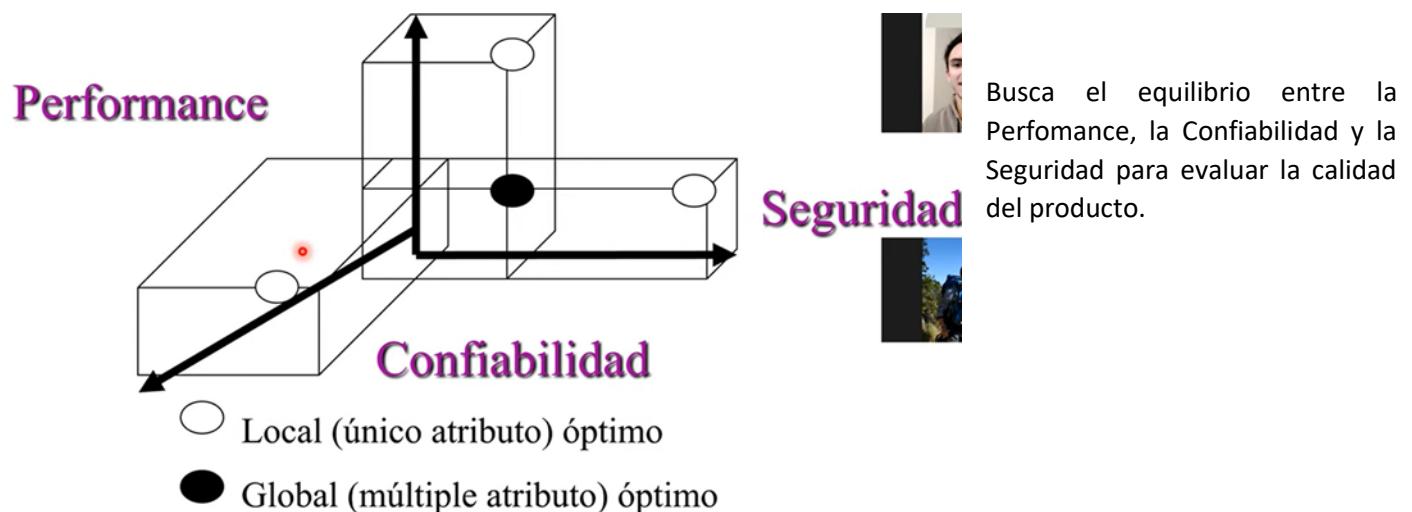
Modelos de calidad de Producto

Al variar los requerimientos para cada producto en particular, no existen modelos de calidad que acrediten para un determinado software qué nivel de calidad posee. Si existen modelos como el Modelo de Barbacci, Calidad de Software (MCCALL) o la ISO 25.010 que permite medir epifenómenos como los RNF del software, para certificar calidad en ciertos aspectos (no los vemos en esta materia).

ISO 25000 -RNF



Modelo de Barbacci / SEI



Calidad del SW McCall



Corrección - Fiabilidad - Usabilidad (facilidad de manejo) -
Integridad - Eficiencia

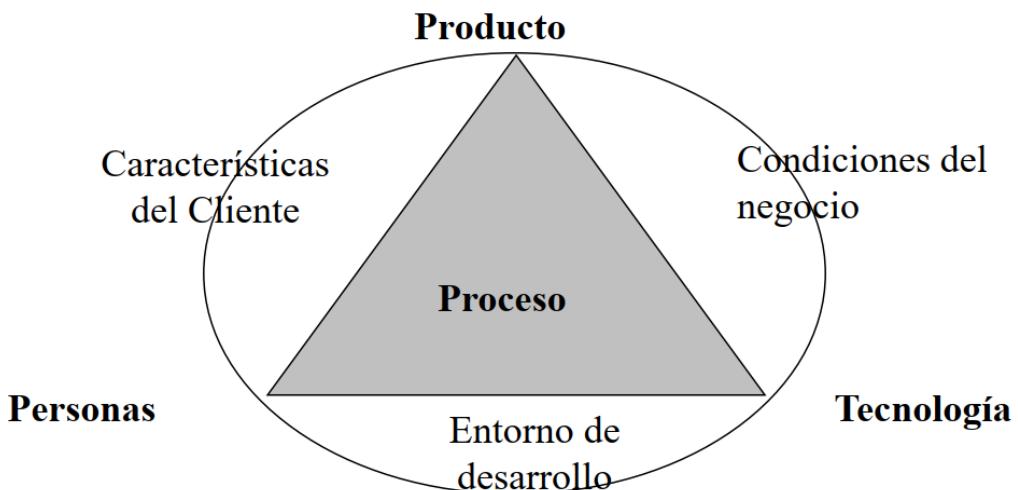
Busca el equilibrio entre la Perfomance, la Confiabilidad y la Seguridad para evaluar la calidad del producto.



Tiene tres grandes agrupadores unos relacionados a la capacidad de verificación del producto (revisión del producto), otras con el producto en uso (operación del producto) y otras con los factores que influyen que el producto pueda crecer en el tiempo (transición del producto).

Calidad del Proceso

- No existe en la realidad un proceso que sirva para todas las organizaciones y proyectos en general.
- La calidad de proceso nunca es un fin en sí mismo, lo que realmente nos interesa es la calidad de producto.
- Proceso con calidad → Producto con calidad
- Se insiste tanto en la calidad del proceso, debido a que es el único factor controlable para mejorar la calidad del software:



- El avance de las tecnologías que se utilizan para construirlo es ajeno a la organización y no es posible controlarlo.
- Las características y necesidades del cliente tampoco se pueden modificar.
- Las personas involucradas en el proceso de desarrollo son difíciles de controlar también.

Se aplica calidad en el producto y en el proceso. No se habla de aseguramiento de calidad en el proyecto, dado que esta se encuentra implícita por el hecho de que el proyecto implementa un proceso. Para garantizar la calidad se realizan auditorias.

La calidad del producto se realiza mediante actividades de revisiones técnicas y de auditorías. Estas son las herramientas principales para el seguimiento.

Para obtener calidad en el producto se debe definir un proceso que debe ser conocido por todos los involucrados en el equipo, donde además de tener tareas técnicas (requisitos, análisis, diseño, etc.) se incorporan disciplinas transversales como SCM, QA, Planificación de Proyectos y su Seguimiento.

Objetivos de QA:

- Realizar los controles apropiados del software y de su proceso de desarrollo.
- Asegurar el cumplimiento de los estándares y procedimientos para el software y el proceso.
- Asegurar que los defectos en el producto, proceso o estándares son informados a la gerencia para que puedan ser solucionados.

Estándares de Gestión de Calidad del Software:

Estándares de Producto

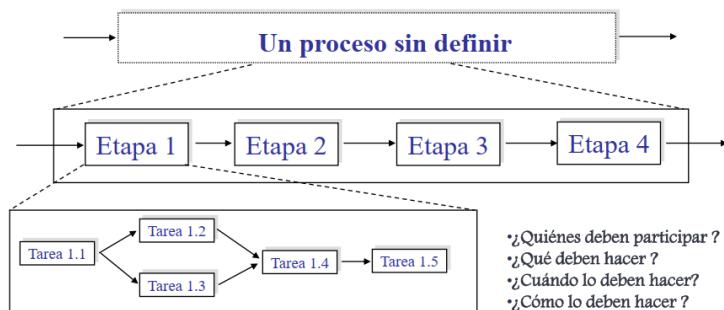
Definen las características que todos los componentes del producto deberían exhibir. Por ejemplos: estilos/buenas prácticas de programación, estándares de documentación.

Estándares de Proceso

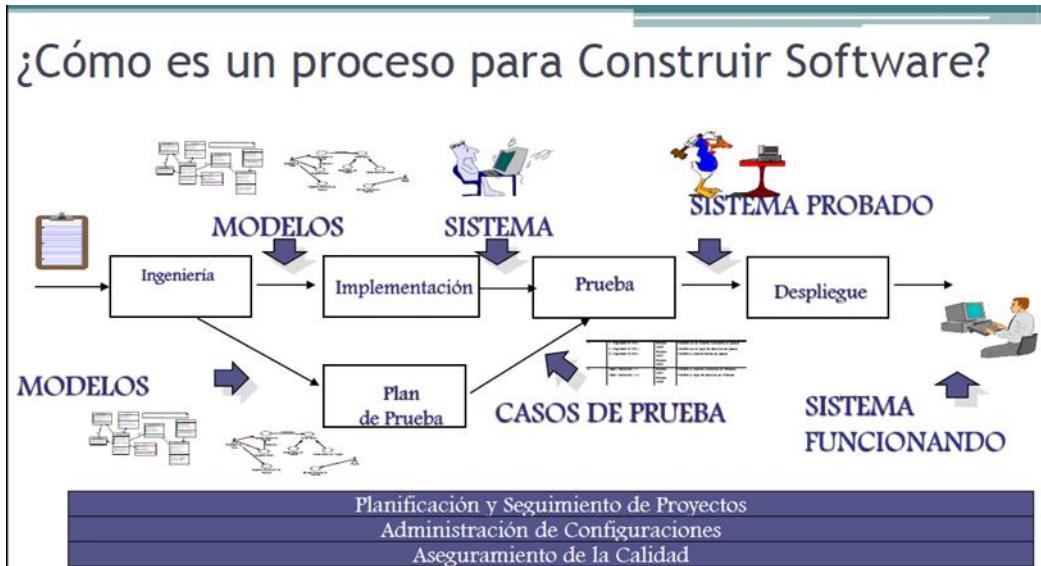
Establecen los procesos que deben seguirse durante el desarrollo, por ejemplo: incluyen definiciones de requerimientos, diseño, validación, etc. Definen como deben ser implementados los procesos de software.

Definición de Procesos

- Lo primero que se debe realizar en una organización para lograr tener un proceso de calidad, es **definirlo** de forma explícita y comunicarlo a todo el equipo, para que así todos tengan una base y el conocimiento de la forma de trabajar. Se deben definir aspectos como etapas, subetapas, roles, qué se debe hacer, en qué momentos se deben hacer, cómo lo deben hacer, etc.



- Dentro de esa definición, la cual plantea las etapas para construir la parte técnica (Ingeniería de Requerimientos, Análisis, Diseño, Implementación, Prueba y Despliegue), también se deben incorporar disciplinas transversales a ellas, como la Planificación y Seguimiento de Proyectos, SCM y QA, independientemente de la metodología adoptada (tradicional o empírica).
- El proceso definido y adoptado, debe aportar valor al producto (es posible utilizar modelos de mejora de proceso como Kanban) y utilizarlos en los distintos proyectos de forma ADAPTADA.
- La adaptación de los procesos se da en aspectos negociables del mismo. Por ejemplo, el realizar Testing no es algo negociable y que se pueda eliminar del mismo.



Procesos definidos

En los procesos definidos se considera que el proceso es el único factor controlable, por lo tanto, se asume que la mejora de la calidad continua se realiza sobre el proceso. Si el proceso cuenta con calidad, entonces el producto será de calidad.

Todos los modelos de calidad están basados en procesos definidos, por lo que asumen que la calidad del producto se obtiene si se tiene un proceso de calidad para construir el producto.

Procesos empíricos

Los procesos empíricos están basados en la experiencia, por lo que no consideran que la calidad en el proceso determine la calidad en el producto. Por otra parte, no están de acuerdo con las auditorías, ya que asumen que los equipos son autoorganizados. Sin embargo, la calidad en estos procesos se lleva a cabo mediante revisiones técnicas y la constante inspección y adaptación del trabajo.

Según los empíricos, siempre que las personas hagan lo que tengan que hacer el producto tendrá calidad.

Actividades relacionadas con el Aseguramiento de la Calidad del Software.

Administración de la calidad de Software

Busca asegurar que se alcancen los niveles requeridos de calidad para el producto de SW, definiendo estándares y proceso de calidad apropiados (que deben ser respetados por todos). El fin en sí de la Administración de Calidad de SW es generar una cultura de calidad y que esta sea vista como una responsabilidad de todos en el equipo.

La idea es insertar en las actividades acciones tendientes a detectar lo más temprano posible oportunidades de mejora sobre el producto y sobre el proceso. Hay que tener ciertas **consideraciones** respecto al Reporte del Grupo de Aseguramiento de Calidad (GAC):

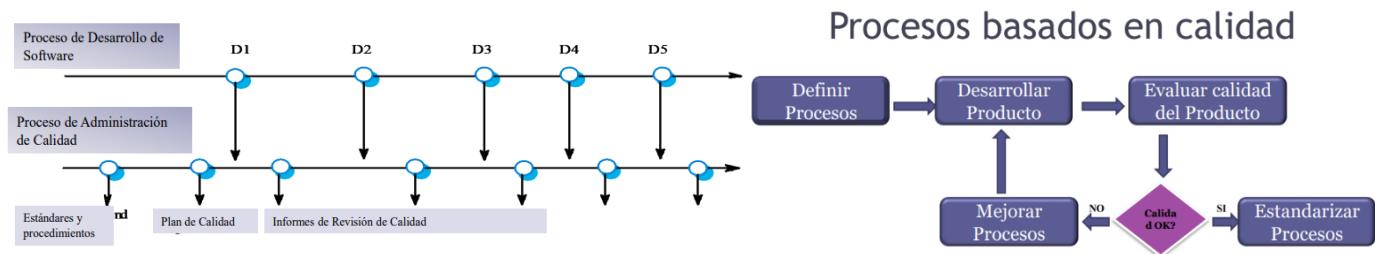
- No debería reportar la gente que hace calidad al mismo gerente que reporta los proyectos (porque le quita independencia y libertad).
- El grupo de aseguramiento de calidad debería depender del gerente general.
- Cuando sea posible, el grupo de aseguramiento de calidad debería reportar a alguien realmente interesado en el aseguramiento de calidad.

Entre las actividades que desempeña la administración de calidad, se encuentran:

- **Aseguramiento de calidad:** define estándares, procesos, procedimientos y modelos de calidad sobre los cuáles se van a realizar las comparaciones.
- **Planificación de Calidad:** se debe planificar la calidad, qué actividades se van a llevar a cabo, cuándo. Se selecciona los estándares aplicables para nuestro proyecto en particular y se los modifica si fuera necesario. Esta actividad abarca determinar los productos de SW que se desea que tengan calidad y determinar sus atributos de calidad más significativos.
- **Control de Calidad:** es la ejecución de lo planificado, y se revisa en qué situación está el proyecto. Asegura que los procedimientos y estándares son respetados por el equipo de desarrollo de SW. Existen dos enfoques para el control de calidad:
 - **Revisiones de calidad:** principal método de validación de la calidad de un proceso o un producto. El grupo examina parte de un proceso/producto + documentación, para encontrar potenciales problemas.
 - Tipos de revisiones de calidad:
 - Inspecciones para remoción de defectos (producto);
 - Revisiones para evaluación de progreso (producto y proceso);
 - Revisiones de Calidad (producto y estándares).
 - Evaluaciones de SW automáticas y mediciones.

Entre las funciones del Aseguramiento de Calidad de SW:

- Prácticas de aseguramiento de calidad.
- Evaluación de la planificación del proyecto de SW.
- Evaluación de requerimientos.
- Evaluación del proceso de diseño.
- Evaluación de las prácticas de programación
- Evaluación del proceso de integración y prueba del software
- Evaluación de los procesos de planificación y control de proyectos.
- Adaptación de los procedimientos de calidad para cada proyecto.



Calidad de procesos en la práctica: ¿cómo garantizamos calidad de proceso en la práctica? Cumpliendo con algunas de las siguientes tareas:

- Definir proceso estándares tales como:
 - Cómo deberían conducirse las revisiones;
 - Cómo debería realizarse la administración de configuración, etc.
- Monitorear el proceso de desarrollo para asegurar que los estándares sean respetados.
- Reportar en el proceso a la Administración De Proyectos y al responsable del SW.
- No use prácticas inapropiadas simplemente porque se han establecido los estándares.

Principales Modelos de Calidad existentes (CMMI – SPICE – ISO) y sus métodos de evaluación.

Lineamientos para la implementación de modelos de calidad en las organizaciones.

Modelos para la mejora de procesos

La mejora continua de procesos significa comprender los procesos existentes y cambiarlos para incrementar la calidad del producto o reducir los costos y el tiempo de desarrollo. Los procesos definidos están de acuerdo con esta definición, ya que consideran que la calidad del producto final depende de la calidad del proceso.

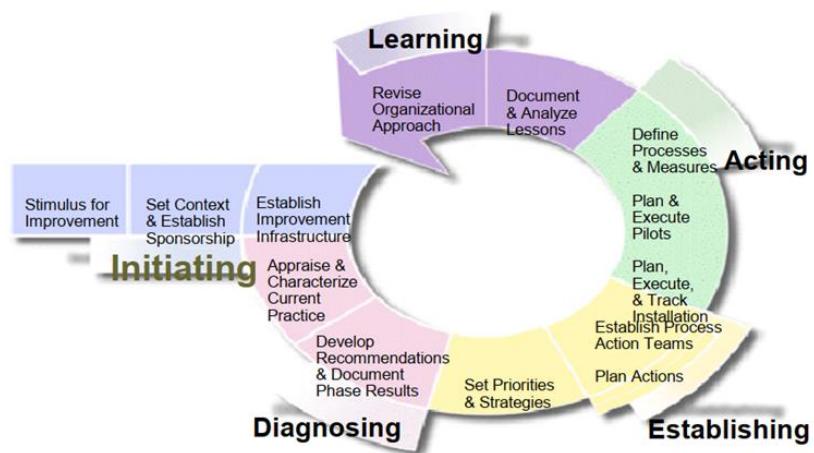
Los modelos NO te dicen cómo hacer las cosas. Son modelos descriptivos

El propósito de los modelos de mejora es analizar el proceso que tiene la organización y armar un proyecto cuyo resultado, en lugar de ser un producto, es un proceso mejorado que se vuelca de nuevo a la organización con la idea de que se produzca un producto mejor.

Modelo IDEAL (Initiating, Diagnosis, Establishing, Acting, Learning)

Nos da el contexto para crear un proyecto cuyo resultado va a ser un proceso definido. Es un modelo cíclico que mejora el proceso existente en una organización.

En el inicio empieza buscando un sponsor (apoyo en la organización (económico)). Esto es importante porque una mejora de proceso nunca es crítica, siempre hay algo más importante, entonces si no tengo un aval para ejecutar este tipo de proyectos, suelen no terminar exitosamente. Debemos entonces analizar dónde estamos y dónde queremos ir (se le llama análisis de brecha).



Se trata de un modelo de mejora de procesos que sirve como guía para inicializar, planificar e implementar acciones de mejora. Su nombre se debe a las 5 fases que lo componen:

- **Inicialización:** se reconocen las necesidades de cambio en la organización, razones para iniciar, determinar metas buscadas al proponer un cambio en el proceso. Se requiere que la organización apoye esta decisión de mejora (sponsoreo).
- **Diagnóstico:** Establecer la madurez actual de la organización y los riesgos asociados al proceso de mejora (revisar estado actual y futuro de la org.).
- **Establecimiento:** Durante la fase se elabora un plan detallado con acciones específicas, entregables y responsabilidades para el programa de mejora basado en los resultados del diagnóstico y en los objetivos que se quieren alcanzar. Para elaborar el plan se parte de definir las prioridades para el esfuerzo de mejora.
- **Ejecutar/Acción:** Efectuar los cambios y reunir información para aprender de la mejora. Se implementan las acciones planeadas en un proyecto piloto (no en todos los procesos de la organización, ya que es una prueba). Si la solución es satisfactoria para la org, se implanta en la empresa.
- **Aprendizaje:** busca garantizar que el próximo ciclo sea más efectivo. Durante la misma se revisa toda la información recolectada en los pasos anteriores y se evalúan los logros y objetivos alcanzados para lograr implementar el cambio de manera más efectiva y eficiente en el futuro. Extrapolar la mejora al resto de proyectos en caso de que sea exitosa la ejecución o realizar correcciones en caso de que fracase el proyecto piloto.

Modelo SPICE (Software Process Improvement Capability Evaluation)

Es un modelo creado para la mejora de procesos software (ISO 15504). Es una adaptación para la evaluación de procesos de desarrollo software por niveles de madurez, dividido en 6 niveles.

Es un modelo dual, teórico. Tiene 2 partes:

- **Modelo de Calidad**
- **SPICE + IDEAL:** son ambos modelos de mejora de proceso. Estos modelos de mejora se usan para crear Proyectos de Mejora para una organización. El resultado de este proyecto va a ser un proceso definido que se usará para hacer Proyectos.

Este modelo establece conjuntos predefinidos de procesos con objeto de definir un camino de mejora para una organización. En concreto, establece 6 niveles de madurez para clasificar a las organizaciones.

Nivel	Estado
Nivel 0 - Organización inmadura	La organización no tiene una implementación efectiva de los procesos
Nivel 1 - Organización básica	La organización implementa y alcanza los objetivos de los procesos
Nivel 2 - Organización gestionada	La organización gestiona los procesos y los productos de trabajo se establecen, controlan y mantienen
Nivel 3 - Organización establecida	La organización utiliza procesos adaptados basados en estándares
Nivel 4 - Organización predecible	La organización gestiona cuantitativamente los procesos
Nivel 5 - Organización optimizando	La organización mejora continuamente los procesos para cumplir los objetivos de negocio

Modelo de Calidad para evaluar procesos

Plantean una definición teórica (lineamientos) del proceso que se utilizará en una organización (con diferentes niveles de detalle según lo que se necesite), para luego compararlo con la ejecución del proyecto donde se implementa (instancia) ese proceso de desarrollo. Esto permite medir qué tanto se está cumpliendo en el proyecto con lo que plantea la definición teórica del proceso de desarrollo (definición de roles, asignación de responsabilidades, actividades a realizar, etc.), a través de las auditorías de proyecto.

¡No es lo mismo que un estándar de proceso! El objetivo de estos modelos de calidad es acreditar que una organización tiene cierto nivel de madurez en su proceso de desarrollo adoptado, o bien que cierta área de esa organización tiene determinado nivel de madurez.

Existen un montón de modelos de calidad. Los más utilizados en Argentina son el CMMI, CMMI 2.0, etc.

Capability Maturity Model Integration – CMMI

Es un modelo de calidad para la mejora y evaluación de procesos para el desarrollo, mantenimiento y operación de sistemas de software, que constituye un conjunto de buenas prácticas que son publicadas en modelos.

Es un modelo de referencia, son descriptivos. Te dice que tienes que alcanzar determinado objetivo, pero vos definís qué práctica vas a implementar para lograrlo.

Este modelo empírico recopiló buenas prácticas a lo largo de la historia de diferentes organizaciones, tomando diferentes prácticas de aquellas que han logrado desarrollos exitosos.

El objetivo es que pueda ser usado para guiar la mejora de procesos en un proyecto, división o una organización completa.

La acreditación se logra evaluando de manera objetiva (comparación con el modelo de evaluación SCAMPI) y subjetiva (Experiencia y pensamiento del evaluador) el proceso, y el uso de este proceso instanciado en proyectos.

Constelaciones

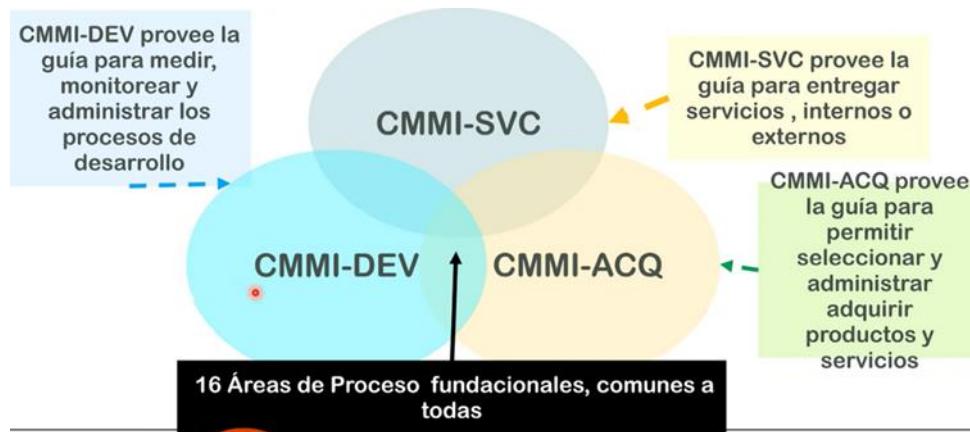
Existen 3 constelaciones o modelos de negocio que cubren los modelos de CMMI:

- **Desarrollo (CMMI-DEV):** provee la guía para medir, monitorear y administrar los procesos de desarrollos de software. Tiene dos representaciones, por etapas y continua.
- **Adquisición (CMMI-ACQ):** provee la guía para permitir seleccionar, administrar y adquirir productos y servicios a otra empresa que posee su propia forma de trabajo, delegando el control de ese proyecto para obtener el

producto o servicio final. Es decir, cuando la empresa no hace cosas, sino que contrata gente que haga las cosas/trabajos por ellos (no es lo mismo que subcontratación de personal).

- **Servicios (CMMI-SVC):** provee la guía para entregar servicios, externos o internos.

Estas constelaciones poseen los modelos, capacitaciones y toda la información que es necesaria para que las empresas puedan acreditar distintos niveles de madurez.



Áreas de proceso

- Un área de proceso es un conjunto de prácticas relacionadas en una zona que, cuando se implementan en conjunto, satisfacen un conjunto de objetivos considerados importantes para hacer mejoras significativas en el mismo proceso.
- Existen en total 22 áreas de procesos en todos los niveles de madurez, las cuales son iguales en ambas representaciones.
- 16 de esas 22 áreas de procesos son comunes a todas las constelaciones CMMI.
- Nivel 1: no existen áreas de proceso, ya que se considera que la organización está en estado de inmadurez.
- Nivel 2: son 7 en total, de las cuales la última es opcional (depende si la organización realiza o no actividades relacionadas)
 - **REQM** → Requirement Management *Gestión de Reqs*
 - **PP** → Project Planning *Gestión de proyecto*
 - **PMC** → Project Monitor and Control *Gestión de proyecto*
 - **MA** → Measure and Analytics *Métricas de proyecto*
 - **PPQA** → Product and Process Quality Assurance *Disciplina transversal*
 - **SCM** → Software Configuration Management *Disciplina transversal*
 - **SAM** → Supplier Agreement Management

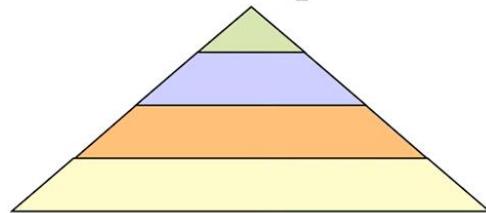
Nivel	Categoría			
	Administración de Proyectos	Soporte	Administración de Procesos	Ingeniería
5 Optimizado		* Análisis y Causal y Resolución (CAR)	* Administración de Performance Organizacional (OID)	
4 Cuantitativamente Administrado	* Administración Cuantitativa del Proyecto (QPM)		* Performance del Proceso Organizacional (OPP)	
3 Definido	* Administración de Riesgos (RSKM) * Administración Integrada de Proyectos (IPM)	* Análisis y Resolución de Decisión (DAR)	* Definición del Proceso Organizacional (OPD) * Foco en el Proceso Organizacional (OPF) * Capacitación Organizacional (OT)	* Desarrollo de Requerimientos (RD) * Solución Técnica (TD) * Integración de Producto (PI) * Verificación (VER) * Validación (VAL)
2 Administrado	* Administración de Requerimientos (REQM) * Planificación de Proyectos (PP) * Monitoreo y Control de Proyectos (PMC) * Administración de Acuerdo con el Proveedor (SAM)	* Aseguramiento de calidad de Proceso y de Producto (PPQA) * Administración de Configuración (CM) * Medición y Análisis (MA)		
1 Inicial	Procesos sin definir o improvisados			

Representaciones CMMI-DEV

- **Por etapas:** CMM (el antecesor a CMMI), se basaba mucho en por etapas. Identificaba a las organizaciones y las dividía en maduras (eran las del nivel del 2 al 5) e inmaduras (nivel 1), mientras más maduras más capacidad de lograr sus objetivos, bajando sus riesgos. La ventaja de esta es que evaluaba la organización (el área que se quería trabajar). Esto facilita la comparación entre organizaciones. (madurez organizacional).

Es necesario cumplir con los lineamientos definidos para cada área de proceso de los niveles inferiores, y de las áreas de proceso definidas en el nivel que se está acreditando. Es decir, la acreditación es acumulativa para los niveles inferiores.

Por Etapas



Niveles que se acreditan en la representación por etapas:

- **Inicial:** Las organizaciones en este nivel no disponen de un ambiente estable para el desarrollo y mantenimiento de software. Aunque se utilicen técnicas correctas de ingeniería, los esfuerzos se ven minados por falta de planificación. El éxito de los proyectos se basa la mayoría de las veces en el esfuerzo personal, aunque a menudo se producen fracasos y casi siempre retrasos y sobrecostes. El resultado de los proyectos es impredecible. Nivel de las organizaciones inmaduras.
- **Administrado:** la organización ha logrado todos los objetivos genéricos y específicos del nivel de madurez 2, es decir, los proyectos de la organización han asegurado que los requisitos son gestionados y de que los procesos se planifican, se realizan, se miden y controlado. También, hay un razonable seguimiento de la calidad.
- **Definido:** la organización ha alcanzado todos los objetivos específicos y de las áreas de proceso asignadas a los niveles de madurez 2 y 3. En este nivel los procesos están bien caracterizados y entendidos, y se describen en las normas, procedimientos, herramientas y métodos. Aquí los procesos se describen con más detalle y más rigurosidad que en el nivel de madurez 2. También se realiza formación del personal, técnicas de ingeniería más detalladas y un nivel más avanzado de métricas en los procesos. Se implementan técnicas de revisión de a pares.

- **Cuantitativamente administrado:** Se caracteriza porque las organizaciones disponen de un conjunto de métricas significativas de calidad y productividad, que se usan de modo sistemático para la toma de decisiones y la gestión de riesgos. El software resultante es de alta calidad. Las medidas detalladas del rendimiento de los procesos son recogidas y analizadas estadísticamente.
- **Optimizado:** este nivel se centra en mejora continua del rendimiento de los procesos a través de los aumentos y mejoras tecnológicas innovadoras.
Los objetivos cuantitativos de mejora de procesos para la organización se establecen y se revisan de forma continua a fin de reflejar los cambios objetivos de negocio, y se utilizan como criterios para la administración de la mejora de procesos. El alcanzar estas áreas se detecta mediante la satisfacción o insatisfacción de varias metas claras y cuantificables.

Para ser de un nivel, se debe cumplir con los requisitos de ese nivel y con los de los anteriores.

Nosotros hacemos foco en el nivel 2, las áreas del proceso son:



- Administración de Requerimientos.
- Planeamiento de Proyectos.
- Monitoreo y Control de Proyectos.
- **Administración de Acuerdo con el Proveedor.**
- Medición y Análisis
- Aseguramiento de Calidad del Proceso y del Producto.
- Administración de Configuración.

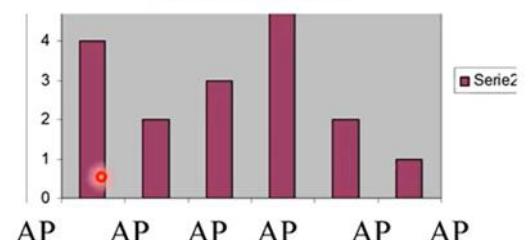
Administración de acuerdo con el proveedor no es obligatoria. Es necesaria si se contrata a una empresa externa como proveedora, ya que se debe comprobar que cumpla con el nivel de calidad que se tiene internamente. Hay 2 tipos de proveedores:

- Proveedor Tipo A: aquellos que cumplen con el mismo nivel de calidad.
- Proveedor Tipo B: aquellos que no cumplen con el mismo nivel de calidad.

Si alcanzo nivel 2 significa que la organización tiene la madurez para gestionar sus proyectos y que el producto que se producirá será algo esperable.

- **Continua:** El objetivo del uso de esta representación, es acreditar la capacidad de la organización ante cada una de las áreas de proceso de forma individual. En lugar de medir la madurez de toda la organización, mido la capacidad de un proceso en particular. Procesos de nivel cero son los que no se ejecutan (Por ejemplo: si le pregunto a una empresa si hace gestión de configuración de software y me dice que no lo hace, entonces, es nivel 0 en esa área). Apunta a mejorar áreas de proceso que uno elige. Se centra en la mejora de un proceso o un conjunto de ellos relacionados a un área de proceso que una organización desea mejorar, una organización puede obtener la certificación para un área de proceso en cierto nivel de capacidad.

Continua



Roles – Grupos

Los roles se adaptan a las necesidades de la organización. Cuando hablamos de grupo no nos referimos a conjunto de personas, sino de alguien que asuma ese rol.

Lo importante es que exista alguien responsable de cubrir las actividades de cada uno de los roles o grupos.

Relación de CMMI con Ágil

CMMI cara a cara con Ágil

- “Nivel 1”
 - Identificar el alcance del trabajo
 - Realizar el trabajo

- “Nivel 2”
 - Política Organizacional para planear y ejecutar
 - Requerimientos, objetivos o planes
 - Recursos adecuados
 - Asignar responsabilidad y autoridad
 - Capacitar a las personas
 - Administración de Configuración para productos de trabajo elegidos
 - Identificar y participar involucrados
 - Monitorear y controlar el plan y tomar acciones correctivas si es necesario
 - **Objetivamente monitorear adherencia a los procesos y QA de productos y/o servicios**
 - Revisar y resolver aspectos con el nivel de administración más alto

Referencias:
Verde : Da soporte,
Negro: Neutral,
Rojo: Desigual



No coincide con el monitoreo objetivo pues hace referencia a las auditorías realizadas por agentes externos al equipo. Para poder usar CMMI siendo ágil, ambas deben ceder un poco: Ágil plantea de gestión ágil de requerimientos y no pide que haya un control de qué requerimientos tiene el producto en cada momento de tiempo, pero CMMI sí. De alguna forma se debe tener trazabilidad de los requerimientos en US y demás para CMMI. Así como CMMI acepta no tener el registro de las Daily por ejemplo. Ágil debe ceder Auditorías.

- “Nivel 3”
 - Mantener un proceso definido
 - **Medir la performance del proceso**

- “Nivel 4”
 - Establecer y mantener objetivos cuantitativos para el proceso
 - Estabilizar la performance para uno o más subprocesos para determinar su habilidad para alcanzar logros

- “Nivel 5”
 - Asegurar mejora continua para dar soporte a los objetivos
 - Identificar y corregir causa raíz de los defectos

Negro:
Rojo:



¿Por qué no coincide a medida que subimos de nivel? CMMI dice que tienes que definir un proceso y que después lo tienes que cumplir (lo que se verifica con las auditorías). En cambio, ágil en ningún momento evalúa que hayas seguido el proceso que definiste. CMMI a partir de nivel 3 o 4 plantea métricas/estadísticas de los proyectos y productos, y en base a la comparación de esas medidas se arma una medida del proceso. Ágil plantea que la experiencia no es extrapolable a otros proyectos.

Valores esenciales de cada metodología

CMMI	MÉTODOS ÁGILES
→ Medir y mejorar el proceso [Mejores Procesos ↓ Mejor Producto]	→ Respuestas a clientes → Mínima sobrecarga → Refinamiento de Requerimientos <ul style="list-style-type: none">- Metáforas- Casos de negocio
→ Características de las personas <ul style="list-style-type: none">- Disciplinados- Siguen reglas- Aversión al riesgo	→ Características de las personas <ul style="list-style-type: none">- Comfortable- Creative- Risk Takers
→ Comunicación <ul style="list-style-type: none">- Organizacional- Macro	→ Comunicación <ul style="list-style-type: none">- Person to Person- Micro
→ Gestión de Conocimiento <ul style="list-style-type: none">- Activos de proceso	→ Gestión de Conocimiento <ul style="list-style-type: none">- Personas

Características CMMI vs. Metodologías ágiles

CMMI	MÉTODOS ÁGILES
→ Mejora Organizacionalmente	→ Mejora en el Proyecto
-Uniformidad -Nivelación	- Tradición Oral - Innovación
→ Capacidad/Madurez	→ Capacidad/Madurez
- Éxito por Predictibilidad	- Éxito por darse cuenta de oportunidades
→ Cuerpo de Conocimiento	→ Cuerpo de Conocimiento
- Cruzando dimensiones - Estandarizado	- Personal - Evolucionando - Temporal
→ Reglas de Atajo	→ Reglas de Atajo
- Desalentadas	- Alentadas

CMMI	MÉTODOS ÁGILES
→ Comités	→ Individuos
→ Confianza del Cliente	→ Confianza del Cliente
- En la Infraestructura del Proceso	- Sw funcionando, Participantes
→ Cargado al frente	→ Conducido por Pruebas
- Mover a la derecha	- Mover a la izquierda
→ Alcance de la vista [Involucrado, Producto]	→ Alcance de la vista [Involucrado, Producto]
- Amplio - Inclusivo - Organizacional	- Pequeño - Focalizado
→ Nivel de Discusión	→ Nivel de Discusión
- Palabras - Definiciones - Duradero - Exhaustivo	- Trabajo en mano

En cuanto al enfoque

CMMI	MÉTODOS ÁGILES
→ Descriptivo	→ Prescriptivo
→ Cuantitativo	→ Cualitativo
- Número científicos y duros	- Conocimiento tácito
→ Universalidad	→ Situacional
→ Actividades	→ Producto
→ Estratégico	→ Táctico
→ “¿Cómo lo llamaremos?”	→ “Sólo hazlo!”
→ Gestión de Riesgos	→ Gestión de Riesgos
- Proactiva	- Reactiva

CMMI	MÉTODOS ÁGILES
→ Foco de Negocio	→ Foco de Negocio
- Interna - Reglas	- Externo - Innovación
→ Predictibilidad	→ Performance
→ Estabilidad	→ Velocidad

Similitudes entre ambas

- Meta: organizaciones de alto desempeño;
- Ambas planean;
- Ambas son Consultant Money Makers (CMMs);
- Ninguna es completa;
- Ninguno es aplicable a cualquier proyecto;
- No nuevas ideas;
- Ambas tienen reglas (reglas == requerimientos del proceso):
 - Incumplir tiene serias repercusiones;
 - 'SEPG' (Grupo de proceso de ingeniería de software) & 'Política de Proceso'.

Diferentes tipos de Auditorias: Auditorías de Proyecto y Auditorías al Grupo de Calidad.

Auditorías de calidad de Software

Es una actividad incluida dentro de las disciplinas de soporte, la cual implica una evaluación **independiente** (realizada por un grupo de trabajo que no tienen nada que ver con el equipo del proyecto) de productos o procesos de software que permiten asegurar el cumplimiento con estándares, lineamientos, especificaciones y procedimientos basada en un criterio objetivo incluyendo documentación.

Las auditorías implican esfuerzo y costo para los proyectos, sin embargo, sus beneficios son superiores.

Son un instrumento para el Aseguramiento de Calidad en el Software.

Beneficios de las auditorías

- Se obtienen opiniones objetivas e independientes;
- Permiten identificar áreas de insatisfacción potenciales del cliente;
- Permite asegurar que se están cumpliendo con las expectativas;
- Permite dar visibilidad sobre los procesos de trabajo;
 - Permite visualizar los puntos fuertes de una organización;

- Permite identificar oportunidades de mejora;
- Da visibilidad a la gerencia sobre los procesos de trabajo;
- Determinan que se implementen de manera efectiva: el proceso de desarrollo organizacional y del proyecto, y las actividades de soporte.

Tipos de auditorías

Auditoría de proyecto

Responsable de ver que el proyecto se haya ejecutado con el proceso que se definió. Esto bajo la premisa de que en un proyecto se debe realizar lo que define el proceso, obviamente adaptado a lo que sirve en ese contexto del proyecto particular (teniendo en cuenta que, para obtener una acreditación de calidad, se tiene que ajustar hasta cierto punto al modelo teórico de referencia).

Lo que se hace es tomar evidencias de lo que se está haciendo y contrastarlo con lo documentado en las ERS, arquitectura, diseño, etc.

Acá puede haber diferencias respecto a metodologías tradicionales o empíricas. En SCRUM, por ejemplo, el mismo equipo realiza estas auditorías en la ceremonia de Sprint Retrospective. En cambio, en metodologías tradicionales, el auditor debe ser externo al proyecto y a veces a la empresa.

En resumen: se evalúa el proceso adoptado (definición teórica), pero esa adopción del proceso se da en un proyecto, por ende, la auditoría se realiza sobre el proyecto.

Estas auditorías se llevan a cabo de acuerdo a lo establecido en las PACS (Plan de Aseguramiento de Calidad de Software).

El objetivo de esta auditoría es verificar objetivamente la consistencia del producto a medida que evoluciona a lo largo del proceso de desarrollo.

Auditorías de Configuración Funcional

Valida que el producto cumpla con los requerimientos. La auditoría funcional compara el software que se ha construido (incluyendo sus formas ejecutables y su documentación disponible) con los requerimientos de software especificados en la ERS.

El propósito de la auditoría funcional es asegurar que el código implementa sólo y completamente los requerimientos y las capacidades funcionales descriptos en la ERS.

El responsable de QA deberá validar si la matriz de rastreabilidad está actualizada.

Auditoría de configuración física

Valida que el ítem de configuración cumpla con la documentación técnica que lo describe (esto permiten la trazabilidad y la satisfacción de los requerimientos).

La auditoría física compara el código con la documentación de soporte.

Su propósito es asegurar que la documentación que se entregará es consistente y describe correctamente al código desarrollado.

El PACS debería indicar la persona responsable de realizar la auditoría física.

El software podrá entregarse sólo cuando se hayan arreglado las desviaciones encontradas.

Auditorías externas de Aseguramiento de Calidad (No sé si son un tipo más)

Son auditorías realizadas por un personal externo a la organización, a quienes se encargan de realizar las auditorías de calidad. Esto permite evaluar a los miembros de QA, para determinar si sus evaluaciones dentro de la organización se están realizando de forma correcta.

Roles en una auditoría

- **Auditado:** Participa en la auditoría, y es quien propone la fecha de la auditoría, entrega evidencia, contesta las dudas del auditor, propone planes de acción para las desviaciones encontradas, contesta el reporte de auditoría, propone el plan de acción para las deficiencias citadas en el reporte. Generalmente es el Líder de Proyecto, pero puede ser otro.
- **Auditor:** puede ser una persona o dos y deben ser de afuera del proyecto que se está auditando. Puede ser el grupo de aseguramiento de calidad, que es el grupo que le da soporte y auditorías de estos tipos.
 - Acuerda la fecha de la auditoría.
 - Comunica el alcance de esta.
 - Recolecta y analiza la evidencia objetiva que es relevante y suficiente para tomar conclusiones acerca del proyecto auditado.
 - Realiza la auditoría.
 - Prepara el reporte.
 - realiza el seguimiento de los planes de acción acordados con el auditado.
- **Gerente de SQA (Software Quality Assurance):** Es quien prepara el plan de auditoría, calcula su costo, asigna recursos, resuelve las no-conformidades entre auditor y auditado. (orientado a la gestión de la auditoría)

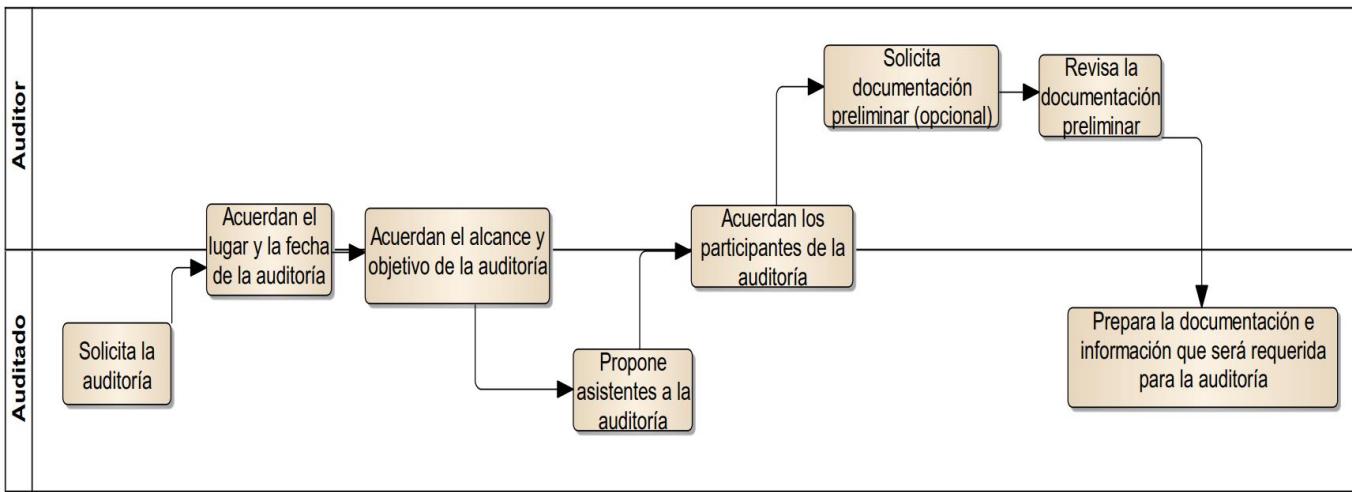
Proceso de Auditorías: Responsabilidades. Preparación y ejecución. Reporte y seguimiento.

Etapas de una auditoría

1. Preparación y planificación

No son sorpresa. El auditado solicita la auditoria y junto con el auditor definen la fecha, el alcance y objetivo, acordando los participantes de esta.

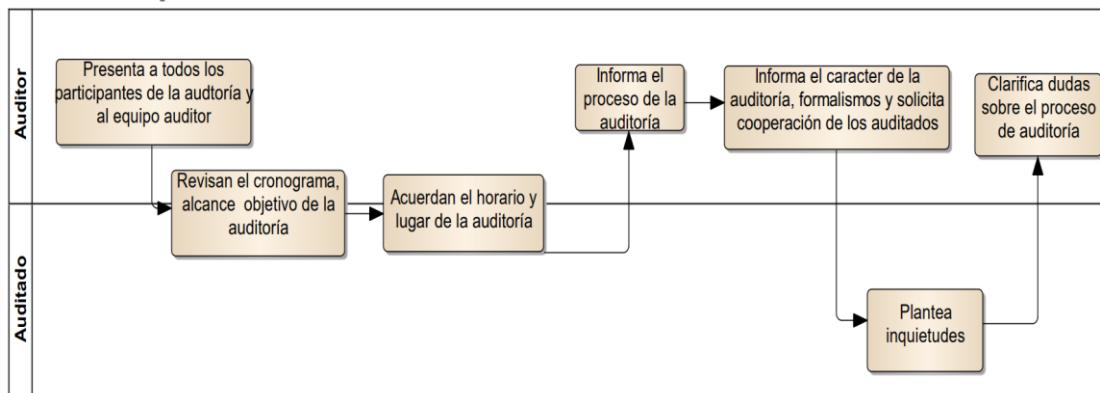
Se deben definir métricas de auditoría, como por ejemplo: esfuerzo por auditoría, cantidad de desviaciones, duración de la auditoría, cantidad de desviaciones clasificadas por auditor de CMMI, etc.



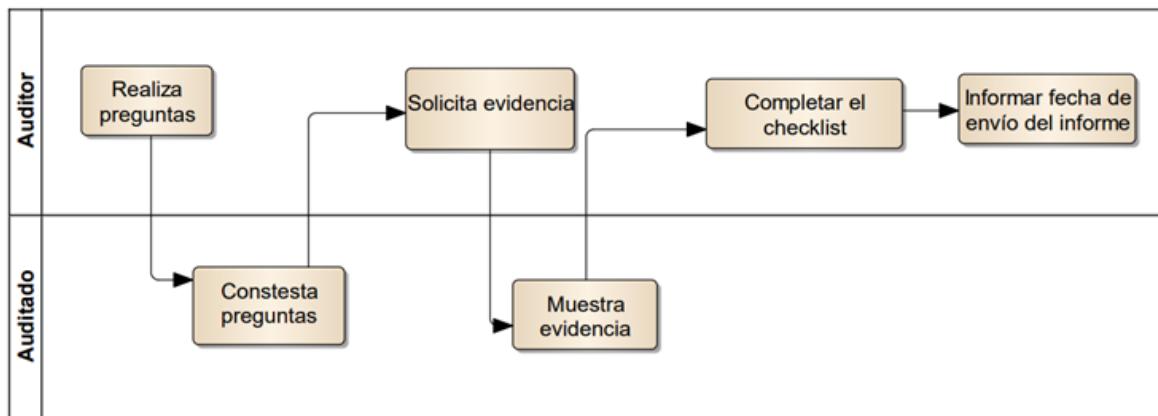
2. Ejecución

El auditor escucha lo que la gente dice que hace y luego ve la documentación, pide evidencia (lo que debería estar haciendo).

- Reunión de apertura: se informa cómo será el proceso, indicando el lugar y la hora.



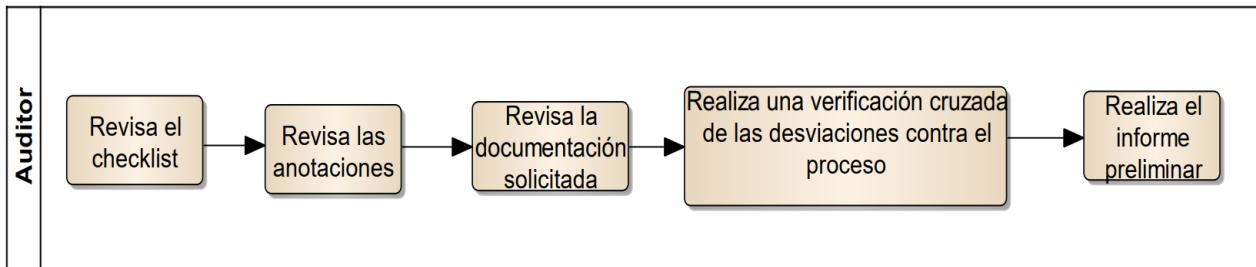
- Ejecución: se responden las preguntas, se muestra la evidencia y se completa el checklist. El auditor escucha lo que la gente dice que hace y luego ve la documentación (lo que debería estar haciendo).



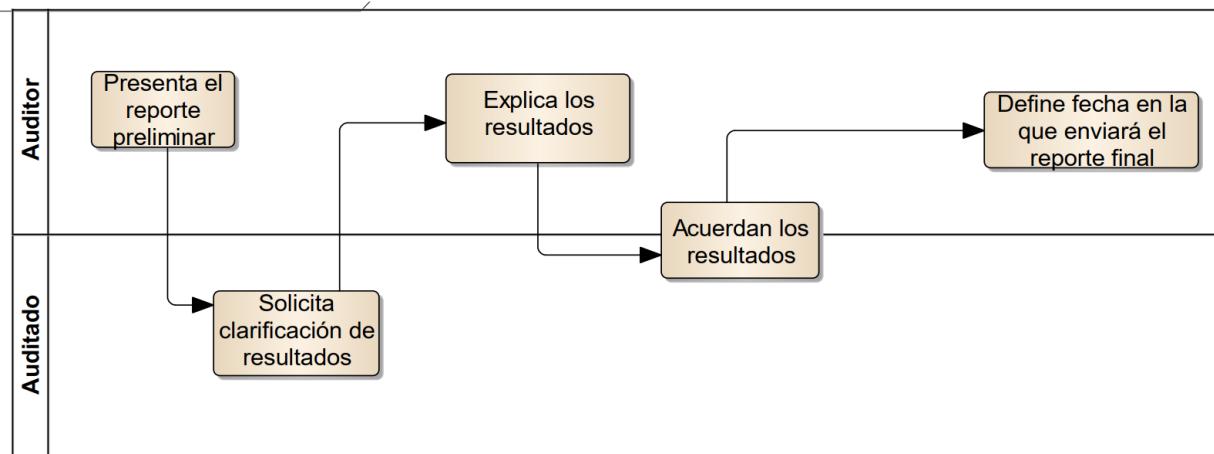
3. Análisis y reporte de resultados

El auditor realiza el reporte de resultados y el auditado analiza la respuesta, puede o no estar de acuerdo con algunas prácticas y se deja asentado el documento final. Se evalúan los resultados, se hace una reunión de cierre y se hace entrega del reporte final.

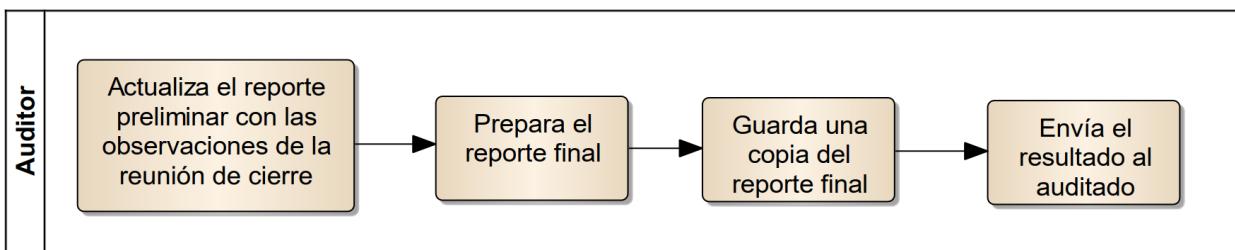
- Evaluación de Resultados



- Reunión de cierre

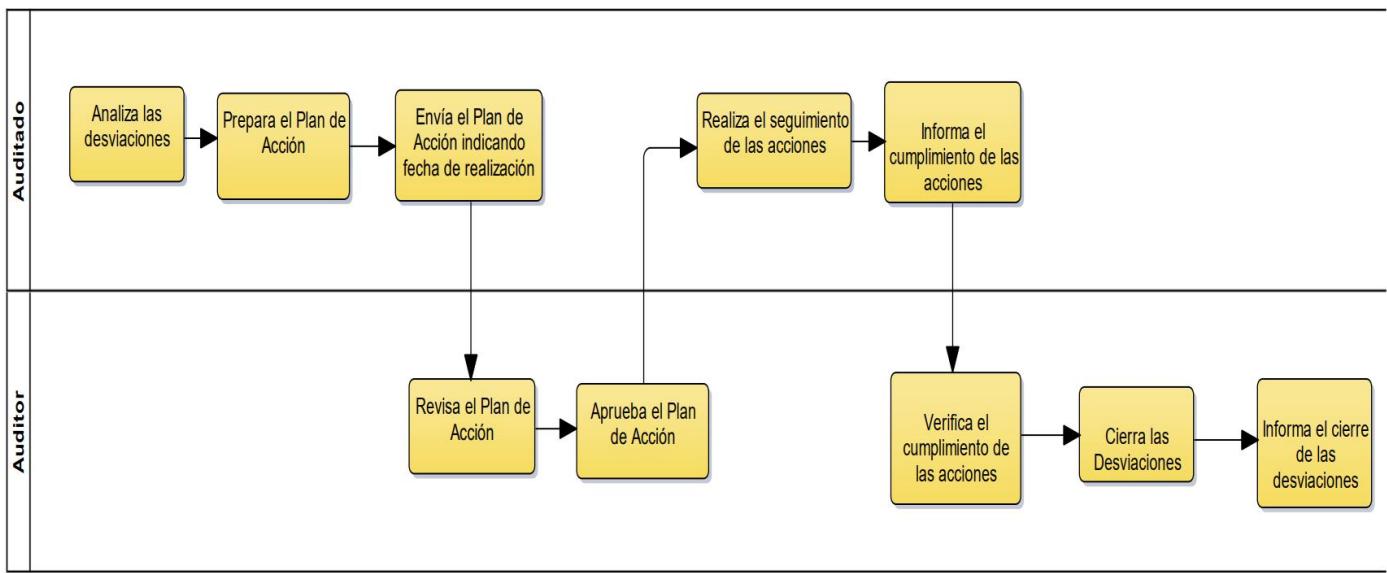


- Entrega de reporte final



4. Seguimiento

Se analizan las desviaciones y prepara un plan de acción que se envía al auditor para que lo revise y apruebe. En función del acuerdo entre auditado y auditor, puede que el auditor haga un seguimiento de las desviaciones que encontró hasta que considere que han sido resueltas.



Herramientas y técnicas utilizadas en auditorías

- Checklists: tienen preguntas tipo para garantizar que independientemente de quién haga la auditoría el foco de las cosas que se controlan sea el mismo. Son de mínima, es decir, se debe garantizar de mínimo contestar estas preguntas, pero el auditor puede solicitar más cosas (hacer más preguntas).
- Muestreo: consiste en seleccionar una muestra representativa de los productos y/o procesos a auditar.
- Revisión de registros
- Herramientas automatizadas

Resultados/hallazgos de una auditoría

- **Buenas prácticas**: cuando es algo superador a lo definido que tenemos que hacer. Mucho mejor de los esperado. Ejemplo: se armó una muy buena herramienta para gestionar el plan de proyecto para que la gente pueda accederlo y mejorarlo.
- **Desviaciones**: cualquier cosa que no se hizo o que no se hizo cómo el proceso lo definió. Hay dos grados:
 - No Adecuado: lo que realmente está mal.
 - Necesita Mejora: si lo estás haciendo, pero necesita mejorarse.

Requiere un plan de acción por parte del auditado.

- **Observaciones**: son cosas que se advierten por el auditor que no llegan a ser desviaciones pero que pueden llegar a ser riesgosas las prácticas y pueden llegar a generar un problema, por eso se destacan a **consideración** del equipo para que ellos decidan si hacerlo o no. Algo para revisar. Deberían mejorarse, pero no requieren un plan de acción. Ejemplo: técnica de estimación mala.

Reporte de auditoría

En este documento se deben informar las siguientes desviaciones:

- Cualquier desviación que resulta en la disconformidad de un producto respecto de sus requerimientos
- Cualquier desviación al proceso definido o a los requerimientos documentados.

Métricas de auditoría

Cada organización establece según sea apropiada o no:

- Esfuerzo por auditoría.
- Cantidad de desviaciones.
- Duración de auditoría.

Calidad de Producto: Planificación de pruebas para el software- Niveles y tipos de pruebas para el software. Técnicas y herramientas para probar software. Técnicas y Herramientas para la realización de revisiones técnicas del software.

Verificación y validación

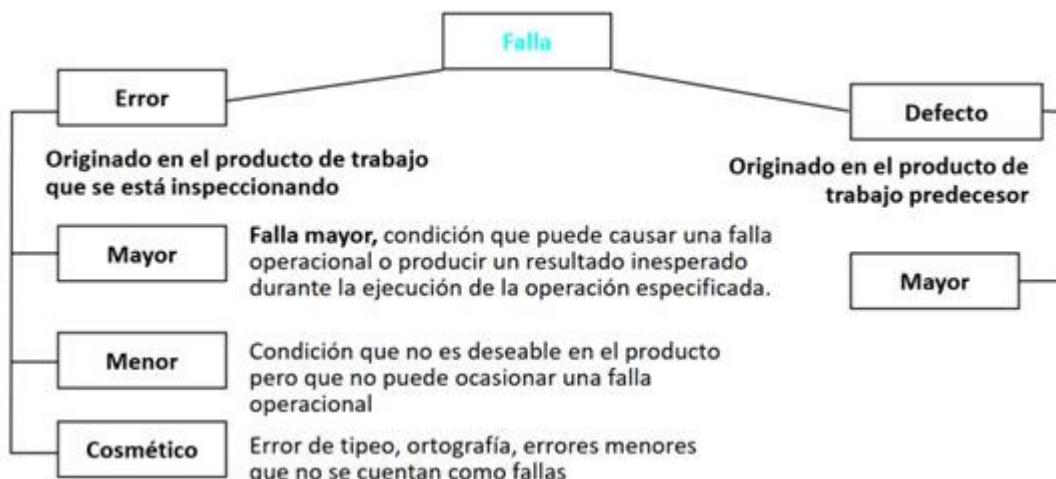
- Es un proceso de ciclo de vida completo.
- Inicia con las revisiones de los requerimientos y continúa con las revisiones del diseño, inspecciones del código hasta la prueba.
- Validación: ¿Estamos construyendo el producto correcto?
- Verificación: ¿Estamos construyendo el producto correctamente?

Falla: error en un producto de trabajo.

Producto de trabajo: salida de cualquier actividad correspondiente al ciclo de vida de desarrollo.

¿Por qué existen las fallas?

- Problemas de comunicación.
- Limitaciones de memoria.



Principios

- Prevenir es mejor que curar.
- Evitar es más efectivo que eliminar.
- La retroalimentación enseña efectivamente.
- Priorizar lo rentable.

- Olvidarse de la perfección, no se puede conseguir.
- Enseñar a pescar, en lugar de dar el pescado.

Existen dos aproximaciones complementarias:

- Revisiones técnicas.
- Pruebas de Software.

Revisiones técnicas – Peer Review

Es una actividad realizada por un colega, cuyo propósito es mejorar la calidad de software, mediante la detección temprana de errores en cualquier artefacto que se genere, por ejemplo, en el código, requerimientos, diseño, arquitectura, riesgos, estimaciones, planes, etc.

Es un proceso estático de validación y verificación; y no corrige errores.

Objetivo: introducir el concepto de verificación y validación. Busca evitar el retrabajo. Motiva a realizar un mejor trabajo.

Nunca se pone en juicio el autor del artefacto, sólo el artefacto en sí, ya que es necesario que se revelen todos los errores entre los miembros del equipo. Se debe desarrollar una cultura de trabajo que brinde apoyo y no buscar culpables cuando se descubran errores.

Es una actividad que trascendió cualquier metodología, filosofía y en muchas ocasiones la utiliza ágil, para transformar las auditorías en peer reviews evitando así traer a alguien externo al equipo.

Por ejemplo, siendo un programador Jr, solicitar a un Sr una revisión técnica respecto a un patrón de diseño arquitectónico que se quiere implementar.

Ventajas:

- Pueden descubrirse muchos errores;
- Pueden inspeccionarse versiones incompletas;
- Pueden considerarse otros atributos de calidad.

Desventajas:

- Es difícil introducir las inspecciones formales;
- Sobrecargan al inicio los costos y conducen a un ahorro sólo después de que los equipos adquieran experiencia en su uso;
- Requieren tiempo para organizarse y “parecen” ralentizar el proceso de desarrollo.

Tipos de revisiones

- Formales: tienen un proceso definido con roles.
 - Inspecciones (Inspección de código de Fagan e inspección de Gilb).
- Informales: cuando no existe un proceso de cómo realizarlo.
 - Walkthrough o recorrido.

Walkthrough o recorrido (informal)

Técnica de análisis estático en la que un diseñador o programador dirige miembros del equipo de desarrollo y otras partes interesadas a través de un producto de software, donde los participantes formulan preguntas y realizan comentarios acerca de posibles errores, violación de estándares de desarrollo y otros problemas.

No existe un proceso formal. Consiste en reuniones informales de colegas donde se debaten las correcciones a aplicar al producto de trabajo. No hay control del proceso.

Tiene los siguientes objetivos:

1. Mínima Sobrecarga
2. Capacitación de Desarrolladores
3. Rápido retorno

Esta técnica no obtiene métricas para aprender y dejar registros. Sin embargo, es una de las técnicas más elegidas en los enfoques agiles. Hay una junta de revisión entre colegas después de completar cada iteración del software (una revisión rápida), en la que pueden exponerse los conflictos y problemas de calidad sobre el producto.

Inspecciones (formal)

Tiene un proceso formal y cuenta con un conjunto de roles. Es necesario la utilización de un checklist, que ayuda a la memoria para saber que cosas controlar. Se toman métricas y finalmente se realiza un reporte de la revisión al final de la inspección para analizar los defectos encontrados.

Es una actividad que garantiza la calidad del software, cuyo éxito depende de la planificación. Tiene como objetivos:

1. Descubrir errores.
2. Verificar que el software alcanza sus requerimientos.
3. Garantizar que el software fue construido de acuerdo con ciertos estándares.
4. Conseguir un software desarrollado de manera uniforme.
5. Hacer que los proyectos sean más manejables.

Son procesos time boxing y exigen un alto esfuerzo intelectual.

Roles participantes

Al ser una técnica formal, si o si debe contar con los siguientes roles:

1. **Autor:** creador o encargado de mantener el producto a inspeccionar. Inicia el proceso seleccionando a un moderador y junto a este eligen al resto de los roles. Entrega el producto a ser inspeccionado al moderador.
2. **Moderador:** planifica y lidera la revisión. Trabaja junto al autor para elegir los demás roles. Entrega el producto a inspeccionar al inspector 2 días antes de la reunión. Coordina la reunión de forma tal que no ocurran conductas inapropiadas y realiza un seguimiento de los defectos encontrados.
3. **Anotador:** registra los hallazgos de la inspección. Usualmente termina confeccionado el reporte de la revisión.
4. **Lector:** lee el producto a ser inspeccionado. Este rol es necesario para que los participantes no se dispersen.

5. **Inspector:** Examina el producto antes de la reunión para encontrar defectos. Registra sus tiempos de preparación. todos pueden ser inspectores. Todos pueden inspeccionar.

Ciertos roles pueden ser asumidos por la misma persona.

Las revisiones técnicas...

SON	NO SON
<ul style="list-style-type: none">● La forma más barata y efectiva de encontrar fallas● Una forma de proveer métricas al proyecto● Una buena forma de proveer conocimiento cruzado● Una buena forma de promover el trabajo en grupo● Un método probado para mejorar la calidad del producto	<ul style="list-style-type: none">● Utilizadas para encontrar soluciones a las fallas● Usadas para obtener la aprobación de un producto de trabajo● Usadas para evaluar el desempeño de las personas

Etapas del proceso de inspección

1. **Planificación:** el moderador, a pedido del autor, planifica la inspección definiendo el lugar, el tiempo de duración y los roles. La duración de las reuniones no debe superar las 2 horas, dado que la inspección es un proceso de alto esfuerzo intelectual.
2. **Visión general:** esta etapa es opcional. El autor realiza una descripción general del producto a inspeccionar.
3. **Preparación:** es la preparación de cada rol para la reunión. Cada rol adquiere una copia del producto de trabajo que deberá leer y analizar, con el fin de encontrar potenciales defectos. Esta preparación permite que la reunión de inspección sea más productiva.
4. **Reunión de inspección:** el equipo realiza un análisis para recolectar los potenciales defectos previos y descartar falsos positivos. El lector lee el producto de trabajo y los inspectores comparten los defectos encontrados, los cuales son registrados por el anotador. La reunión finaliza con una conclusión acerca de si se acepta o no el producto de trabajo inspeccionado. Finalmente se realiza un informe detallando que se revisó, por quien, que se descubrió y que se concluyó.
5. **Corrección:** finalizada la reunión, el autor realiza las correcciones de los defectos encontradas.
6. **Seguimiento:** Dependiendo de la gravedad puede existir un proceso de re-inspección. En caso de que los defectos sean graves, se realiza nuevamente una inspección. De lo contrario, si el defecto era muy simple, el autor simplemente lo corrige. Otros defectos que no implican una re-inspección, pueden implicar que el autor se reúna con el moderador únicamente para tratar las correcciones realizadas.

Definición de estándares

Un aspecto importante del aseguramiento de calidad es la definición o selección de estándares que deben aplicarse al proceso de desarrollo de software o al producto de software.

Los estándares son importantes por tres razones:

- Se basan en conocimiento sobre la mejor o más adecuada práctica para la organización. Con frecuencia, este conocimiento se adquiere sólo después de una gran cantidad de pruebas y errores. Configurarla dentro de un estándar, ayuda a la empresa a reutilizar esta experiencia y a evitar errores del pasado.
- Al usar estándares se establece una base para decidir si se logró un nivel de calidad requerido. Desde luego, esto depende del establecimiento de estándares que reflejen las expectativas del usuario para la confiabilidad, la usabilidad y el rendimiento del software.
- Los estándares aseguran que todos los trabajadores dentro de una organización adopten las mismas prácticas. En consecuencia, se reduce el esfuerzo de aprendizaje requerido al iniciar un nuevo proyecto o trabajo.

Para la gestión de calidad de software se utilizan dos estándares:

- Estándares de producto: Se aplican al producto de software a desarrollar. Incluyen estándares de documentos y estándares de codificación.
- Estándares de proceso: establecen los procesos que deben seguirse durante el desarrollo, por ejemplo, incluyen las definiciones de requerimientos, procesos de diseño y validación, etc.

Testing en ambientes Ágiles.

Contexto

El testing de software, es una de las tareas a realizar en el ámbito del aseguramiento de calidad de software, en conjunto con el control de calidad del proceso y del producto. La calidad del software se obtiene y se logra a lo largo de todo el desarrollo de este (detectar errores en etapas tempranas es siempre menos costoso, que detectarlos después), por lo que una buena administración de configuración (SCM) es el puntapié inicial para permitir la realización de tareas de aseguramiento de calidad.

Hay que recordar que SCM permite determinar la configuración de ítems, trazabilidad, control de cambios, auditorías del producto y reportes de estado. QA agrega a esto, el control de calidad del proceso. Existen diversos estándares (ISO, CMMI, etc.) que permiten acreditar la calidad del proceso, debido a que la calidad del producto no es algo estandarizable por la variación de requerimientos de un caso a otro.

Dentro del aseguramiento de la calidad del software, existen actividades destinadas al control de calidad del proceso y del producto. Se realiza un control de calidad sobre el proceso de desarrollo, bajo el argumento de que el proceso de desarrollo posee un gran impacto en la calidad del producto. Es decir, en teoría la calidad del producto depende de la calidad del proceso que se está utilizando.



Luego de desarrollar el software, las actividades de testing revelan la calidad del Software en sí, es decir, si el producto satisface con los requerimientos definidos por el cliente en etapas anteriores. Cabe destacar que la actividad de testing no asegura la calidad por sí solo, sino que debe estar acompañada de las demás actividades.

QA != Testing

Definición de Testing

Proceso mediante el cual se somete a un software o un componente de software a condiciones específicas con el fin de determinar y demostrar si el mismo es válido o no en función de los requerimientos especificados.

El testing es una actividad destructiva cuyo objetivo es encontrar defectos, cuya presencia es asumida de antemano en el código.

Testing de software es un proceso, o una serie de procesos, diseñados para asegurar que el código hace lo que fue diseñado para hacer, o visto de otra manera, que no haga nada que no esté intencionado. Se dice que es el proceso de ejecutar un programa con la intención de encontrar fallas. El software debería ser predecible y consistente, sin presentarle sorpresas a los usuarios.

El Testing es parte del QA (Quality Assurance) y forma parte del control de calidad. Se hace cuando el producto ya está construido (o lo que se desea testear), en cambio el QA se aplica en todo el ciclo de vida (las buenas prácticas en la implementación son parte de QA).

Las actividades de testing se consideran exitosas si se encuentran defectos en la ejecución de los casos de prueba. Por lo que, un software con alta calidad lograda a lo largo de todo el desarrollo (implementación de QA), dificulta encontrar defectos y puede dirigir a un testing no exitoso. Por otro lado, si el software tiene un bajo nivel de calidad, el costo de retrabajo es alto, ya que van a existir muchos idas y vueltas entre las actividades de testing y las de desarrollo, para la corrección de errores.

El testing es una actividad costosa, por lo que es necesario lograr un nivel de cobertura óptima teniendo en cuenta el costo-beneficio. Nunca se podrá cubrir el 100% de las pruebas, por cuestiones lógicas de tiempo y costos. Esta cobertura se comienza a definir desde el momento en los cuales se establecen los criterios de aceptación.

Principios del Testing

- El Testing es una actividad destructiva que encuentra defectos cuya presencia se asume.
- Se testea con una actitud negativa tratando de demostrar que algo es incorrecto.
- El Testing exitoso es aquel que encuentra defectos.
- El costo del Testing está entre un 30% y un 50% del valor del producto.
- El Testing pone en evidencia defectos, pero no agrega calidad ni garantiza que el producto no tiene errores.
- Un desarrollo exitoso puede llevar a un Testing no exitoso.
- El Testing es necesario siempre.
- Se parte de la suposición de que siempre se tendrá defectos por encontrar, ya que es una característica inherente de los productos desarrollados por equipos de personas.
- Puede empezar antes de la codificación porque necesita de los requerimientos para armar los casos de prueba.
- El testing NO asegura que se tenga un producto de calidad (ni la agrega), ni que el proceso por el que se desarrolló sea de calidad.

- Agrupamiento de defectos: los defectos en el software suelen agruparse en un conjunto limitado de módulos o áreas (regla de Pareto, 80% de los defectos se agrupan en el 20% de la funcionalidad). Bajo este principio, las pruebas deben priorizarse y enfocar el esfuerzo en ese conjunto acotado de funcionalidades.

Principio	Explicación
1. Una parte necesaria de un caso de prueba es definir el resultado esperado.	Si el resultado esperado de un caso de prueba no ha sido predefinido, lo más probable es que un resultado erróneo se interpretará como un resultado correcto, debido a el fenómeno de "el ojo viendo lo que quiere ver", a pesar de la definición destructiva adecuada de prueba, hay todavía un deseo subconsciente de ver el resultado correcto.
2. Un programador debe evitar testear su propio programa.	Los propios desarrolladores "no quieren" encontrar sus propios defectos, por lo que el carácter de pruebas destructivas se deja de lado. Además, puede que el programa contenga errores por malentendidos del programador sobre el dominio, y si él mismo testea, no se van a detectar estos defectos.
3. Una empresa de desarrollo no debe testear sus propios programas.	Misma explicación que el principio 2 orientado a empresas, agregando que si las mismas empresas testean sus programas, es posible que destinen menos recursos y eviten encontrar defectos para cumplir con el calendario y los costos establecidos.
4. Cualquier proceso de prueba debe incluir una inspección minuciosa de los resultados de cada prueba.	Se deben realizar inspecciones minuciosas para detectar la totalidad de los defectos encontrados tras la ejecución de una prueba, ya que pueden surgir más de un defecto (y esto es lo que se busca) por cada caso de prueba.
5. Los casos de prueba deben escribirse para condiciones de entrada que no son válidas e inesperadas, así como para las que son válidas y esperadas.	Muchos errores que se descubren repentinamente en el desarrollo de software aparecen cuando se usa de alguna manera nueva o inesperada, y no responde cómo debería (catcheando el error)
6. Examinar un programa para ver si no hace lo que se supone que debe hacer es sólo la mitad de la batalla; la otra mitad es ver si el programa hace lo que no se supone que debe hacer	Los programas deben ser examinados para detectar defectos secundarios no deseados.
7. Evite los casos de prueba desechables, a menos que el programa sea realmente un programa de descarte.	Los casos de pruebas utilizados en el testing deben ser <u>reproducibles</u> , es decir, no se deben realizar casos de prueba sobre la marcha (ad-hoc) ya que es imposible reportar un defecto sin tener las condiciones y los pasos en los cuáles el defecto surgió. Además, realizar testing es muy costoso, por lo cuales los casos de prueba deben ser reutilizados para volver a testear escenarios luego de la corrección de errores.

8. No planee un esfuerzo de prueba bajo la suposición tácita de que no se encontrarán errores.	Es un error pensar de esta manera al momento de realizar software. Se debe tener en claro la definición de testing, en la cual se define que el objetivo de esta actividad es encontrar errores, y se presume de antemano su existencia.
9. La probabilidad de que existan más errores en una parte de un programa es proporcional al número de errores ya encontrados en esa parte.	El concepto es útil porque nos da una idea de en qué sección del programa hacer foco o asignar más recursos, si una sección particular de un programa parece ser mucho más propenso a errores que otras secciones, es recomendable realizar pruebas adicionales y es probable que encontremos más errores.
10. Las pruebas son extremadamente creativas e intelectualmente desafiantes.	Aunque existen métodos y estrategias para abarcar un mejor nivel de cobertura testing en los casos de pruebas, siempre es necesario un poco de creatividad del diseñador de estos.

Mitos del Testing

- “El testing es el proceso para demostrar que los errores no están presentes”.
- “El propósito del testing es demostrar que un programa realiza sus funciones previstas de forma correcta”
- “El testing es el proceso que demuestra que un programa hace lo que se supone que debe hacer”

Estas definiciones o afirmaciones sobre el testing son incorrectas, ya que el objetivo de testear un programa es agregar valor al producto revelando su calidad y brindando confianza en el software, de forma más concreta, encontrar y remover defectos en el código de este. Entonces, no se prueba un sistema para mostrar que funciona, sino que se comienza con la suposición de que el software contiene defectos, y se realiza el testing para encontrar la mayor cantidad de ellos posible.

Por otro lado, un programa puede hacer lo que se supone que debe hacer, y aun así, contener defectos. Es decir, un error está claramente presente si un programa no hace lo que se supone que debe hacer, pero los errores también están presentes si un programa hace lo que no se supone que debe hacer.

¿Cuánto Testing es suficiente?

El **testing exhaustivo es imposible** por la cantidad de tiempo que requiere. El momento en que se deja de hacer testing depende del nivel de riesgo o costo asociado al proyecto. Los riesgos permiten definir prioridades de que se debe testear primero y con qué esfuerzo.

El criterio de aceptación se utiliza normalmente para decidir si una determinada fase de testing ha sido completada. Este puede ser definido en términos de:

- Costos.
- % de tests corridos sin fallas.
- Inexistencia de defectos de una determinada severidad.

- Pasa exitosamente el conjunto de pruebas diseñado y la cobertura estructural.
- Good Enough: Cierta cantidad de fallas no críticas es aceptable.
- Defectos detectados es similar a la cantidad de defectos estimados.

El criterio de aceptación sirve para definir y negociar en ágil con el Product Owner y en tradicional con el Líder del Proyecto a cuántos defectos son aceptables para terminar.

Conceptos importantes

Defecto versus Error

La diferencia entre ambos conceptos es el momento en el cual se detectan y solucionan los errores o defectos. Un error es detectado y corregido en una misma etapa, y un defecto es un error que se traslada de una etapa a otra etapa posterior en la cual se introdujo. El testing encuentra defectos, ya que son errores que surgieron en etapas anteriores, pero se detectan en la etapa de prueba.

Hay que aclarar que ambos conceptos pueden o no generar fallas en el sistema, es decir, un mal funcionamiento de este. Por ejemplo, un error en los colores de una interfaz no es una falla, ya que no conllevan a un mal funcionamiento del sistema.

Defecto

Un defecto posee dos características principales, que permiten catalogarlos en la etapa de testing:

- Severidad: define la gravedad del defecto, y es determinada por la persona que realiza el testing, por lo que esta característica es de carácter técnico. El valor de la severidad se asigna dependiendo de la siguiente escala:
 - Bloqueante → el defecto no permite continuar con la ejecución del sistema.
 - Crítico → el sistema funciona, pero la funcionalidad que se está testeando tiene un defecto crítico.
 - Mayor → la funcionalidad que se está testeando funciona, pero no de forma correcta.
 - Menor → la funcionalidad se ejecuta correctamente, pero con advertencias erróneas o errores de baja importancia.
 - Cosmética → formato de fechas, formato de números, distribución de componentes en una GUI, temas de presentación de interfaces, etc.
- Prioridad: la prioridad define el impacto del defecto en la funcionalidad para el negocio, y permite ordenar la atención de los defectos según las necesidades del cliente. Una escala posible para la prioridad puede ser:
 - Urgencia
 - Alta
 - Media
 - Baja

Estas características son independientes entre sí, ya que varían dependiendo del tipo de negocio y de sistema que se está desarrollando. Por ejemplo, un defecto cosmético puede tener baja prioridad para un sistema de inventarios, pero puede tener una alta prioridad si se trata de un sistema de una empresa de marketing, en donde el aspecto y la presentación es algo muy importante.

Casos de prueba

Son una secuencia de pasos que hay que seguir para obtener un resultado esperado, y se tienen que dar ciertas condiciones previas que fijan una situación para que se pueda ejecutar la prueba.

Es el artefacto más importante del Testing. Este se hace una vez, pero sirve para realizar infinitas pruebas, debiendo obtener en todas el mismo resultado esperado.

Se trata de minimizar la cantidad de casos de prueba maximizando la cantidad de defectos encontrados.

El objetivo de los casos de prueba es descubrir defectos.

Los casos de prueba salen de los requerimientos, permitiendo hacer tanto la verificación como la validación.

Está compuesto por: un objetivo (lo que se desea controlar), condiciones de prueba (Datos de entrada y de entorno que deben estar presente para llevarse a cabo la prueba) y un resultado esperado.

Ciclos de prueba

Es la ejecución de un conjunto de casos de prueba en una versión determinada del producto. Generalmente se tienen 2 ciclos, y el primero es conocido como ciclo 0. El ciclo 0 siempre es manual, es donde se configura todo y a partir del ciclo 1 ya se pueden automatizar las pruebas.

En caso de detectarse defectos, el producto vuelve a desarrollo para su corrección.

Si existe una cantidad alta de ciclos de prueba, es probable que QA no sea bueno, por lo que deberían revisarse ciertos aspectos acerca de la calidad del proceso y producto, de manera de evitar tener grandes cantidades de ciclos, que se traducen en mucho retrabajo, lo cual a su vez conduce a incrementar los costos de desarrollo.

Ciclo de pruebas con regresión

Este concepto plantea la necesidad de ejecutar exhaustivamente todos los casos de prueba en cada ciclo de prueba, como si fuesen el ciclo 0. Este planteo es el ideal, pero el menos utilizado, por cuestiones de practicidad y tiempo.

El otro enfoque, consiste en aplicar una prueba exhaustiva de todos los casos de prueba en el ciclo 0, pero a partir del ciclo 1 (si el incremento o producto vuelve nuevamente a testing) ejecutar sólo los casos de prueba asociados a defectos encontrados previamente, y no los casos de prueba que hayan pasado sin encontrar defectos. Esto permite agilizar el proceso de pruebas, pero estadísticamente es muy probable que al arreglar un defecto reportado por testing, en el desarrollo de su corrección se introduzcan nuevos errores en otras partes del producto. Por lo que, si no se prueban todos los casos, a pesar de que previamente no se hayan detectado defectos, pueden encontrarse nuevos, debido al proceso descrito anteriormente.

Una solución a esta encrucijada es no aplicar regresión (ejecutar sólo los casos de prueba con defectos asociados), y cuando finalmente estén todos los defectos “corregidos”, hacer una ejecución de todos los casos de prueba (como si se aplicara regresión), para aumentar la confianza de que no se han introducido nuevos errores en las correcciones.

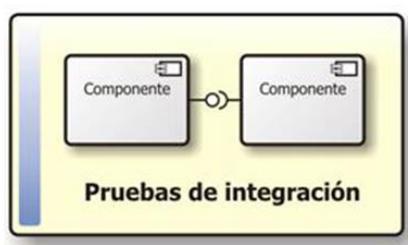
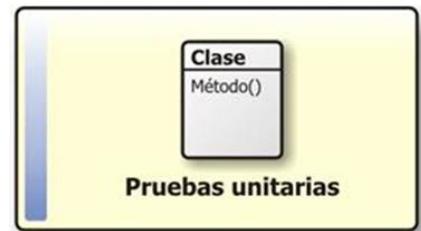
La automatización del testing permite hacer tantos ciclos de prueba con regresión como se deseen. La desventaja es el proceso de automatizar la ejecución de los casos de prueba, pero se puede ver ese esfuerzo de automatización como el esfuerzo propio del ciclo 0 manual, y luego esa automatización permite ejecutar los casos de prueba N veces. En empresas donde el core de negocio no es hacer software, conviene asumir ese esfuerzo y automatizar las pruebas.

Claramente, la mejor estrategia es aplicar regresión, porque no existe un punto medio al ser un método de caja negra, ya que no se puede saber con certeza lo que va a afectar al corregir un defecto y en dónde impactará esa corrección. Sin embargo, en la práctica se utiliza sin regresión, debido a los costos y tiempos que implica ese proceso.

Niveles de prueba

Los niveles de prueba determinan el foco o la granularidad de la prueba que se está por ejecutar. En general, primero se comienzan probando componentes pequeños, y luego se integran en pruebas de mayor granularidad. Esto es al revés de cómo se desarrollan los componentes.

- **Pruebas unitarias:** las hace el desarrollador y están incluidas en el DoD. Son pruebas que se realizan sobre un componente, de manera independiente a los otros. Se hacen teniendo acceso al código fuente, y se pueden usar herramientas para realizarlas (automatización, depuración). Se suelen reparar los errores apenas se encuentran, sin registrarlos formalmente.
- **Pruebas de integración:** El propósito de la ejecución de estas pruebas es verificar el funcionamiento de dos o más componentes juntos que se relacionen entre ellos, es decir, clases en las cuales existe una interacción a modo de peticiones, a través de sus interfaces (boundaries). El resultado de estas pruebas es el build (el CI o continuous integration), el cual luego se envía al equipo de testing y es lo que se prueba en siguientes etapas.



Se suele llevar a cabo una prueba de integración incremental, desde lo más general (que abarca más componentes) a lo más específico (top-down) o desde lo más específico a lo más general (bottom-up).

- **Pruebas de sistema o de versión:** En estas pruebas se realizan testeos sobre la versión de un incremento de producto o de un producto (dependiendo si se usa Agile o métodos tradicionales), y existen razones psicológicas que demuestran que deben realizarlas personas ajenas al desarrollo de los programas (obligatoriamente). Estas pruebas se llevan adelante siguiendo un proceso sistemático y metodológico, que permite encontrar defectos que puedan ser reproducibles y reportar de qué manera ocurre el defecto detectado, es decir, cual es el camino de pasos que hay que seguir para encontrar el error. Estas pruebas están basadas en casos de prueba. Se trata de emular de la mejor manera posible un entorno de trabajo idéntico al entorno real en el que se usara el SW. Se llevan a cabo pruebas del funcionamiento real y cotidiano del producto. Abarca requerimientos funcionales y no funcionales.



- **Pruebas de aceptación de usuario:** se realizan en el despliegue y debería realizarlas el usuario para verificar que se cumple con lo que el mismo requirió. Busca que el cliente/usuario se familiarice con el producto, generando comodidad y confianza sobre el mismo (su objetivo principal no es encontrar fallas). También busca reproducir un entorno de producción. Comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa), como la prueba en



ambientes de trabajo reales (pruebas beta). En la teoría, los usuarios arman sus pruebas de usuario. En la práctica, las arma Testing.

En la teoría, además del usuario, deben estar presentes el gerente de testing, el líder del proyecto y empleados con roles funcionales.

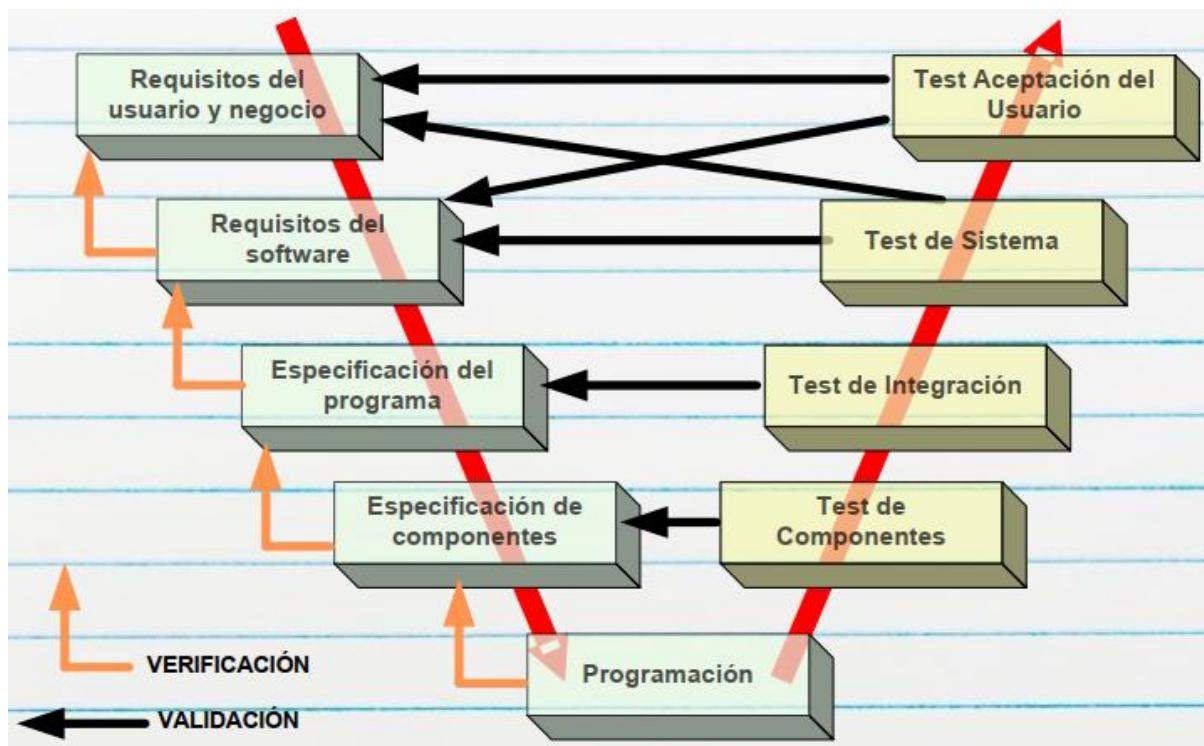
En Agile, además del usuario, deben estar todos los miembros del equipo (Sprint Review).

Modelo en V

Este modelo se utiliza para poder determinar cuándo se está en condiciones de realizar determinadas actividades, en función de la etapa de desarrollo en la que se encuentre el software.

Acá se puede notar cómo las pruebas se realizan en orden de granularidad inverso al proceso de desarrollo del software. Es decir, se desarrolla desde componentes de granularidad gruesa, como son los requerimientos abstraídos de detalles de implementación, hacia componentes de menor granularidad, como son clases o módulos, dependiendo del paradigma. En cambio, para las pruebas, la granularidad se da en sentido inverso.

- Testing de Componentes → Testing unitario.
- Especificación del programa → diseño/arquitectura.



Ambientes de Testing

Son todos los recursos, tanto hardware como software, que se requieren para poder trabajar con el producto. Existen distintos ambientes en el desarrollo de software, los cuales poseen distintas necesidades, según la etapa de desarrollo en la que se encuentre el software.

Desarrollo

Implica hardware y software necesarios (librerías, extensiones, IDEs, compiladores, etc.) para poder desarrollar y desplegar el producto y utilizarlo. En este ambiente se realizan las pruebas unitarias (y también las de integración generalmente).

Pruebas

Es el ambiente que utilizan los testers para llevar a cabo las pruebas, y los desarrolladores no deben tener acceso. En este se realizan las pruebas del sistema. Normalmente acá se realizan las pruebas de sistema.

Preproducción

Debería tener las mismas características que el ambiente productivo para poder comprobar que el producto funcionará una vez desplegado en producción de manera correcta. En este ambiente se realizan las pruebas de aceptación.

El problema de este entorno es que muchas veces resulta difícil (o imposible en algunos casos), debido a que es muy costoso replicar principalmente las configuraciones de hardware. Por ejemplo, sería prácticamente imposible replicar la configuración de servidores del motor de búsqueda de Google, para realizar pruebas de aceptación.

Producción

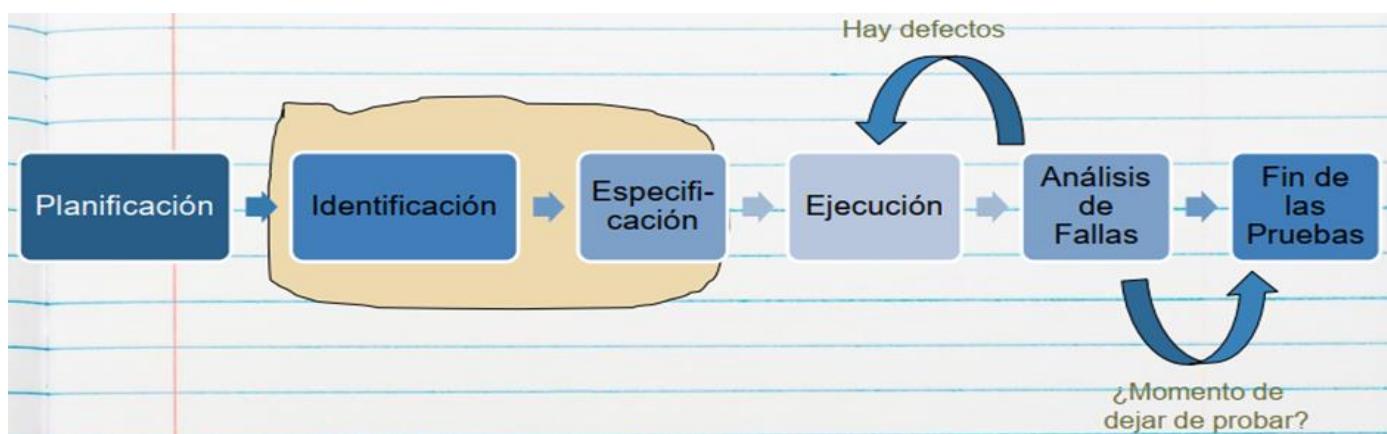
Este entorno, es la configuración de software y hardware que tienen los usuarios finales del software, y que utilizan para el desempeño de sus actividades laborales. Obviamente en este entorno no se realizan pruebas (probar en este ambiente tiene grandes consecuencias), ya que en teoría la versión del producto cumple con los criterios del Definition of Done.

Proceso de pruebas

El Testing es un proceso definido (que se lleva a cabo durante todo el ciclo de vida del producto), por lo que está compuesto de etapas detalladas.

Testing Ad Hoc: se realiza testing sin ningún proceso definido, perdiendo la trazabilidad ya que no se registra el paso a paso de lo que se probó, sino que se prueba libremente sin llevar ningún registro.

Generalmente el proceso de pruebas es estándar (puede tener algunas variaciones, pero sigue una estructura general).



- **Planificación:** Determina como se incluirá el Testing en el plan del proyecto, y como será el Test Plan (que recursos se usará, que riesgos se tendrá, cuál será el criterio de aceptación, que entornos se emularán, quien realizará cada Testing, cuando, etc.). El resultado de la planificación es el Plan de Pruebas, que debe contener:
 - Riesgos y objetivos del Testing.
 - Estrategia de Testing.
 - Recursos.

- Criterio de Aceptación.
- **Diseño (Identificación y especificación de casos de prueba):** Revisando las bases de la planificación del Testing, se identifican los datos necesarios, diseñan y priorizan los CP que se llevaran a cabo (se define que entorno se usara, como se llevaran a cabo, por quien, cuando, etc.). Analiza si los requerimientos son testeables o no. Se define si se usa regresión o no.
- **Ejecución:** Cuando se ejecutan los CP, se registran y se comparan los resultados generando un reporte de defectos (con defectos encontrados, condiciones, entornos). Se trata de automatizar lo más que se pueda respecto de estas ejecuciones (ahormando tiempo/costo). Incluye: Creación de los datos necesarios para la prueba, automatización de todo lo que sea necesario, implementar y verificar el ambiente, ejecutar los casos de prueba, registrar los resultados de la ejecución y comparar los resultados reales con los esperados.
- **Análisis de fallas (Evaluación y Reporte):** Se hace un seguimiento de la corrección de los defectos encontrados hasta que se cierren todos los CP, es decir que se hayan solucionado todos. Se evalúa el criterio de aceptación, se reporta el resultado de las pruebas a los interesados, se verifica los entregables y que los defectos se hayan corregido y se evalúa cómo resultaron las actividades de testing y se analizan las lecciones aprendidas.

Acá se confecciona el informe de reportes.

- **Fin de las pruebas:** En las empresas más maduras se deja de probar recién cuando no hay defectos bloqueantes, críticos, mayores y los defectos menores y cosméticos son muy pocos, los que se ponen en la nota de Release. Se confecciona el informe final (opcional).

Artefactos de Testing

Plan de pruebas

Este plan, es análogo al plan de proyecto que se elabora en el enfoque tradicional (no forma parte de ese plan). En este plan, se definen:

- Datos necesarios para las pruebas.
- Recursos destinados a las pruebas.
- Herramientas a utilizar.
- Si se aplicará regresión (o no).
- Etc.

Para la confección de este plan no es necesario el código, sino que sólo se necesitan los requerimientos, en el formato que sea que se encuentren (ERS o Casos de Uso en métodos tradicionales, User Stories en Agile, etc.)

Casos de prueba

- Es el artefacto más importante del testing, y consiste en una secuencia de pasos a seguir, las cuales describen un conjunto de acciones a realizar para lograr un resultado concreto y esperado. Para esto se deben explicitar las condiciones de inicio (las cuales fijan una situación particular), y los datos utilizados en ese escenario.
- Se confeccionan con el objetivo de descubrir la mayor cantidad de defectos y minimizar el esfuerzo y tiempo para elaborarlos y ejecutarlos.
- Mientras no cambien los requerimientos, estos casos pueden ser utilizados las veces que sea necesario.
- La característica más importante de los casos de prueba es ser REPRODUCIBLES. Si se encuentra un defecto y no es reproducible a través de un caso de prueba, no es un defecto.

Los casos de prueba se derivan de diferentes fuentes:

- Documentos del cliente;
- Información relevada;
- Requerimientos;
- Especificaciones de programación;
- Código.

Reporte de incidentes o defectos

Luego de la ejecución de los casos de prueba, se elabora un reporte, indicando cuáles fueron los casos de prueba que han pasado, y aquellos en los que se ha encontrado defectos, indicando cuáles fueron esos defectos encontrados y cómo reproducirlos para su detección y corrección.

Este artefacto permite a los desarrolladores corregir esos defectos, para enviarlos nuevamente a testing.

Informe final

suele contener métricas e información final del incremento del producto, se reporta cuántos ciclos y la cantidad de errores por ciclo, métodos, tipos de prueba utilizados, etc. Este es el único artefacto que es negociable en algunas organizaciones informales, se acuerda en la contratación del trabajo.

Tiene utilidades estadísticas.

El Testing y el Ciclo de Vida

Si desarrollamos con ciclo de vida en cascada (Ciclo de Vida Secuencial) las pruebas se hacen al final ya que es el momento en el que nos entregan el producto. En cambio, si desarrollamos con ágil (Ciclo de Vida Iterativo/Incremental), hacemos testing por iteración. El problema es que se pueden incluir errores por la integración de los incrementos.

Estrategias de prueba

Se utilizan para pasar por la mayor cantidad de funcionalidades con la menor cantidad de casos pruebas confeccionados, debido a que el tiempo y el presupuesto para diseñarlos y ejecutarlos es limitado.

Hay dos grandes enfoques: Caja Negra y Caja Blanca. Cada uno tiene fortalezas y debilidades particulares: un método puede ser bueno para algunas cosas, y no para otras cosas. El mejor método es no usar un único método, sino usar una variedad de técnicas ayudará a un testing efectivo.

Caja Negra

En esta estrategia no se dispone de la estructura interna de la implementación, sino que se analizan las funcionalidades como una caja negra, en términos de entradas y salidas de esa “caja”.

El proceso consiste en ingresar determinados datos a esa funcionalidad vista como caja negra, y luego comparar los resultados obtenidos con los resultados esperados. Para ello, se realiza un testing exhaustivo de entrada, probando

cada posible entrada conducida como caso de prueba, no necesariamente las válidas. En transacciones no sólo se debe considerar todos los datos posibles, sino todas las secuencias posibles de transacciones previas.

Se clasifican en basados en especificaciones y basados en experiencia:

- Métodos basados en especificaciones: Son aquellos que se ejecutan utilizando la documentación de especificaciones realizadas del producto.
 - Partición de equivalencias: proceso sistemático que consiste en identificar clases de equivalencia, que definen subconjuntos de datos que producen un resultado equivalente.
 - Análisis de los valores límites: variante del anterior. Utilizar los límites o valores de borde de las clases de equivalencia para la definición de los casos de prueba.
- Métodos basados en experiencia: la experiencia y los conocimientos del tester son fundamentales para determinar las entradas del sistema y analizar los resultados.
 - Adivinanza de defectos: enfoque basado en la intuición y experiencia para identificar pruebas que probablemente expongan defectos del software, elaborando una lista de defectos posibles o situaciones propensas a error y realizando pruebas a partir de esa lista.
 - Testing exploratorio: el tester mientras va probando el software, va aprendiendo a manejar el sistema y junto con su experiencia y creatividad, genera nuevas pruebas a ejecutar.

Partición de equivalencias

Analiza las condiciones externas (entradas y salidas) involucradas en una funcionalidad determinada. A esas condiciones externas las divide en clases de equivalencia, las cuales son subconjuntos de valores posibles, que arrojan resultados equivalentes en la funcionalidad que se quiere probar.

Ejemplos de entradas: campos de texto, fechas, coordenadas de un mapa, archivos, señales de sensores, etc.

Ejemplos de salidas: mensaje en pantalla, advertencias, listados, tablas, emisión de señal, etc.

Procedimiento

1. Identificar condiciones externas de entrada y salida.
 - a. Si hay que ingresar dos datos que son iguales, pero con distintos valores, separarlo en condiciones externas diferentes. Por ejemplo: 2 calles, una condición externa es calle1 y la otra calle2.
2. Identificar subconjuntos de valores posibles (válidos y no válidos) de las condiciones externas identificadas, que produzcan resultados equivalentes en la ejecución de la funcionalidad.
 - a. Los subconjuntos no válidos tienen asociados mensajes de error en la condición externa de salida "Mensajes de Error" o "Mensajes de Advertencia" (no es necesario poner todos los mensajes, con algunos genéricos es suficiente).
 - b. Para no olvidarse de algún subconjunto → la unión de todos los subconjuntos debe ser igual al universo de la condición externa (teoría de conjuntos).
3. Armar los casos de prueba tomando de cada condición externa, un sólo valor particular (especificar) de cada subconjunto de valores posibles.

Consideraciones

- Prioridad
 - Alta → caminos felices o errores críticos en el dominio de negocio.
 - Baja → casos de prueba relacionados a validaciones (valores no ingresados, con mal formato, etc.).
- Nombre caso de prueba: describir el escenario de la funcionalidad que se está probando, ¡de forma clara! Por ejemplo: “Ingresar al sitio web con edad menor a 18 años”
- Precondiciones
 - Conjunto de características que tienen que cumplirse en el contexto de la funcionalidad, para que pueda ser ejecutada.
 - Deben ser valores concretos, específicos. Por ej: “El usuario Juan está logueado con permiso de administrador.” - “La fecha actual es 25/10/2022” - “Las coordenadas del GPS son: 41°24'12.2" N” - “La tarjeta tiene el número: XXXX XXXX XXXX XXXX”
 - Los datos que se seleccionan de entre un grupo determinado (por ejemplo forma de pago), son precondiciones que deben estar previamente cargadas en el sistema.
- Pasos
 - Siempre arrancan seleccionando la opción que desencadena la funcionalidad: “El *nombreUsuario* ingresa a la opción ...”
 - Después: “El *nombreUsuario* ingresa/selecciona ...”
 - No tiene que haber ambigüedad, pensar que otra persona los tiene que leer y ejecutar.
 - Normalmente finalizan con un botón de confirmación o algo así.
- Resultado esperado
 - Puede mostrar varias cosas el sistema.
 - Deben ser resultados concretos, específicos.
 - Suelen ser mensajes de advertencia/error, listados, posición en el mapa, etc. pero siempre indicando con un dato particular. Por ejemplo: El sistema muestra el mensaje de advertencia “Debe seleccionar una fecha”.
 - Se debe relacionar la salida con el paso en la cual se produce.

Caja blanca

En estos se dispone del código (puede ser también pseudocódigo o un diagrama de flujo). Nos permiten diseñar casos de prueba, maximizando la cantidad de defectos con la menor cantidad de casos de prueba posibles.

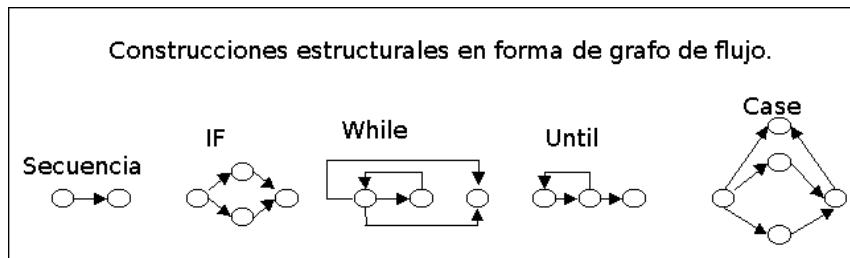
Tiene dos fallas: La primera es que el número de caminos lógicos únicos puede ser sumamente grande, tomando muchísimo tiempo de probar absolutamente todas de ellas. La segunda falla es que las pruebas de camino exhaustivas no aseguran que el programa sea el correcto para las especificaciones, tampoco detecta caminos faltantes ni determina errores de datos sensibles.

Hay distintas coberturas con nivel diferente (la forma de recorrer los distintos caminos que provee el código para desarrollar una funcionalidad):

Cobertura de enunciados o caminos básicos

- Objetivo → encontrar todos los caminos independientes de una funcionalidad que deben recorrerse (testear) al menos una vez, para probar cada uno de ellos con casos de prueba.
- Permite obtener una métrica denominada complejidad ciclomática (M), que representa la cantidad de caminos independientes que posee una funcionalidad, y nos da un límite inferior para el número de casos de prueba que hay que construir, para ejecutar todas las instrucciones de la funcionalidad al menos una vez.
- El procedimiento es:

1. En primer lugar, se requiere representar la funcionalidad a través de un grafo de flujo (los algoritmos que implementen métodos recursivos quedan fuera de esta prueba de cobertura)

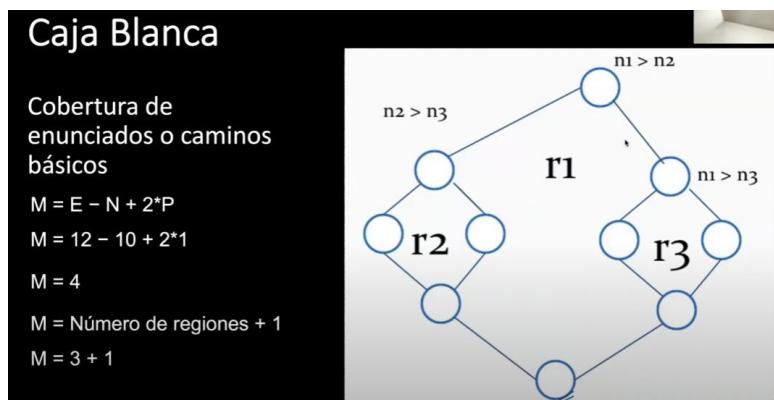


2. Luego se calcula la complejidad ciclomática (M). Para esto hay dos formas:

a. $M = E - N + 2*P$, siendo:

- M = complejidad ciclomática.
- E = número de aristas del grafo.
- N = número de nodos del grafo.
- P = número de componentes conexos o nodos de salida.

b. También se puede obtener la complejidad ciclomática como $M = \text{número de regiones cerradas} + 1$.



En el ejemplo, la cantidad mínima de casos de prueba para garantizar la cobertura de enunciados es de 4. Esto no quita que se puedan construir más de 4, pero esa es la menor cantidad de casos de prueba, que me permite maximizar la detección de defectos.

3. Se define el conjunto mínimo de caminos independientes (asignando valores que provocan ir tomando cada uno de los caminos de la funcionalidad). No son casos de prueba. Siguiendo el ejemplo:

TC 1	TC 2	TC 3	TC 4
N1 = 8	N1 = 8	N1 = 4	N1 = 4
N2 = 4	N2 = 4	N2 = 8	N2 = 8
N3 = 4	N3 = 8	N3 = 4	N3 = 8

4. Luego se diseñan y ejecutan los casos de prueba que permitan la ejecución de cada uno de los caminos identificados, donde los datos de la tabla de arriba se mapean a las precondiciones o a valores que ingresa el usuario en los casos de prueba (dependiendo de cómo es la funcionalidad).
5. Se ejecutan los casos de prueba y se comprueba que los resultados sean los esperados.
- Dependiendo del valor de la complejidad ciclomática (M), el programa puede entrar en una de las siguientes clasificaciones (heurística):

Complejidad Ciclomática	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, Muy alto riesgo

Cobertura de sentencias

- Sentencia: cualquier instrucción como asignación de variable, invocación de métodos, etc. Las estructuras de control NO son sentencias, son decisiones.
- Objetivo → buscar la cantidad mínima de casos de pruebas que me permitan pasar, ejecutar o evaluar todas las sentencias.
- Ejemplo: Partiendo de la siguiente porción de código, podemos determinar que existen 2 sentencias (de asignación de variables). Entonces, para ejecutar o evaluar las sentencias, es necesario solo un caso de prueba, asignando valores a "A", "C", "B" y "D" es posible ejecutar ambas sentencias ya que las condiciones de IF son independientes entre sí.

```

IF (A>0 && C==1)
    X = X + 1
IF (B==3 || D<0)
    Y=0
END
  
```

TC1
A=5
C=1
B=3
D=-3

Cobertura de decisión

- Decisión: estructura de control concreta, y dentro de cada decisión existen combinaciones que están unidas por operaciones booleanas de condiciones.
- Objetivo → buscar la cantidad mínima de casos de prueba que me permitan evaluar todas las ramas de las decisiones. Esto apunta a que el código tome cada rama de forma correcta, es decir, evaluar que la decisión derive en la rama del true y en la rama del false cuando corresponda en cada caso.

- En una decisión IF, si o si necesitamos dos casos de prueba: rama true y rama false.
- Las decisiones son las que se encuentran dentro de los paréntesis de los IF, y cada uno de los términos son condiciones, es decir $A>0$ y $C==1$, y el conjunto de las condiciones unidas por el booleano, representan una combinación ($A>0 \&& C==1$).
- En esta cobertura, no interesa qué condición (o combinación de condiciones) hay dentro de la decisión, siempre y cuando se garantice que la decisión es evaluada en su rama verdadera y en su rama falsa.
- En el ejemplo, con tan solo dos casos de prueba es suficiente, ya que las decisiones son independientes entre sí (no están anidados), por lo que independientemente de la rama que se evalúe en la primera decisión, voy a poder valorar cualquiera en la segunda. Entonces en el TC1 se evalúa la rama de “true” en ambas decisiones, y en TC2 se evalúa la rama de “false” en la dos.

```

IF (A>0 && C==1)
    X = X + 1
IF (B==3 || D<0)
    Y=0
END
  
```

TC1	TC2
A=5	A=5
C=1	C=5
B=3	B=5
D=-3	D=5

Cobertura de condición

- Condiciones: son cada una de las evaluaciones lógicas dentro de una decisión, que están unidas por operadores lógicos.
- Objetivo → buscar la cantidad mínima de casos de pruebas que me permitan evaluar cada una de las condiciones, en su valor verdadero y en su valor falso, independientemente de que rama se tome en la decisión en la que se encuentre la condición (recordar que la decisión está formada por 1 o más condiciones combinadas de forma booleana).
- Siguiendo el ejemplo anterior, son necesarios 2 casos de prueba, ya que las condiciones son independientes y se pueden evaluar todas las condiciones en true en una prueba, y todas las condiciones en false en otra prueba, debido a que no importa la rama que tome cada decisión.

TC1	TC2
A=0	A=5
C=1	C=5
B=3	B=5
D=-3	D=5

Cobertura de decisión condición

- Objetivo → evaluar las ramas verdaderas y falsas de las decisiones, y a su vez evaluar valores verdaderos y falsos de las condiciones.
- Siguiendo el ejemplo anterior, se necesitan 2 test cases, los cuales permiten evaluar las condiciones y decisiones al mismo tiempo en un mismo caso de prueba, con los valores de la imagen.

```

IF (A>0 && C==1)
    X = X + 1
IF (B==3 || D<0)
    Y=0
END
  
```

TC1	TC2
A=5	A=0
C=1	C=5
B=3	B=5
D=-3	D=5

Cobertura múltiple

- Objetivo → evaluar el combinatorio de todas las condiciones, en todos sus valores de verdad posibles.
- Por ejemplo: Si tuviésemos un caso como el representado en el grafo de flujo, los casos de prueba van a ser 7, ya que solo el valor de A y B verdaderos al mismo tiempo, permiten

1 DECISIÓN	A	F	F	V	V	V	V	V
	B	F	V	F	V	V	V	V
2 DECISIÓN	C	-	-	-	V	V	F	F
	D	-	-	-	V	F	V	F

ejecutar escenarios en donde se incluye la decisión de C y D. Este es un escenario con decisiones dependientes entre sí, si fuesen independientes, con 4 pruebas alcanza.

Tipos de prueba

Smoke Test

Es un tipo de prueba que se hace para validar que no haya fallas groseras, de gran magnitud, en el producto de software. Se realiza antes de comenzar el ciclo 0. Nos ahorra empezar a testear formalmente si encuentra un error, porque todavía hay algo que corregir.

Consiste en una corrida rápida para ver si el producto está en condiciones de pasar al ciclo de prueba.

Testing Funcional

Controla que el software se comporte de la misma manera que lo especificado en la documentación, cumpliendo con las funcionalidades y características definidas. Se basa en los requerimientos funcionales y el proceso de negocio. Hay testing funcional basado en dos aspectos:

- Basado en requerimientos: cuando se prueban requerimientos específicos, apunta a probar una funcionalidad sola (utilizan a los requisitos definidos en una ERS o los acuerdos que contienen las pruebas de usuario y los criterios de aceptación de una US para realizar las pruebas.)
- Basado en proceso de negocio: cuando se prueba un proceso de negocio completo, es decir, se prueba todo el proceso. Por ejemplo, en una venta se prueba la búsqueda del artículo, la selección y facturación del mismo.

Testing No Funcional

Se basa en cómo trabaja el sistema haciendo foco en los requerimientos no funcionales (foco en el “como”, no en el “que”). Son las pruebas más complejas, por su gran dependencia al entorno del sistema, por lo que debe ser lo más parecido posible al del cliente. Sin embargo, se tienen características que no pueden probarse, como el ancho de banda del internet, la seguridad o performance en una determinada situación. Y se pueden solucionar con las pruebas de aceptación. Incluye varios tipos de prueba como:

- Performance: Se ve el tiempo de respuesta (escenario esperado respecto a los tiempos de respuesta) y la concurrencia. Deben pasar esta prueba sí o sí.
- Carga: no solo mira performance, mira el comportamiento de los dispositivos de hardware (procesadores, discos, etc.) y de las comunicaciones.
- Stress: queremos forzar al sistema para que falle, se lo somete a condiciones más allá de las normales. Se ve el tiempo que hay que esperar para probar nuevamente el sistema y la robustez del sistema (tiempo de recuperación).
- Mantenimiento: para ver si el producto está en condiciones de evolucionar, se controla que haya documentación, manual de configuración, etc. Se observa la facilidad que existe para corregir un defecto.
- Usabilidad: que sea cómodo para el usuario.
- Portabilidad: se prueba en los distintos entornos acordados con el cliente.

- Fiabilidad: probamos que podemos depender del sistema. Resultados que se obtienen, seguridad física del software.
- De interfaz de usuario: suelen ser más complejas las GUI's que las interfaces de comandos.
- De configuración.

Test Driven Development – TDD

Este es un tipo de proceso en el que nos enfocamos en construir primero la prueba unitaria cuando tengo los requerimientos y luego codear el componente. Si pienso en las pruebas después tengo menos errores, el fundamento filosófico de esto es que si no tengo claro que tengo que hacer no puedo crear pruebas para eso.

Es una técnica de desarrollo del software que involucra dos prácticas:

1. **Test first development:** en esta técnica primero se escriben las pruebas unitarias referentes a la característica de producto a implementar. Definidas las pruebas unitarias, se piensa en la codificación necesaria para que las pruebas se ejecuten con éxito. Dado que las pruebas unitarias prueban unidades concretas de código, esta técnica obliga al desarrollador a modularizar su codificación haciendo que los métodos o clases a probar tengan una única responsabilidad. Además de pruebas unitarias, pueden incluirse en primera instancia pruebas de integración.
2. **Refactoring:** esta técnica consiste en reestructurar el código existente sin modificar el comportamiento que el mismo provee. Implica la mejora de aspectos no funcionales del software, cuyo objetivo es mejorar la claridad del código y reducir su complejidad, con el fin de hacerlo más mantenible.

Mejora continua de procesos con Kanban

Kanban

No es ni:

- Un proceso de desarrollo de software.
- Una metodología de administración de proyectos.

Definición

Kanban es un enfoque para la gestión de formas de trabajo (procesos) para obtener mejora continua.

Esto lo logra a través del principio de “empieza por donde estés”, es decir, no plantea introducir cambios revolucionarios, sino que introduce mejoras graduales a un proceso de desarrollo de software o a una metodología de administración de proyectos ya existente en una organización. Esto permite reducir la resistencia al cambio por parte de las personas, fomentando la evolución gradual de los procesos existentes.

Este enfoque surge con estudios de Toyota, para mejorar las técnicas de almacenamiento y tiempo de stockeo en supermercados.

Para su aplicación en el desarrollo de software, al igual que Lean fue necesaria una adaptación.

Kanban aprovecha los principios de Lean:

- Definiendo el **valor** desde la perspectiva del cliente.

- Limitando el trabajo en proceso WiP.
- Identificando y eliminando desperdicios.
- Identificando y eliminando las barreras en el flujo, es decir, todo lo que atrasaría el proceso: Relacionado con el principio de lograr entregar lo antes posible.
- Cultura de mejora continua.

Prácticas de Kanban

Visualización del trabajo

El método Kanban utiliza un mecanismo de señalización para hacer visible el trabajo que es requerido por el cliente. Para ello, divide el trabajo en piezas de trabajo (que pueden ser U.S., Features, bugs, temas, épicas, cambios, etc.) y las escribe en tarjetas señalizadoras (kan-bans) que serán ubicadas en tableros kanban.

Estas tarjetas permiten indicar cuando el trabajo se puede “pullear” para realizar un trabajo determinado, además de señalar en qué parte del proceso se encuentra.

El tablero kanban representa un sistema de flujo en el que las piezas de trabajo fluyen a través de las diversas etapas de un proceso, de izquierda a derecha. Cada etapa es representada por una columna del tablero, sobre las cuales se aplica la teoría de colas.

Se logra transparencia al hacer visible para todo el equipo el trabajo mediante un tablero kanban siempre disponible y a la vista.

Existen dos tipos de columnas:

- Producción: piezas sobre las que se está trabajando.
- Acumulación: piezas que están listas para pasar a la siguiente etapa (sistema de arrastre).

Limitar el WiP (Work in Progress)

Se deben asignar límites concretos a cuántas piezas pueden estar en progreso en cada etapa del flujo de trabajo (en cada columna), para evitar atascamientos o cuellos de botella.

Las políticas para limitar el WiP crean un sistema de arrastre (pull): el trabajo es “arrastrado” al sistema cuando otro de los trabajos es completado y queda capacidad disponible, en lugar de “empujar” estos trabajos al paso siguiente cuando hay nuevo trabajo demandado.

Tener demasiado trabajo no finalizado es un desperdicio de tiempo, de dinero y alarga los tiempos de entrega. Observar, limitar y optimizar la cantidad de trabajo en progreso es esencial para tener éxito con Kanban, consiguiendo mejorar la calidad y el tiempo de entrega de servicio y aumentar la tasa de entrega.

Gestionar el flujo de trabajo

El trabajo fluye sobre un tablero permanente, no existe el concepto de iteración, proceso o proyecto. El tablero puede ser compartido con otros proyectos y otros equipos, ya que se trabaja cada pieza de trabajo de forma individual. Lograr que el flujo sea continuo e ininterrumpido.

Al flujo que se observa en ese tablero, se lo debe analizar, identificando diferentes características del proceso de trabajo: cuellos de botella, recursos ociosos, tiempos de entrega, tiempos de espera, etc.

Hacer explícitas las políticas

Las políticas de proceso deben ser escasas, simples, estar bien definidas, visibles, deben aplicarse siempre, y tienen que ser fácilmente modificables. Es una buena práctica poder cambiar fácilmente las políticas, ya que si producen efectos contraproducentes para nuestro proceso o también se considera que no pueden aplicarse las mismas, deben poder cambiarse.

También es importante visualizar las políticas; por ejemplo, colocando resúmenes entre las columnas donde se describe lo que debe estar hecho antes de que una tarjeta se mueva de una columna a la siguiente. También se suele colocar el WIP de cada columna.

Las políticas de calidad o DoD deben estar definidas, publicadas y promovidas para lograr no sólo que se cumplan, si no buscar mejorar continuamente.

Mejorar y evolucionar

Kanban propone una cultura de mejora continua, donde no introduce cambios significativos en los procesos de desarrollo, sino que identifica ese proceso, lo visualiza (hacer explícito para todos los miembros) y propone mejoras sobre él, las cuales se van aplicando de forma gradual a lo largo del tiempo.

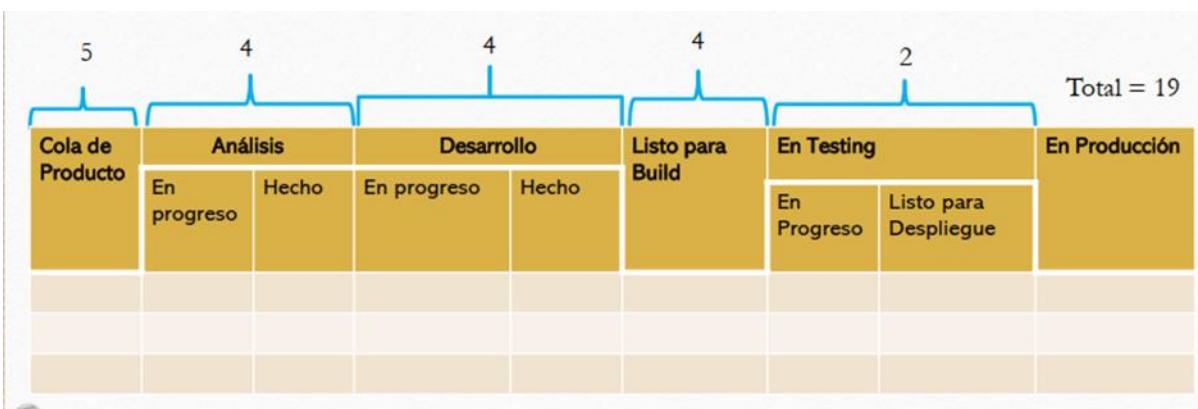
Círculo de retroalimentación

Son una parte esencial para cualquier proceso controlado que nos ayuda a realizar cambios evolutivos. Se debe definir con qué frecuencia se realizan las reuniones, donde depende en qué contexto se presentan ya que es importante para el resultado.

Si se realizan revisiones demasiadas frecuentes pueden obligar a cambiar cosas que no se vieron con los cambios anteriores, pero si no son demasiadas frecuentes, existe un bajo rendimiento durante mucho tiempo.

¿Cómo aplicar Kanban?

1. Empezar con lo que se tiene: entender el proceso de desarrollo actual que se está utilizando.
2. Identificar las unidades de trabajo a utilizar (US, CU, defectos).
3. Identificar clases de servicio: diferentes trabajos tienen distintas políticas para tratarlos (DoD). Por ejemplo: requerimientos, defectos, desarrollo y solicitudes.
4. Visualizar el flujo de trabajo diseñando un tablero representando el flujo de trabajo a través de columnas.
5. Definir políticas para cada clase de servicio identificada. Esto implica acordar los WiP para cada columna, asignar color a las tarjetas kan-ban, capacidad de trabajo destinada, fechas de entrega, etc.
6. Identificar cuellos de botella y resolverlos (asignando más recursos o ajustando el WiP donde corresponda).



Métricas de Kanban

Las métricas más representativas de Kanban son:

- Cycle Time.
- Lead Time.
- Touch Time.
- Eficiencia del Ciclo de Proceso.

Estas métricas están desarrolladas en el tema de métricas de la unidad 2 de este apunte.

Valores de Kanban

Los valores de Kanban se podrían resumir en una sola palabra, “respeto”. Sin embargo, es importante desgranar esto en una serie de nueve valores.

- **Transparencia:** La creencia de que compartir información abiertamente mejora el flujo de valor de negocio. Utilizar un lenguaje claro y directo es parte del valor.
- **Equilibrio:** El entendimiento de que los diferentes aspectos, puntos de vista y capacidades deben ser equilibrados para conseguir efectividad. Algunos aspectos (como demanda y capacidad) causarán colapso si no se encuentran equilibrados por un periodo prolongado.
- **Colaboración:** Trabajar juntos. El Método Kanban fue formulado para mejorar la manera en que las personas trabajan juntas, por ello, la colaboración está en su corazón.
- **Foco en el cliente:** Conociendo el objetivo para el sistema. Cada sistema kanban fluye a un punto de valor realizable — cuando los clientes reciben un elemento solicitado o servicio. Los clientes en este contexto son externos al servicio, pero pueden ser internos o externos a la organización como un todo. Los clientes y el valor que estos reciben es el foco natural en Kanban.
- **Flujo:** La realización de ese trabajo es el flujo de valor, tanto si es continuo como puntual. Ver el flujo es un punto de partida esencial en el uso de Kanban.
- **Liderazgo:** La habilidad de inspirar a otros a la acción a través del ejemplo, de las palabras y la reflexión. Muchas organizaciones tienen diferentes grados de jerarquía estructural, pero en Kanban, el liderazgo es necesario a todos los niveles para alcanzar la entrega de valor y la mejora.
- **Entendimiento:** Principalmente conocimiento de sí mismo (tanto individual como de la organización) para ir hacia adelante. Kanban es un método de mejora, por lo que conocer el punto de inicio es la base de todo.
- **Acuerdo:** El compromiso de avanzar juntos hacia los objetivos, respetando — y donde sea posible, acomodando — las diferencias de opinión o aproximaciones. Esto no es gestión por consenso sino un compromiso dinámico para mejorar.
- **Respeto:** Valorando, entendiendo y mostrando consideración por las personas. De manera apropiada al pie de esta lista se encuentra la base sobre la cual reposan el resto de los valores.

Principios directores

1. **Sostenibilidad:** relativo a encontrar un foco sostenible y un foco en la mejora.
2. **Orientación al servicio:** enfocado a conseguir rendimiento y satisfacción del cliente.
3. **Supervivencia:** relativa al mantenimiento de la competitividad y adaptabilidad.

Principios fundacionales

Hay seis principios fundamentales de Kanban, los cuales pueden ser divididos en dos grupos: los principios de gestión de cambio y los principios de entrega o despliegue de servicios (Fig 3)

Principios de gestión de cambio

Cada organización es una red de individuos, conectados psicológicamente y socio-lógicamente para resistir al cambio. Kanban reconoce estos aspectos humanos con *tres principios de gestión de cambios*.

1. Empezar con lo que estés haciendo ahora:
 - Entender los procesos actuales tal y como están siendo realizados en la actualidad, y
 - Respetar los roles actuales, las responsabilidades de cada persona y los puestos de trabajo.

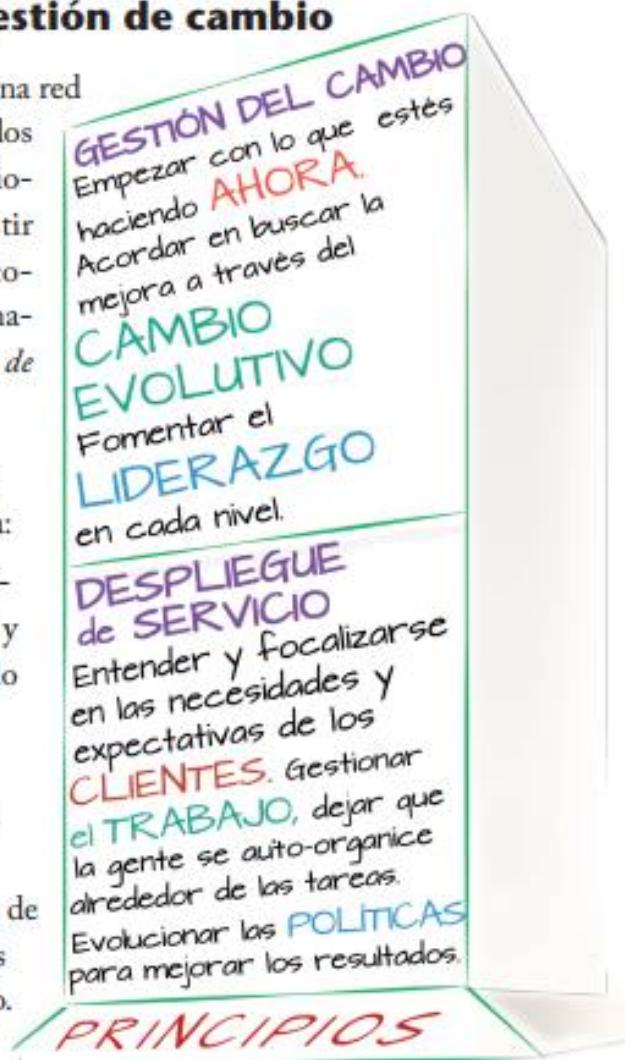


Figure 3 Principios de Kanban

2. Acordar en buscar la mejora a través del cambio evolutivo.
3. Fomentar el liderazgo en cada nivel de la organización — desde las contribuciones individuales de cada persona hasta las posiciones más senior de la organización.

Hay dos razones clave para que “empezar desde donde estés” sea una buena idea. La primera es que se minimiza la resistencia al cambio ya que respetamos las prácticas actuales y a las personas que las llevan a cabo. Esto es crucial para involucrar a todos en los retos y desafíos del futuro. La segunda es que los procesos actuales, junto con sus deficiencias obvias, contienen la sabiduría y la potencialidad de mejora que incluso las personas que trabajan con ellos no aprecian en su totalidad. Dado que el cambio es esencial, no hay que imponer soluciones desde diferentes contextos, sino buscar la mejora evolutiva en todos los niveles de la organización. A partir de las prácticas actuales hay que establecer los objetivos de mejora desde la situación actual a la situación deseada y evaluar las mejoras y como éstas nos van acercando hacia el objetivo.

Principios de despliegue de servicios

Las organizaciones, independientemente de su tamaño, son ecosistemas de servicios interdependientes. Kanban reconoce esto con tres *principios de despliegue de servicios*, aplicables no solo a un servicio, sino a todos ellos

1. Entender las necesidades y expectativas de tus clientes y focalizarse en ellas.
2. Gestionar el trabajo: dejar que la gente se auto-organice alrededor de las tareas.
3. Evolucionar las políticas para mejorar los resultados hacia el cliente y del negocio

Estos principios están muy alineados con el principio director de orientación a servicios y entrega de valor al cliente. Cuando el trabajo en sí mismo y el flujo de valor al cliente no es claramente visible, las organizaciones se enfocan en vez de eso en lo que *es* visible: la gente trabajando en el servicio. ¿Están siempre ocupados? ¿Tienen las suficientes habilidades? ¿Podrían trabajar más duro? El cliente y el valor al cliente reciben menos atención. Estos principios hacen hincapié en que el foco debe estar en los consumidores del servicio y en el valor que reciben del mismo.

Anexo

Diferencias entre Scrum y Kanban

Diferencia	SCRUM	KANBAN
Tiempo	Las iteraciones son de tiempo fijo.	El tiempo fijo es opcional. La cadencia puede variar. Pueden estar marcadas por la previsión de los eventos en lugar de tener un tiempo prefijado.
Compromiso	El equipo asume un compromiso de trabajo por iteración.	El compromiso es opcional.
Métrica por defecto	Para la planificación y mejora es la velocidad .	El por defecto es Lead Time (tiempo de entrega promedio)
Equipos	Multifuncionales	Multifuncionales o especializados.
Diagramas de seguimiento	Deben emplearse gráficos Burndown	No se prescriben diagramas de seguimiento.
Limitación WIP	Es indirecta y está marcada por el sprint.	Es directa para cada etapa de trabajo o el estado del trabajo.
Agregar trabajo.	No se puede agregar alcance una vez comenzado el sprint.	Siempre que haya capacidad disponible, se puede agregar trabajo.
Estimaciones.	Se deben realizar estimaciones.	La estimación es opcional.
Compartido entre equipos	Cada equipo tiene su sprint backlog.	Todos los equipos comparten la misma pizarra o tabla kanban.
Permanencia tablero	En cada sprint se limpia.	El tablero es persistente.
Roles	Se prescriben tres roles. PO, SM y Equipo.	No se prescriben roles.
Priorización	El product backlog debe estar priorizado.	La priorización es opcional.
Funcionalidades	Espera que las funcionalidades se dividan tal que se completen en un sprint.	No se prescribe el tamaño de la funcionalidad.

Kanban no tiene el concepto de iteración ni proyecto, porque el trabajo fluye a lo largo de las distintas etapas del proceso de manera continua. No se corta o para en ningún momento este proceso.

El tablero en Kanban es permanente, mientras que, en Scrum, el mismo se limpia al finalizar cada sprint, ya que contiene las columnas de “to-do”, “doing” y “done” propio del sprint que se está ejecutando.

Ser ágil y hacer ágil

Hacer ágil

Cuando implementamos métodos y prácticas ágiles estamos “haciendo” algo para lograr la agilidad. Esto nos ayuda a tener los eventos artefactos y procesos adecuados para entregar valor de forma continua con un buen nivel de calidad, gracias a esto podemos tener algunos beneficios como: adaptación a los cambios, mejorar la visibilidad, incrementar la productividad, mejorar la calidad y reducir los riesgos. Un buen programa de entrenamiento nos puede ayudar a alcanzar este estado.

Doing Agile (hacer agil) es practicar rituales y ceremonias ágiles. Las reuniones de pie, las demostraciones y revisiones quincenales, la planificación del sprint, los tableros Kanban y los puntos de la historia. Todos estos elementos son indicaciones claras de “Doing Agile”.

Pero simplemente realizar algunas prácticas que se reconocen como Agile no se traducen en un entorno de trabajo ágil, en equipos de alto rendimiento, en mejor calidad o en un tiempo de comercialización más rápido.

Ser ágil

Cuando además de implementar prácticas adoptamos una forma diferente de pensar estamos “siendo” alguien distinto, alguien que vive la agilidad y cuyo comportamiento está alineado a los valores y principios ágiles, esto nos da beneficios adicionales como: deleite del cliente, placer por el trabajo, compromiso, innovación, creatividad y aprendizaje continuo. Cuando hablamos de ser ágil a nivel organizacional nos referimos a empresas que adoptan una cultura que permite obtener estos beneficios mediante un cambio de mentalidad en todos los niveles de liderazgo, no solo de los equipos de trabajo.

Ser ágil (Being Agile) realmente funciona de manera diferente. Incluye una estructura de toma de decisiones drásticamente nueva, participación empresarial totalmente integrada, equipos persistentes autoorganizados y multidisciplinares.

Además, el enfoque es en el producto (no proyecto), usando [DevOps](#) y una cultura totalmente pensada en el aprendizaje continuo.

En definitiva, significa convertir esos rituales en hábitos diarios que impulsan la acción de mejora continua. Esto es más que hacer agilidad, ser ágil es lo que es.

A medida que la transformación se expande y madura, deja de hacer agilidad, ser ágil es en lo que se transforma. Hay varias prácticas que pueden emplearse para ayudar a las organizaciones a mejorar:

- Confía en tus equipos. Direccionalmente, el liderazgo senior establece los objetivos estratégicos, pero los equipos autoorganizados y auto empoderados (liderados por Products Owners autónomos y autorizados) determinarán los mejores enfoques, arquitecturas técnicas y todo el “cómo” a desarrollar, probar y lanzar. Esto significa que no hay más comités directivos. Los problemas se resuelven en el nivel más bajo posible (nivel de equipo).
- Practica la transparencia. Esto se aplica a todo en lo que trabaja el equipo. La transparencia debe ser un mantra para todos los equipos ágiles, así como para su liderazgo. Todos los entregables son trabajos en progreso, por lo tanto, deben estar abiertos para inspección en todo momento. La clave es que los Product Owners establezcan expectativas con el liderazgo superior sobre la naturaleza de esos entregables y cómo se entregan. La mejor medida de progreso es el incremento de trabajo disponible para los usuarios.

- Gestionar los indicadores de progreso de manera diferente. Dejamos de administrar por informes de estado y eliminamos los mandatos basados en fechas. Podemos seleccionar objetivos de lanzamiento de productos e hitos, pero no serán de alcance fijo; serán de naturaleza más temática sin características específicas completamente definidas. Los radiadores de información son increíblemente importantes. La transparencia es un principio básico ágil y, como tal, debe integrarse en cada transformación.

Para aquellos que son habitualmente ágiles, ver errores rápidamente es una prueba de que estás aprendiendo, mejorando constantemente y acercándote a tus clientes.

Aquellos que piensan en una mentalidad ágil ponen la confianza y la transparencia por encima de todo en su cultura. Entonces, mírate en el espejo y pregúntate si eres realmente ágil, o si simplemente estás haciendo posturero. Por qué puede ser que no te valga solo hacer agilidad, ser ágil es lo que tienes que ser.

Conclusión

En mi experiencia las mejores transiciones hacia la agilidad se logran cuando además de la adopción de prácticas ágiles también se adopta el mindset Ágil, esto definitivamente es mucho más complejo, ya que un entrenamiento no será suficiente, se necesita de un acompañamiento adecuado, de preferencia de un coach experimentado que pueda ayudar a lograr una transición cultural.

Hacer ágil es definitivamente distinto a ser ágil, y aunque ninguno de los dos es fácil no debemos perder de vista los objetivos que queremos alcanzar, si queremos el mayor de los beneficios debemos desafiar nuestras actuales formas de pensar y promover ese cambio alrededor de nosotros. Mi recomendación es solicitar apoyo de un experto, ya sea interno o externo, no solo en metodologías Ágiles sino también en el proceso de transformación hacia nuevos paradigmas.

Cuadro comparativo Gestión Tradicional vs. Gestión ágil

Aspecto a comparar	Gestión Tradicional	Gestión Ágil
Respuesta a cambios	Poco flexible	Altamente flexible
Incertidumbre en el desarrollo del proyecto	Baja → se elimina en etapa de requerimientos	Alta → se elimina cuando sea necesario
Tiempos de entrega de software útil	Tardío	Temprano
Retroalimentación del cliente para incluir nuevas necesidades o correcciones	Baja	Alta
Participaciones en estimaciones	Las hace el líder de proyecto	Las hace el equipo de desarrolladores
Conformación de equipo	Jerárquico, con roles definidos.	Autogestionado según fortalezas de cada miembro
Proceso de desarrollo	Definido	Empíricos

Ciclos de vida admitidos	Cualquiera	Iterativos e incrementales
Se estima	Recursos y tiempos necesarios para el desarrollo	Alcances a implementar en iteraciones (Tamaño)
Son constantes	Los requisitos identificados en etapa de requerimientos	El equipo, recursos y tiempo de trabajo.
Planificación de tareas	Estimadas con detalle al planificar el proyecto	Estimadas con detalle al planificar la iteración

Preguntas frecuentes de la ingeniería de software (Sommerville)

Pregunta	Respuesta
¿Qué es software?	Programas de cómputo y documentación asociada. Los productos de software se desarrollan para un cliente en particular o para un mercado en general.
¿Cuáles son los atributos del buen software?	El buen software debe entregar al usuario la funcionalidad y el desempeño requeridos, y debe ser sustentable, confiable y utilizable.
¿Qué es ingeniería de software?	La ingeniería de software es una disciplina de la ingeniería que se interesa por todos los aspectos de la producción de software.
¿Cuáles son las actividades fundamentales de la ingeniería de software?	Especificación, desarrollo, validación y evolución del software.
¿Cuáles son los principales retos que enfrenta la ingeniería de software?	Se enfrentan con una diversidad creciente, demandas por tiempos de distribución limitados y desarrollo de software confiable.

¿Cuáles son los mejores métodos y técnicas de la ingeniería de software?	Aun cuando todos los proyectos de software deben gestionarse y desarrollarse de manera profesional, existen diferentes técnicas que son adecuadas para distintos tipos de sistema. Por ejemplo, los juegos siempre deben diseñarse usando una serie de prototipos, mientras que los sistemas críticos de controles de seguridad requieren de una especificación completa y analizable para su desarrollo. Por lo tanto, no puede decirse que un método sea mejor que otro.
--	--

Cuadro comparativo Auditorías vs. Revisiones técnicas

Aspecto	Auditoría	Revisión
Para qué sirven	Para revelar qué se está haciendo en la realidad, contrastado con lo que se comprometió a hacer.	Encontrar errores en un artefacto lo antes posible, para evitar su propagación.
Roles	Auditor, auditado y gerente de calidad	Autor, moderador, anotador, lector, inspector
Beneficios	Opiniones independientes, identificar áreas de insatisfacción potencial para el cliente, asegurar que se cumplen expectativas, dar visibilidad a procesos de trabajo, etc.	Detección de errores, evitar propagación de defectos, reducción de retrabajo (costos), aumentar calidad de un artefacto, etc.
Etapas	<ol style="list-style-type: none"> 1. Planificación 2. Ejecución 3. Análisis y Reporte de resultados 4. Seguimiento 	<p>En revisiones informales no hay pasos a seguir.</p> <p>En revisiones formales:</p> <ol style="list-style-type: none"> 1. Planificación 2. Visión general 3. Preparación 4. Reunión de Inspección 5. Corrección 6. Seguimiento
Tipos	De proyecto, de calidad, de producto (física y funcional)	Formales e informales

Hallazgos	Buenas prácticas, observaciones y desviaciones	Errores
-----------	--	---------

Cuadro comparativo CMMI por Etapas y Continuo

Aspecto	Por etapas	Continuo
Características	Acreditan determinado nivel de madurez de una organización como un todo, dependiendo de qué tantas prácticas realicen, de acuerdo a lo especificado en el modelo CMMI.	Acreditan determinado nivel de madurez de un área de proceso de una organización, dependiendo de qué tantas prácticas realicen concernientes a esa área, de acuerdo a lo que la organización quiera especializar.
Ventajas	Permiten integrar las formas de trabajo de toda la organización. Garantiza cierto nivel de calidad en toda la organización, según el nivel que se acredite. Permite sentar bases en la organización y aumentar los niveles de calidad en toda la empresa.	Permite a una organización especializarse en un área de proceso, independientemente del resto de áreas. La organización elige qué prácticas realizar, de acuerdo al área en la que se especialice.
Desventajas	Se deben realizar ciertas actividades en áreas de proceso que quizás no sean relevantes para la empresa.	Pueden quedar áreas de proceso sin demasiada calidad, debido a que se centran los esfuerzos en unas pocas. La organización como un todo se sigue considerando inmadura, si es que no implementa las actividades que plantea CMMI nivel 1 en las áreas de proceso.
Diferencias	<ul style="list-style-type: none"> ● Niveles: 5 → 1 a 5 ● Niveles: indican madurez organizacional ● Definidos por un conjunto de áreas de proceso ● Provee una secuencia para mejorar determinadas áreas de proceso progresivamente 	<ul style="list-style-type: none"> ● Niveles: 6 → 0 a 5 ● Niveles: indican capacidad de un área de proceso ● Definidos por cada área de proceso ● Permite mejorar en un área de proceso que se deseé

Respuesta a pregunta de parcial

“Realice un análisis de los principios Lean y explique las prácticas que propone Kanban para aplicar en forma concreta los principios.”

- Eliminar desperdicios
 - Visualizar el trabajo (permite visualizar el flujo de trabajo y exponer aquellas actividades que no aportan valor y aquellas que generan tiempos de espera que “trablan” el flujo de trabajo)
 - Limitar el WIP (el trabajo a medias genera retrabajo, lo cual es un desperdicio. Al limitar el WIP, se evita ese trabajo a medias para concentrar esfuerzos)
- Amplificar el aprendizaje
 - Hacer explícitas las políticas (que todos los miembros tengan acceso a las políticas usadas para cada unidad de trabajo, actividades a realizar, información necesaria del dominio, etc.)
 - Evolución y mejora continua (no introducir cambios grandes, sino que visualizar el proceso de trabajo actual y aplicar mejoras en él)
- Embasar la integridad conceptual
- Postergar decisiones hasta el momento responsable
 - Limitar el WIP (particularmente el WIP de la columna backlog)
- Ver el todo
 - Visualizar el trabajo (a través del tablero permite que todos los miembros observen todo el trabajo que se hace, el sentido de cada una de las tareas y como estas conforman el flujo de trabajo como un todo que desencadena en la satisfacción de los clientes)
- Dar poder al equipo
 - Haciendo explícitas las políticas y conformando un sistema de trabajo pull, donde cada miembro toma el trabajo cuando esté en condiciones de realizarlo, y no se lo impone nadie.
- Mejorar colaborativamente
- Entregar lo antes posible
 - Gestionar el flujo de trabajo (permite ver de principio a fin los pasos que se deben seguir para lograr cumplir con las necesidades del cliente y hacer aquellos ajustes que sean necesarios para entregar el mayor valor posible, en el menor tiempo posible)