# Taller de implementación de metodologías ágiles

Guión de jornadas

# Escaleta de curso

# Bienvenida

## Introducción

El presente guión de jornadas corresponde al dictado del **Taller de implementación de metodologías ágiles** desarrollado por el equipo del **Laboratorio de Software** (ex Laboratorio de Testing y Aseguramiento de la Calidad) del centro **INTI Córdoba**.

En él encontraremos una serie de temas e informaciones que apoyan y complementan el dictado del curso de carácter práctico, enfocado a generar en quienes lo atiendan una experiencia real y práctica de ingeniería de software, orientada fuertemente hacia las líneas de las metodologías ágiles, considerando también un consistente sustento teórico.

Así, reforzando la orientación pragmática del curso y el equipo capacitador, se tiene como objetivo que los asistentes finalicen el curso con una experiencia de caso real derivada directamente de la industria, como también con un análisis y propuesta de implementación de metodología de trabajo acorde al contexto y entorno donde se desenvuelven; de modo que puedan impactar en sus organizaciones y procesos los beneficios de estas temáticas con resultados tangibles y de provecho.

## Consideraciones del material de jornadas

El formato de **guión** seleccionado para el material responde a una necesidad de dictado. Esto es, se establecen pautas, temáticas e informaciones de referencia pero el curso en desarrollo tomará los caminos que el grupo conformado para recibirlo y dictarlo consideren más oportunos y de criterio al momento.

Esto busca una participación activa y a modo de “coaching”, asegurando exponer realidades, acercar a los participantes y potenciar conocimientos a partir de esa apertura.

El (o los) caso(s) **práctico(s)** a tratarse será convenido por el equipo capacitador según requerimientos particulares de las jornadas y el grupo receptor.

## Sobre el curso

El presente curso está basado en material de cursos y charlas sobre desarrollo, testing y metodologías ágiles, como así también en la experiencia del equipo del Laboratorio de Software en diversas ramas de la industria del Software.

Presenta una introducción a las metodologías ágiles, sus conceptos y las diversas corrientes prácticas, que se sustentará luego con casos prácticos, herramientas y experiencia para llegar a realizar un planteo generalizado de cada caso particular presente

en el curso. Con el objetivo final de conocer de qué manera la metodología puede aplicarse a diversos contextos concretos.

# Taller de metodologías

Como bienvenida al **Taller de implementación de metodologías ágiles** se recorrerá un camino de conceptos y evolución histórica de las metodologías de desarrollo de software, enmarcados por la disciplina de la ingeniería de software y apoyados en herramientas y técnicas de uso actual en la industria y el medio, para acompañar el día con actividades que acerquen la práctica de las metodologías a los participantes del taller.

## Introducción

El desarrollo de Software es, en estos momentos de la historia, uno de los resultados de la evolución tecnológica principales. Esto se justifica claramente al comprender la revolución comunicacional y de tecnología de información que se presentó cuando los sistemas (software) comenzaron a cruzar horizontalmente todos los ámbitos de la vida, tanto industrial, comercial, académica como la social y cotidiana.

Esto implica una situación casi “omnipresente” del Software, o hasta de dependencia del mismo; como herramienta, como innovación, como medio para lograr objetivos y concretar proyectos.

Podemos si, entonces, notar una tendencia “feliz” en la situación del Software, pero la realidad es que desde sus incipientes y complejos inicios hasta el día de hoy, se encuentra en un camino señalado por crisis, pruebas y errores, recetas de todo tipo, y prácticas desde una formalidad exagerada hasta un libertinaje individual.

Fue la ingeniería, disciplina que desde cerca influyó y vigiló la evolución del Software, quien echó sus armas sobre este para comenzar a establecer una *Ingeniería de Software,* con la intención de permitir a los encargados de llevar adelante esta novedad, convertirse en profesionales certeros, ordenados, mensurables, alineados a procesos y tendientes a la calidad.

### A la concepción y desarrollo de un proyecto

Se entiende, o queremos entender, que la situación que origina una necesidad de Software en algún ámbito, dará inicio a un *proyecto*.

Un proyecto es el canal por el cual se planificará, organizará, ejecutará y seguirá un grupo de tareas y personas que conformarán un resultado esperado en un tiempo determinado.

Solemos entonces referirnos y enmarcarnos en estos a la hora de llevar adelante la satisfacción de esas necesidades de Software, sabiendo que tendrán sus particularidades por la materia que los orienta, como así también el contexto y el dominio que los contenga.

## Al desarrollo de Software

Es el desarrollo de Software una disciplina de creación de herramientas informáticas que involucra, en sus conceptos más básicos e iniciales, la intervención de personas con máquinas para instruirlas y que éstas funcionen bajo esas órdenes.

Podemos definir al Software como *la interfaz y la lógica abstracta, computacional, que vincula al ser humano con tecnología tangible y otros softwares existentes, dando la posibilidad de interactuar y retro-alimentarse con estos.*

Dejando de lado la discusión sobre la bisagra entre ciencia y ficción, esta actividad o disciplina fue conformándose y moldeándose de diversas maneras, con distintos objetivos y fuentes de inspiración.

En ciertos casos se ha llegado a una madurez de proceso, balanceando la teoría y la práctica, la ciencia y el arte, que presenta una apuesta seria a desafiar las crisis por las que el Software ha venido tropezando y haciéndose a sí mismo.

## A la ingeniería de Software

La ingeniería de Software es una de las encargadas de dar dirección a este crecimiento, sino la más importante, en sentar bases, métodos, técnicas, estándares y prácticas que sirvan de experiencia, de interfaz comunicacional, de eje estructural a una disciplina global, en auge y de importancia vital como lo es la automatización de la información y la tecnología.

Entonces si, la ingeniería de Software busca de alguna manera formalizar el proceso, la concreción de proyectos Software, el desarrollo mismo de Software. A través de una planificación y organización, que tendrá en cuenta variables técnicas, de criterio, de presupuesto, psicológicas, de calidad, seguridad y negocio entre otras tantas.

La ingeniería de Software buscará, mediante el ingenio, la ciencia, la experiencia y la práctica fomentar herramientas y actividades que permitan resolver los problemas planteados para el Software.

Con estos objetivos es que, de alguna manera, en algún momento... se empezaron a gestar y difundir las **Metodologías ágiles** para el desarrollo de Software.

# Metodologías ágiles

Para definir acabadamente y de una manera clara a las metodologías ágiles, es recomendable hacer un repaso histórico de la evolución de las metodologías y procesos de desarrollo de Software, ya que éstas son resultado o respuesta del proceso evolutivo.

## Historia de las metodologías de desarrollo de Software

A lo largo de la historia del desarrollo tecnológico, y en particular del software, se han generado diversas maneras de llegar a la concreción de proyectos.

Mientras se diseminaba la novedad de la informática y exigía cada vez mayores esfuerzos de trabajo por su crecimiento e inserción en el mundo, aparecieron formalizaciones en las maneras de producir o crear software para enfrentar las demandas y problemas.

Así hacen aparición las metodologías o procesos de desarrollo de Software, buscando organizar y mejorar las prácticas y los resultados.

Si bien estos procesos han dado una significativa mejora y un marco dentro del cual surgieron conceptos y pudieron adoptarse otros, el desarrollo de software sigue teniendo sus matices, con procesos y herramientas diversos que continúan también un camino propio.

La evolución que tuvieron estuvo marcada por diferentes cambios de foco, algo que fue de alguna manera guiando las nuevas tendencias y estableciendo los nuevos procesos.

Se inició intentando acompañar la evolución tecnológica mientras se interpretaba la novedad del Software, luego se dio una tendencia a alimentarse de otras disciplinas relacionadas que pudieran aportar modelos y prácticas a problemas de gran magnitud y complejidad.

La división del problema fue establecida, se identificaron áreas y tareas específicas ineludibles a la hora de dar vida a un Software (al menos de una magnitud considerable), luego se pulieron esas técnicas y se propuso un paralelismo en actividades que pudiera reducir el acarreamiento de problemas y errores.

Frente a eso surgió la noción de iteración, acompañada de un gran trabajo de documentación y definición de herramientas que facilitaron la comunicación entre el equipo y con los clientes.

Finalmente, al evaluar las falencias del último enfoque exitoso surgieron como respuesta las metodologías ágiles, quienes son hoy “la estrella” en lo que a proceso de trabajo y desarrollo de software se refiere, sobre todo cuando estudiamos los usos y costumbres de empresas e industrias pujantes.

## Modelo en cascada

El modelo o proceso en cascada define una serie de fases a seguir que conforman las tareas necesarias para dar vida a un software. Estas fases son independientes en la teoría y para comenzar con una debe finalizarse la anterior.

Las fases son:

1. Especificación de requisitos y análisis
2. Diseño y arquitectura del software

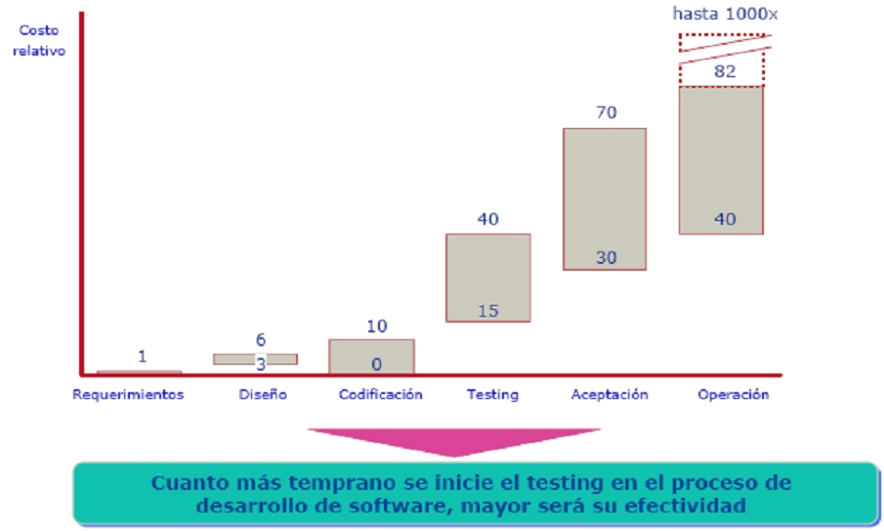
3. Construcción o implementación
4. Testing
5. Mantenimiento

Este planteo implica que no se comenzarán las tareas de prueba hasta no tener completo el producto en su totalidad. A su vez, no se comenzará a construir el software hasta que el análisis haya sido “exhaustivo”, y así.

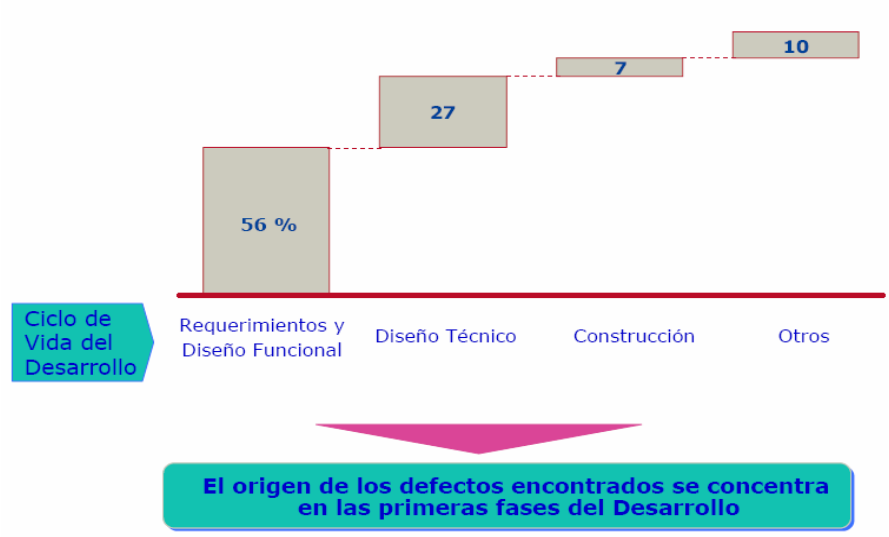
El problema aquí recae en los costos y complicaciones que genera la metodología. Si encontramos un error en la fase de codificación y es realmente un error de especificación de requisitos, eso implica que se han surcado todas las fases arrastrando un error y el único momento de encontrarlo y repararlos es sobre la etapa final de testing, generando una carga de re-trabajo más que considerable. Caso similar sería el de las modificaciones que pueden llegar a surgir en la especificación, luego de que esta etapa haya sido cerrada, dando como resultado avances sin validaciones y pérdidas de tiempo y dinero.

El problema central en esta metodología, como se expresa, es el **costo**. Hablamos de *costo relativo* refiriéndonos a lo que necesitará gastarse en tiempo, dinero o esfuerzo para reparar o corregir una falla según el momento en el que se ha detectado.

Estas relaciones alcanzan una evolución exponencial a lo largo del ciclo del proceso en cascada:




Es interesante también saber que las principales fuentes de generación de defectos se encuentran en las etapas tempranas del proceso, antes que en las etapas finales. Para verlo gráficamente:




Por último, vemos un interesante efecto que suele generarse dentro del proceso, que es el de amplificación de los defectos. Esta amplificación se da por una naturaleza propia del arrastre de los errores a medida que se avanza en el tiempo y de la aparición de nuevos propios de cada fase particular.

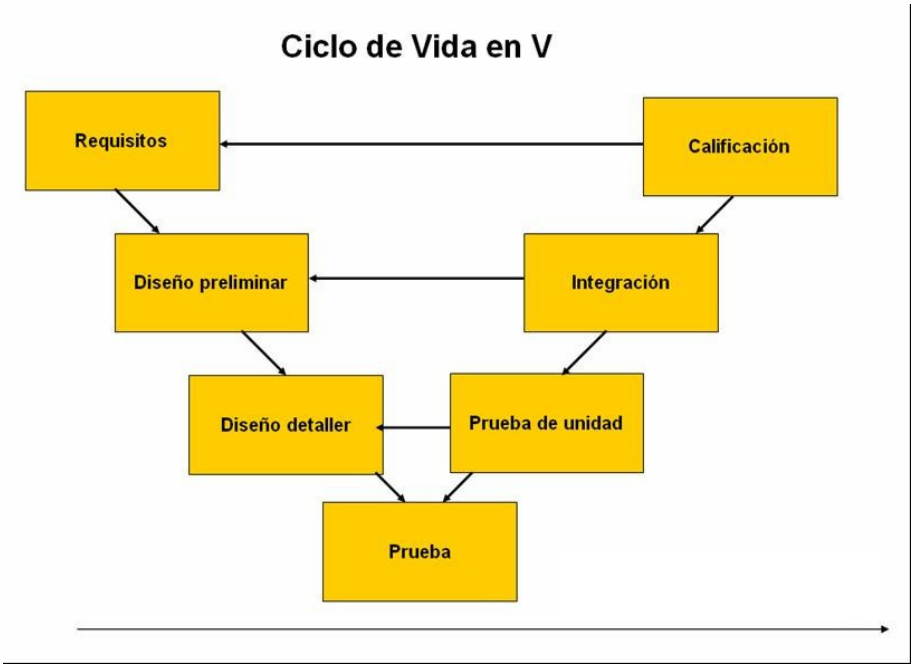
## Modelo en V

El desarrollo en V implica una descripción de actividades y resultados que deben producirse durante el desarrollo del producto, identificando un lado “izquierdo” de la V y un lado “derecho”. El lado izquierdo representa la descomposición de las necesidades, y la creación de las especificaciones del sistema. El lado derecho de la **V** representa la integración de las piezas y su verificación.

Se utiliza también la **V** para dar significado de «Verificación y validación». Existen muchas similitudes con el modelo en cascada clásico por su rigidez y gran cantidad de iteraciones, pero se diferencia ampliamente en que este modelo introduce el uso del principio de **testing temprano.**

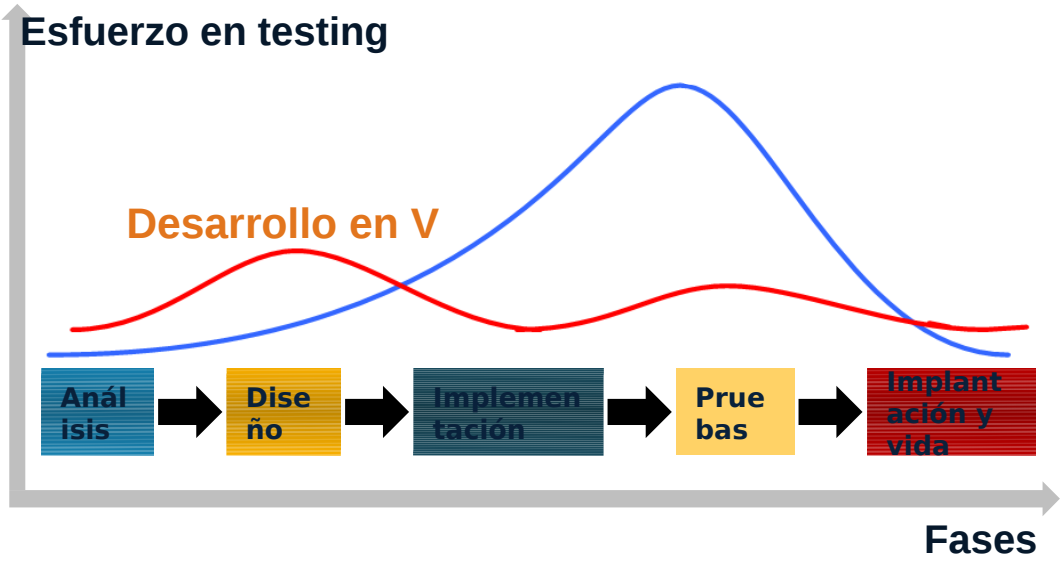
Posee una facilidad práctica para entenderlo, donde existen niveles que muestran la conexión entre las actividades de desarrollo y su correspondiente de testing, o que se encargaría de cubrir en cierta forma lo realizado.

Se va probando en paralelo a medida que se avanza en la rama izquierda, haciendo que la rama derecha vaya ejecutando las pruebas variando sus **niveles** según la fase en la que nos encontremos:




Ventajas principales:

- No es necesario esperar hasta que las fases del producto estén completas para iniciar el diseño de las pruebas
- Esto permite realizar correcciones de defectos antes de que sean trasladados al siguiente nivel



## Modelos iterativos

El desarrollo iterativo y creciente (o incremental) es un proceso de desarrollo de software, creado en respuesta a las debilidades del modelo tradicional de cascada y en V.

La idea principal detrás de mejoramiento iterativo es desarrollar un sistema de programas de manera incremental, permitiéndole al desarrollador sacar ventaja de lo que se ha aprendido a lo largo del desarrollo anterior, incrementando, versiones entregables del sistema. El aprendizaje viene de dos vertientes: el desarrollo del sistema, y su uso (mientras sea posible). Los pasos claves en el proceso son comenzar con una implementación simple de los requerimientos del sistema, e iterativamente mejorar la secuencia evolutiva de versiones hasta que el sistema completo esté implementado. En cada iteración, se realizan cambios en el diseño y se agregan nuevas funcionalidades y capacidades al sistema.

### Proceso unificado

El RUP es el Proceso Unificado de Rational, creado por la empresa Rational Software adquirida por IBM, que junto con el lenguaje de modelado UML constituye la metodología estándar más usada para el análisis, diseño, implementación y documentación de sistemas orientados a objetos, al menos durante fines de los años 90 y principios del 2000.

Se basa en un conjunto de metodologías adaptables al contexto y necesidades de cada organización, teniendo seis principios clave:

1. Adaptar el proceso
2. Equilibrar prioridades
3. Demostrar valor iterativamente
4. Colaboración entre equipos
5. Elevar el nivel de abstracción
6. Enfocarse en la calidad

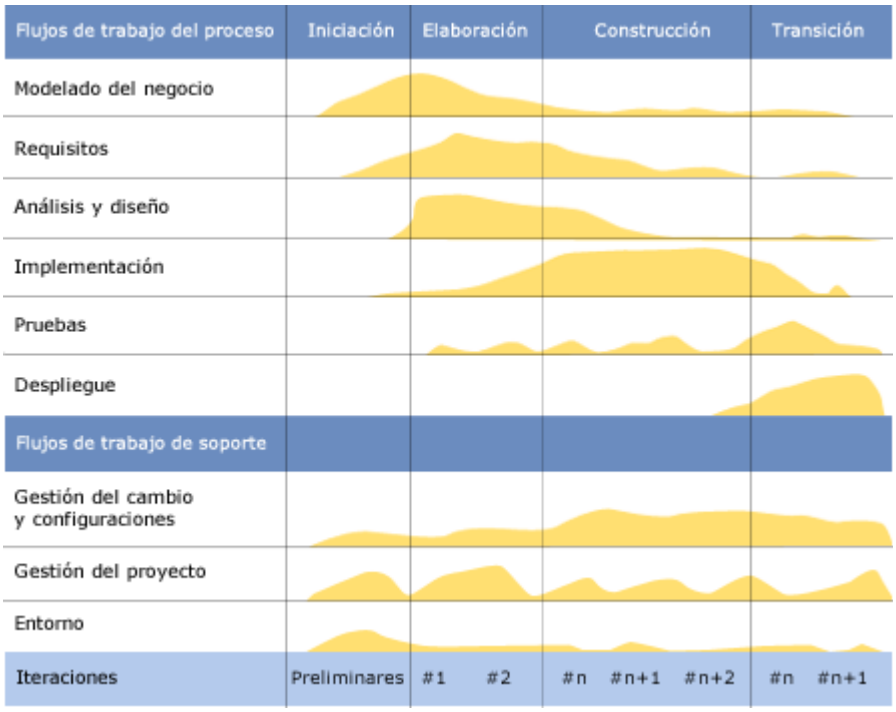
Es una actividad llevada a cabo a través de todo el ciclo iterativo de desarrollo donde cada iteración tiene una misión o meta diferente, tiene un carácter exploratorio, porque sus especificaciones tienden a cambiar con frecuencia.

Toma como base de pruebas no sólo las especificaciones, sino una colección de fuentes diversas, incluso no documentadas, pero debe evitar producir más documentación de la estrictamente necesaria.

El plan de prueba y un plan detallado de esfuerzo de prueba debería ser todo lo producido antes de la ejecución de pruebas.

Su ciclo de vida identifica fases pero no las limita a una dependencia estricta del resto, pudiendo dar inicio, por ejemplo, al testing en un momento temprano.

Paralelamente cada fase se va desarrollando e iterando sobre si misma, alimentando el avance y las tareas de las otras fases. Lo vemos gráficamente:




Existen 3 conceptos claves que dominan la metodología:

1. Dirigido por casos de uso
2. Centrado en la arquitectura
3. Iterativo e incremental

En lo referente a dirigido por los casos de uso, significa que los requerimientos están enfocados a dar valor al cliente y que el proceso debe garantizar que todo el desarrollo, pruebas, planificación, documentación etc, está orientado a cubrir estas expectativas del cliente y asegurar que los requerimientos de valor se ponen en producción.

En lo referente a centrado en arquitectura, significa que hay un énfasis a diseñar una arquitectura de calidad, y es la arquitectura también la que guía la forma cómo se debe planear y hacer el desarrollo.

En lo referente a iterativo e incremental, significa que el proyecto se divide en varios ciclos de vida (llamadas iteraciones) que deben dar como resultado un ejecutable. Por cada una de las iteraciones se va agregando requerimientos y sobre todo valor al cliente; por este motivo es incremental.

## Motivos que dan surgimiento a las ágiles

El modelo del proceso unificado se extendió de una manera global, al menos en lo que respecta a Software. Su gran producción

bibliográfica y el armado de la teoría como un producto, junto con la creación de herramientas y lenguajes propios, hizo que ésta fuera una de las metodologías más populares, colándose en la gran mayoría de las currículas de las carreras de grado universitario en el sector computación, sistemas, software e informática.

Si bien representó un gran avance desde el punto de vista de lo iterativo y de la comunicación, los volúmenes de documentación y de gestión que necesitaban los procesos que se desprendían de ella hicieron que el desarrollo de Software se convirtiera en algo complejo de abarcar, teniendo que pensar en una serie de pasos burocráticos siempre.

De esta manera, la crisis del Software no estaba saldada... la experiencia real evidenciaba drásticos problemas en la industria y la incapacidad de responder a las necesidades crecía, teniendo también en consideración que las mismas necesidades estaban exponencialmente subiendo sin control.

No había una respuesta en estos procesos, no se vislumbraba una al menos. Todas las etapas tenían sus equivalentes de documentación, era prácticamente todo necesario, se debía cumplir una estructura rígida, formal y la devolución de software funcional era algo a pensar a largo plazo si se hablaba de un proyecto importante. En los casos donde se podía reducir el volumen de documentación la ganancia tampoco era la esperada, ya que el proceso en si exigía un mínimo y la respuesta a errores era devastadora... El cliente podía validar el error generalmente al momento de probar el software, lo cual se daba muy adelante en el tiempo y la corrección implicaba un trabajo hacia atrás muy costoso.

## Manifiesto ágil

El 17 de febrero de 2001 diecisiete críticos de los modelos de mejora del desarrollo de software basados en procesos, convocados por Kent Beck, quien había publicado un par de años antes *Extreme Programming Explained*, libro en el que exponía una nueva metodología denominada Extreme Programming, se reunieron en Snowbird, Utah para tratar sobre técnicas y procesos para desarrollar software. En la reunión se acuñó el término “Métodos Ágiles” para definir a los métodos que estaban surgiendo como alternativa a las metodologías formales a las que consideraban excesivamente “pesadas” y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas previas al desarrollo.

Los integrantes de la reunión resumieron los principios sobre los que se basan los métodos alternativos en cuatro postulados, lo que ha quedado denominado como Manifiesto Ágil.

El manifiesto expresa:

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

- *A los **individuos y su interacción**, por encima de los*

*procesos y las herramientas.*

*• El **software que funciona**, por encima de la documentación exhaustiva.*

*• La **colaboración con el cliente**, por encima de la negociación contractual.*

*• La **respuesta al cambio**, por encima del seguimiento de un plan.*

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

# Conceptos principales

## Iteración

Iteraciones en el contexto de un proyecto se refieren a la técnica de desarrollar y entregar componentes incrementales de funcionalidades de un negocio. Una iteración resulta en uno o más paquetes atómicos y completos del trabajo del proyecto que pueda realizar alguna función tangible del negocio. Múltiples iteraciones contribuyen a crear un producto completamente integrado. A esto se lo compara comúnmente con el enfoque de desarrollo en cascada.

El proceso en sí mismo consiste de:

- Etapa de inicialización
- Etapa de iteración
- Lista de control de proyecto

## Historias de usuario

Una historia de usuario es una representación de un requisito de software escrito en una o dos frases utilizando el lenguaje común del usuario. Las historias de usuario son utilizadas para la especificación de requisitos (acompañadas de las discusiones con los usuarios y las pruebas de validación). Cada historia de usuario debe ser limitada, esta debería poderse escribir sobre una nota adhesiva pequeña. Existen distintos enfoques de quién escribe las historias de usuario y qué tratamiento se les da a ellas, pero generalmente recae en trabajo conjunto de los analistas de negocio y el cliente o dueño de producto.

Las historias de usuario son una forma rápida de administrar los requisitos de los usuarios sin tener que elaborar gran cantidad de documentos formales y sin requerir de mucho tiempo para administrarlos, también permiten responder rápidamente a los requisitos cambiantes.

### Características

Las historias de usuario deben ser:

- Independientes unas de otras: De ser necesario, combinar las historias dependientes o buscar otra forma de dividir las

historias de manera que resulten independientes.

- Negociables: La historia en si misma no es lo suficientemente explícita como para considerarse un contrato, la discusión con los usuarios debe permitir esclarecer su alcance y éste debe dejarse explícito bajo la forma de pruebas de validación.
- Valoradas por los clientes o usuarios: Los intereses de los clientes y de los usuarios no siempre coinciden, pero en todo caso, cada historia debe ser importante para alguno de ellos más que para el desarrollador.
- Estimables: Un resultado de la discusión de una historia de usuario es la estimación del tiempo que tomará completarla. Esto permite estimar el tiempo total del proyecto.
- Pequeñas: Las historias muy largas son difíciles de estimar e imponen restricciones sobre la planificación de un desarrollo iterativo. Generalmente se recomienda la consolidación de historias muy cortas en una sola historia.
- Verificables: Las historias de usuario cubren requerimientos funcionales, por lo que generalmente son verificables. Cuando sea posible, la verificación debe automatizarse, de manera que pueda ser verificada en cada entrega del proyecto.

## Uso

Las historias de usuario conforman la parte central de muchas metodologías de desarrollo ágil, tales como XP; Estas definen lo que se debe construir en el proyecto de software, tienen una prioridad asociada definida por el cliente de manera de indicar cuales son las más importantes para el resultado final, serán divididas en tareas y su tiempo será estimado por los desarrolladores. Generalmente se espera que la estimación de tiempo de cada historia de usuario se sitúe entre unas 10 horas y un par de semanas. Estimaciones mayores a dos semanas son indicativo de que la historia es muy compleja y debe ser dividida en varias historias.

Al momento de implementar las historias, los desarrolladores deben tener la posibilidad de discutirlas con los clientes. El estilo sucinto de las historias podría dificultar su interpretación, podría requerir conocimientos de base sobre el modelo o podría haber cambiado desde que fue escrita.

Cada historia de usuario debe tener en algún momento pruebas de validación asociadas, lo que permitirá al desarrollador, y más tarde al cliente, verificar si la historia ha sido completada. Como no se dispone de una formulación de requisitos precisa, la ausencia de pruebas de validación concertadas abre la posibilidad de discusiones largas y no constructivas al momento de la entrega del producto.

Si bien el estilo puede ser libre, la historia de usuario debe responder a tres preguntas: ¿Quién se beneficia?, ¿qué se quiere? y ¿cuál es el beneficio? Por ello, algunos autores recomiendan redactar las historias de usuario según el formato:

**Como (rol) quiero (algo) para poder (beneficio)**

### Beneficios

- Al ser muy corta, ésta representa requisitos del modelo de negocio que pueden implementarse rápidamente (días o semanas)
- Necesitan poco mantenimiento
- Mantienen una relación cercana con el cliente
- Permite dividir los proyectos en pequeñas entregas
- Permite estimar fácilmente el esfuerzo de desarrollo
- Es ideal para proyectos con requisitos volátiles o no muy claros

### Limitaciones

- Sin pruebas de validación pueden quedar abiertas a distintas interpretaciones haciendo difícil utilizarlas como base para un contrato
- Se requiere un contacto permanente con el cliente durante el proyecto lo cual puede ser difícil o costoso
- Pueden resultar difíciles las pruebas de usuario, que sólo se ven en las metodologías SCRUM para escalar a proyectos grandes
- Requiere desarrolladores muy competentes

## Estimación

La estimación toma en las metodologías ágiles una dimensión e importancia principal.

Se trata de comenzar a predecir cuánto tiempo tomará desarrollar cierta tarea, y así también puede llegarse a tener una idea aproximada de la duración que tendrá un proyecto.

Al partir de historias de usuario para realizar estimaciones, o tareas derivadas de estas, se aplican diversas técnicas de estimación según las características de cada implementación de metodología ágil y según la cultura de la organización o grupo.

## Reuniones

Otro de los conceptos centrales en metodologías ágiles es lo relacionado a comunicación y documentación.

Aquí debe entenderse y desmitificarse la idea de que en el uso de Metodologías Ágiles no existe documentación formal.

La tendencia de estos procesos es a prevalecer el encuentro cara a cara, la comunicación directa y abierta, sin generar complicaciones y llevando a documento lo estrictamente necesario, de manera de asegurar una posibilidad de mantener eso y que sea útil para comunicar decisiones o cuestiones técnicas.

De esa manera es que también se definen una serie de

reuniones específicas cada una para establecer parámetros de documentación, para resolver dudas en distintos planos y para dar una mirada general de las tareas a todo el grupo, sobre le cual también se centran las metodologías considerándolo un equipo.

### Impacto en la ingeniería de Software

Las metodologías ágiles han concretado un espacio fuerte, consolidado y que se ha extendido a la totalidad de organizaciones reconocidas que producen software y tecnología de información.

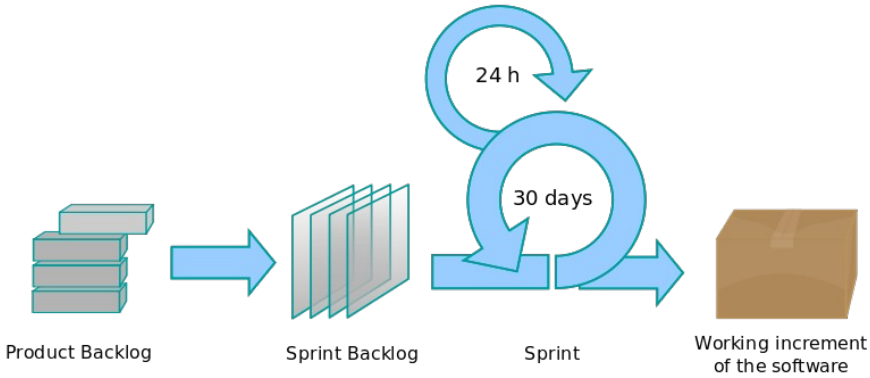
De esa misma manera han dado un cambio radical en la dirección de las líneas de la ingeniería de Software, a nivel teórico y práctico, permitiendo atacar la crisis del Software desde el punto de vista de la rápida entrega de valor y la respuesta eficiente al cambio, teniendo en cuenta que este es un factor deseable y constante en un proceso de desarrollo de Software.

## Implementaciones de metodologías ágiles

### Scrum

Usualmente suele definirse a scrum como un marco de trabajo para la gestión y desarrollo de software basada en un proceso iterativo e incremental utilizado comúnmente en entornos basados en el desarrollo ágil de software.

SCRUM es un modelo de referencia que define un conjunto de prácticas y roles, y que puede tomarse como punto de partida para definir el proceso de desarrollo que se ejecutará durante un proyecto.




### Roles

Los roles principales en Scrum son:

### Product Owner (Dueño de producto)

El *Product Owner* representa la voz del cliente. Se asegura de que el equipo Scrum trabaje de forma adecuada desde la perspectiva del negocio. El Product Owner escribe historias de usuario, las prioriza, y las coloca en el Product Backlog.

### Scrum Master (Facilitador)

El *Scrum* es facilitado por un *ScrumMaster*, cuyo trabajo primario es eliminar los obstáculos que impiden que el equipo alcance el objetivo del sprint. El *ScrumMaster* no es el líder del equipo (porque ellos se auto-organizan), sino que actúa como una protección entre el equipo y cualquier influencia que le distraiga. El ScrumMaster se asegura de que el proceso Scrum se utiliza como es debido. El ScrumMaster es el que hace que las reglas se cumplan.

### Equipo de desarrollo

El equipo tiene la responsabilidad de entregar el producto. Un pequeño equipo de 3 a 9 personas con las habilidades transversales necesarias para realizar el trabajo (análisis, diseño, desarrollo, pruebas, documentación, etc).

### Stakeholders (Clientes, proveedores, vendedores, etc)

Se refiere a la gente que hace posible el proyecto y para quienes el proyecto producirá el beneficio acordado que justifica su producción. Sólo participan directamente durante las revisiones del sprint.

### Administradores (Managers)

Es la gente que establece el ambiente para el desarrollo del producto.

## Reuniones

### Stand-up Meeting (Reunión de pie) o Reunión diaria

Cada día de un sprint, se realiza la reunión sobre el estado de un proyecto. Esto se llama *daily standup* o *Stand-up meeting*. El scrum tiene unas guías específicas:

- La reunión comienza puntualmente a su hora.
- Todos son bienvenidos, pero sólo los involucrados en el proyecto pueden hablar.
- La reunión tiene una duración fija de 15 minutos, de forma independiente del tamaño del equipo.
- La reunión debe ocurrir en la misma ubicación y a la misma hora todos los días.

Durante la reunión, cada miembro del equipo contesta a tres

preguntas:

- ¿Qué has hecho desde ayer?
- ¿Qué es lo que harás hasta la reunión de mañana?
- ¿Has tenido algún problema que te haya impedido alcanzar tu objetivo? (Es el papel del ScrumMaster recordar estos impedimentos).

### Scrum de Scrum

Cada día normalmente después del “Stand-up meeting”:

- Estas reuniones permiten a los grupos de equipos discutir su trabajo, enfocándose especialmente en áreas de solapamiento e integración.
- Asiste una persona asignada por cada equipo.

La agenda será la misma que la del Daily Scrum, añadiendo además las siguientes cuatro preguntas:

- ¿Qué ha hecho tu equipo desde nuestra última reunión?
- ¿Qué hará tu equipo antes que nos volvamos a reunir?
- ¿Hay algo que demora o estorba a tu equipo?
- ¿Estás a punto de poner algo en el camino del otro equipo?

### Reunión de planificación del Sprint

Al inicio del ciclo Sprint (cada 15 o 30 días), una “Reunión de Planificación del Sprint” se lleva a cabo.

- Seleccionar qué trabajo se hará
- Preparar, con el equipo completo, el Sprint Backlog que detalla el tiempo que tomará hacer el trabajo.
- Identificar y comunicar cuánto del trabajo es probable que se realice durante el actual Sprint
- Ocho horas como límite

Al final del ciclo Sprint, dos reuniones se llevaran a cabo: la “Reunión de Revisión del Sprint” y la “Retrospectiva del Sprint”

### Reunión de revisión del Sprint

Revisar el trabajo que fue completado y no completado

- Presentar el trabajo completado a los interesados (alias “demo”)
- El trabajo incompleto no puede ser demostrado
- Cuatro horas como límite

### Retrospectiva del Sprint

Después de cada sprint, se lleva a cabo una retrospectiva del sprint, en la cual todos los miembros del equipo dejan sus

impresiones sobre el sprint recién superado. El propósito de la retrospectiva es realizar una mejora continua del proceso. Esta reunión tiene un tiempo fijo de cuatro horas.

## Sprint

El Sprint es el período en el cual se lleva a cabo el trabajo en sí. Es recomendado que la duración de los sprints sea constante y definida por el equipo con base en su propia experiencia. Se puede comenzar con una duración de sprint en particular (2 o 3 semanas) e ir ajustándolo con base en el ritmo del equipo, aunque sin relajarlo demasiado. Al final de cada sprint, el equipo deberá presentar los avances logrados, y el resultado obtenido es un producto potencialmente entregable al cliente. Asimismo, se recomienda no agregar objetivos al sprint o *sprint backlog* a menos que la falta de estos objetivos amenace al éxito del proyecto. La constancia permite la concentración y mejora la productividad del equipo de trabajo.

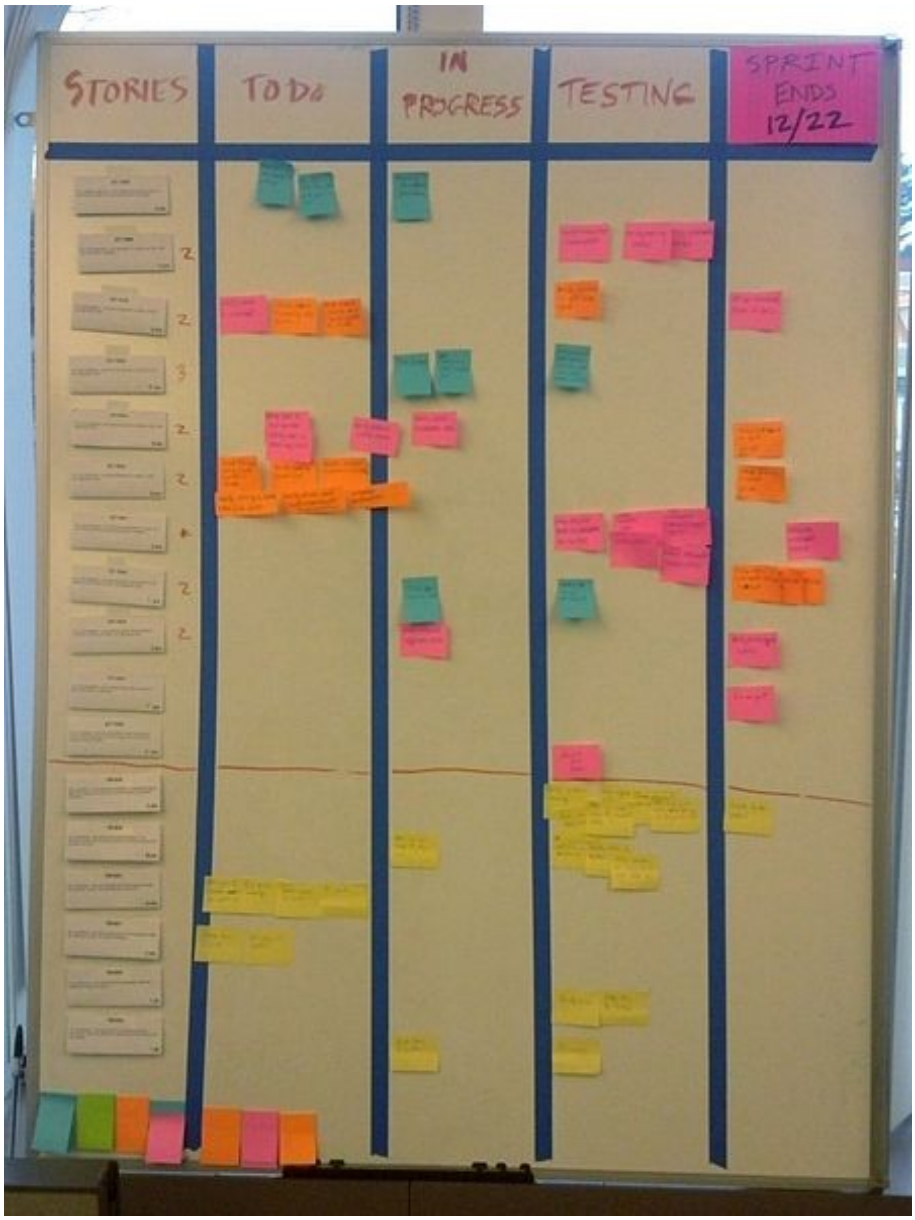
## Documentos

### Product Backlog

El *product backlog* es un documento de alto nivel para todo el proyecto. Contiene descripciones genéricas de todos los requerimientos, funcionalidades deseables, etc. priorizadas según su retorno sobre la inversión (ROI) . Es el *qué* va a ser construido. Es abierto y solo puede ser modificado por el *product owner*. Contiene estimaciones realizadas a grandes rasgos, tanto del valor para el negocio, como del esfuerzo de desarrollo requerido. Esta estimación ayuda al *product owner* a ajustar la línea temporal y, de manera limitada, la prioridad de las diferentes tareas. Por ejemplo, si dos características tienen el mismo valor de negocio la que requiera menor tiempo de desarrollo tendrá probablemente más prioridad, debido a que su ROI será más alto.

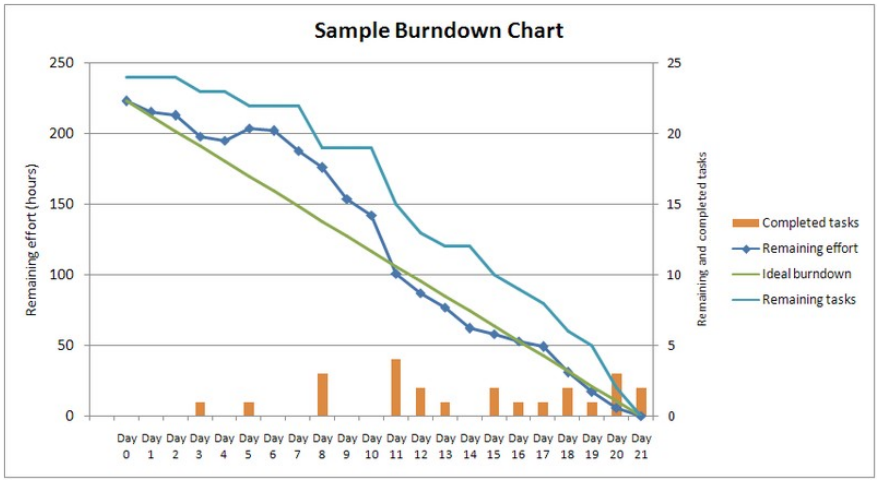
### Sprint Backlog

El sprint backlog es un documento detallado donde se describe el *cómo* el equipo va a implementar los requisitos durante el siguiente sprint. Las tareas se dividen en *puntos historia (story points)* con ninguna tarea de duración superior al total del Sprint. Si se considera que una tarea es mayor que la duración del Sprint, deberá ser dividida en otras menores.




## Burn down chart

La *burn down chart* es una gráfica mostrada públicamente que mide la cantidad de requisitos en el Backlog del proyecto pendientes al comienzo de cada Sprint. Dibujando una línea que conecte los puntos de todos los Sprints completados, podremos ver el progreso del proyecto. Lo normal es que esta línea sea descendente (en casos en que todo va bien en el sentido de que los requisitos están bien definidos desde el principio y no varían nunca) hasta llegar al eje horizontal, momento en el cual el proyecto se ha terminado (no hay más requisitos pendientes de ser completados en el Backlog). Si durante el proceso se añaden nuevos requisitos la recta tendrá pendiente ascendente en determinados segmentos, y si se modifican algunos requisitos la pendiente variará o incluso valdrá cero en algunos tramos.




## Estimación

### Punto de historia

Un punto de historia es una medida arbitraria utilizada por los equipos Scrum. Se usa para medir el esfuerzo requerido para implementar una historia de usuario.

En términos simples, es un número que le dice al equipo qué tan difícil es la historia (de usuario). “Difícil” puede estar relacionado a la complejidad, a la incertidumbre y al esfuerzo.

Pueden tener distintas escalas, siendo numéricas lineales, fibonacci, de tamaño (muy pequeña, pequeña, media, grande, etc), etc.

Los puntos de historia son un término relativo y no se relacionan directamente con un valor en horas (al menos necesariamente), lo que hace más fácil a los equipos ágiles scrum pensar de manera abstracta sobre el esfuerzo requerido para completar una historia.

¿De qué manera entonces sabremos qué representa un valor?

Para lograr esto, cada equipo debe encontrar una historia “base”, la cual no necesariamente tiene que ser la más pequeña, pero si una en la cual todo el equipo pueda interpretar y relacionarse de manera consensuada. A partir de esta, toda la estimación de tamaño estará en comparación con la historia base.

### Planning Póker

Planning poker es una técnica para calcular una estimación basada en el consenso, en su mayoría utilizada para estimar el esfuerzo o el tamaño relativo de las tareas de desarrollo de software. Es una variación del método Wideband Delphi. Es utilizado comúnmente en el desarrollo ágil de software, en particular en la metodología Extreme Programming y Scrum.

El póquer de planeamiento está basado en una lista de características para ser entregados y una baraja de cartas. La lista

de características, por lo general una lista de historias de usuario, describen un software que necesita ser desarrollado.

Las cartas en el mazo están numeradas. Un mazo típico contiene tarjetas mostrando la secuencia de Fibonacci incluyendo un cero: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. Otros mazos utilizan progresiones similares. La razón de utilizar la secuencia de Fibonacci es reflejar la incertidumbre inherente en la estimación.

En la reunión de la estimación a cada estimador se le da un conjunto completo de tarjetas.

La reunión prosigue de la siguiente manera:

- Un moderador, que no jugará, preside la reunión, apoyado y asesorado por el Gestor del Proyecto.
- El desarrollador con más conocimiento de una determinada característica proporciona una breve introducción sobre la misma. El equipo tiene la oportunidad de hacer preguntas y discutir para aclarar los supuestos y riesgos. Un resumen de la discusión es registrado por el Gestor del Proyecto.
- Cada persona coloca una tarjeta boca abajo que representa su estimación. Las unidades utilizadas pueden ser variadas y definidas previamente. Pueden ser días de duración, días ideales o puntos de la historia. Durante el debate, los números no debe ser mencionados en absoluto.
- Todo el mundo muestra sus tarjetas de forma simultánea.
- A las personas con estimaciones altas y bajas se les da un tiempo para ofrecer su justificación para la estimación y la discusión continúa.
- Se repita el proceso de cálculo hasta que se alcance un consenso. El programador que probablemente tenga el entregable tiene una gran parte del voto de consenso, aunque el moderador puede negociar el consenso.
- Se puede utilizar un reloj de arena para asegurar que el debate sea estructurado, el moderador o el Gestor del Proyecto podrá en cualquier punto terminar el reloj y cuando se acaba toda discusión debe cesar y otra ronda de póquer se juega.

El póquer de planificación es una herramienta para la estimación de los proyectos de desarrollo de software. Es una técnica que minimiza el anclaje, pidiendo a cada miembro del equipo que juegue su tarjeta de estimación sin ser visto por los demás jugadores. Después de que cada jugador ha seleccionado una tarjeta, todas las tarjetas están expuestos a la vez, lo que según estudios demuestra una efectividad mayor en la precisión de las estimaciones (suelen ser más pesimistas) y una evasión a la discusión que puede derivar en el denominado “anclaje”.

## Programación Extrema (XP)

La programación extrema o *eXtreme Programming* (XP) es una metodología de desarrollo de la ingeniería de software formulada por Kent Beck, autor del primer libro sobre la materia, *Extreme Programming Explained: Embrace Change* (1999). La programación extrema se diferencia de las metodologías tradicionales

principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad. Los defensores de la XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

Se puede considerar la programación extrema como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto, y aplicarlo de manera dinámica durante el ciclo de vida del software.

## Valores

Los valores originales de la programación extrema son: simplicidad, comunicación, retroalimentación (*feedback*) y coraje. Un quinto valor, respeto, fue añadido en la segunda edición de *Extreme Programming Explained*. Los cinco valores se detallan a continuación:

### Simplicidad

La simplicidad es la base de la programación extrema. Se simplifica el diseño para agilizar el desarrollo y facilitar el mantenimiento. Un diseño complejo del código junto a sucesivas modificaciones por parte de diferentes desarrolladores hacen que la complejidad aumente exponencialmente.

Para mantener la simplicidad es necesaria la refactorización del código, ésta es la manera de mantener el código simple a medida que crece.

También se aplica la simplicidad en la documentación, de esta manera el código debe comentarse en su justa medida, intentando eso sí que el código esté autodocumentado. Para ello se deben elegir adecuadamente los nombres de las variables, métodos y clases. Los nombres largos no decrementan la eficiencia del código ni el tiempo de desarrollo gracias a las herramientas de autocompletado y refactorización que existen actualmente.

Aplicando la simplicidad junto con la autoría colectiva del código y la programación por parejas se asegura que cuanto más grande se haga el proyecto, todo el equipo conocerá más y mejor el sistema completo.

### Comunicación

La comunicación se realiza de diferentes formas. Para los programadores el código comunica mejor cuanto más simple sea. Si el código es complejo hay que esforzarse para hacerlo inteligible. El código autodocumentado es más fiable que los comentarios ya que éstos últimos pronto quedan desfasados con el código a medida que es modificado. Debe comentarse sólo aquello que no va a variar, por ejemplo el objetivo de una clase o la

funcionalidad de un método.

Las pruebas unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de como utilizar su funcionalidad. Los programadores se comunican constantemente gracias a la programación por parejas. La comunicación con el cliente es fluida ya que el cliente forma parte del equipo de desarrollo. El cliente decide qué características tienen prioridad y siempre debe estar disponible para solucionar dudas.

### Retroalimentación

Al estar el cliente integrado en el proyecto, su opinión sobre el estado del proyecto se conoce en tiempo real.

Al realizarse ciclos muy cortos tras los cuales se muestran resultados, se minimiza el tener que rehacer partes que no cumplen con los requisitos y ayuda a los programadores a centrarse en lo que es más importante.

Considérense los problemas que derivan de tener ciclos muy largos. Meses de trabajo pueden tirarse por la borda debido a cambios en los criterios del cliente o malentendidos por parte del equipo de desarrollo. El código también es una fuente de retroalimentación gracias a las herramientas de desarrollo. Por ejemplo, las pruebas unitarias informan sobre el estado de salud del código. Ejecutar las pruebas unitarias frecuentemente permite descubrir fallos debidos a cambios recientes en el código.

### Coraje o valentía

Muchas de las prácticas implican valentía. Una de ellas es siempre diseñar y programar para hoy y no para mañana. Esto es un esfuerzo para evitar empantanarse en el diseño y requerir demasiado tiempo y trabajo para implementar el resto del proyecto. La valentía le permite a los desarrolladores que se sientan cómodos con reconstruir su código cuando sea necesario. Esto significa revisar el sistema existente y modificarlo si con ello los cambios futuros se implementaran mas fácilmente. Otro ejemplo de valentía es saber cuando desechar un código: valentía para quitar código fuente obsoleto, sin importar cuanto esfuerzo y tiempo se invirtió en crear ese código. Además, valentía significa persistencia: un programador puede permanecer sin avanzar en un problema complejo por un día entero, y luego quizás lo resuelva rápidamente al día siguiente, sólo si es persistente.

### Respeto

El respeto se manifiesta de varias formas. Los miembros del equipo se respetan los unos a otros, porque los programadores no pueden realizar cambios que hacen que las pruebas existentes fallen o que demore el trabajo de sus compañeros. Los miembros respetan su trabajo porque siempre están luchando por la alta calidad en el producto y buscando el diseño óptimo o más eficiente para la solución a través de la refactorización del código. Los

miembros del equipo respetan el trabajo del resto no haciendo menos a otros, una mejor autoestima en el equipo y elevando el ritmo de producción en el equipo.

## Características fundamentales

Las características fundamentales del método son:

- Desarrollo iterativo e incremental: pequeñas mejoras, unas tras otras.
- Pruebas unitarias continuas, frecuentemente repetidas y automatizadas, incluyendo pruebas de regresión. Se aconseja escribir el código de la prueba antes de la codificación.
- Programación en parejas: se recomienda que las tareas de desarrollo se lleven a cabo por dos personas en un mismo puesto. La mayor calidad del código escrito de esta manera -el código es revisado y discutido mientras se escribe- es más importante que la posible pérdida de productividad inmediata.
- Frecuente integración del equipo de programación con el cliente o usuario. Se recomienda que un representante del cliente trabaje junto al equipo de desarrollo.
- Corrección de todos los errores antes de añadir nueva funcionalidad. Hacer entregas frecuentes.
- Refactorización del código, es decir, reescribir ciertas partes del código para aumentar su legibilidad y mantenibilidad pero sin modificar su comportamiento. Las pruebas han de garantizar que en la refactorización no se ha introducido ningún fallo.
- Propiedad del código compartida: en vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, este método promueve el que todo el personal pueda corregir y extender cualquier parte del proyecto. Las frecuentes pruebas de regresión garantizan que los posibles errores serán detectados.
- Simplicidad en el código: es la mejor manera de que las cosas funcionen. Cuando todo funcione se podrá añadir funcionalidad si es necesario. La programación extrema apuesta que es más sencillo hacer algo simple y tener un poco de trabajo extra para cambiarlo si se requiere, que realizar algo complicado y quizás nunca utilizarlo.

La simplicidad y la comunicación son extraordinariamente complementarias. Con más comunicación resulta más fácil identificar qué se debe y qué no se debe hacer. Cuanto más simple es el sistema, menos tendrá que comunicar sobre éste, lo que lleva a una comunicación más completa, especialmente si se puede reducir el equipo de programadores.

## Lean Software Development (LSD)

La metodología de desarrollo de software Lean es una translación de los principios y prácticas de la manufactura esbelta hacia el dominio del desarrollo de software. Adaptado del Sistema de producción Toyota, apoyado por una sub-cultura *pro-lean* que está surgiendo desde la comunidad ágil.

### Origen

El término de desarrollo de software Lean tiene origen en un libro del mismo nombre, escrito por Mary Poppendieck y Tom Poppendieck. El libro presenta los tradicionales principios Lean en forma modificada, así como un conjunto de 22 instrumentos y herramientas y las comparaciones con otras prácticas ágiles. La participación de Mary y Tom en la comunidad del desarrollo ágil de software, incluyendo charlas en varias conferencias, ha dado lugar a dichos conceptos, que son más ampliamente aceptados en la comunidad de desarrollo ágil. Ejemplos de ello sería la utilización del término "*Lean-Agile*" por empresas de consultoría como NetObjectives Pace y CC, así como la inclusión de algunos de estos conceptos.

### Los principios Lean

El desarrollo Lean puede resumirse en siete principios, similares a los principios de manufactura esbelta.

#### Eliminar los desperdicios

Todo lo que no añade valor al cliente se considera un desperdicio:

- Código y funcionalidades innecesarias
- Retraso en el proceso de desarrollo de software
- Requisitos poco claros
- Burocracia
- Comunicación interna lenta

Con el fin de poder eliminar los desperdicios deberíamos ser capaces de reconocerlos y encontrarlos. Si alguna actividad podría ser excluida o el mismo resultado podría ser logrado sin ella, esta actividad es considerada un desperdicio. Los procesos y funcionalidades extra que no son usados por el cliente son desperdicios. Las esperas ocasionadas por otras actividades, equipos o procesos son desperdicio. Los defectos y la baja calidad son los desperdicios. Los gastos de gestión que no producen valor real son desperdicios. Se utiliza una técnica llamada *value stream mapping* (o mapa de flujo de valor) para distinguir y reconocer los desperdicios. El segundo paso consiste en señalar las fuentes de los desperdicios y eliminarlos. Lo mismo debe hacerse iterativamente hasta que incluso los procesos y procedimientos que parecían esenciales sean eliminados.

### Ampliar el aprendizaje

El desarrollo de software es un proceso de aprendizaje continuo, a ello se le suman los retos de los equipos de desarrollo y el tamaño del producto final. El mejor enfoque para encarar una mejora en el ambiente de desarrollo de software es ampliar el aprendizaje. La acumulación de defectos debe evitarse ejecutando las pruebas tan pronto como el código está escrito en lugar de añadir más documentación o planificación detallada. Las distintas ideas podrían ser probadas escribiendo código e integrándolo. El proceso de recopilación de requisitos de usuarios podría simplificarse mediante la presentación de las pantallas de los usuarios finales para que estos puedan hacer sus aportes. El proceso de aprendizaje es acelerado con el uso iteraciones cortas cada una de ellas acompañada de refactorización y sus pruebas de integración.

Incrementando la retroalimentación mediante reuniones cortas con los clientes se ayuda a determinar la fase actual de desarrollo y se ajustan los esfuerzos para introducir mejoras en el futuro. Durante las reuniones, tanto los clientes como el equipo de desarrollo, logran aprender sobre el alcance del problema y buscan posibles soluciones para un mejor desarrollo. Por lo tanto, los clientes comprenden mejor sus necesidades basándose en el resultado de los esfuerzos del desarrollo y los desarrolladores aprenden a satisfacer mejor estas necesidades.

Otra idea para ampliar el aprendizaje es a través de la integración del cliente en el ambiente de desarrollo para concentrar la comunicación en las soluciones futuras y no en las soluciones posibles, promoviendo así el nacimiento de la solución a través del diálogo con el cliente.

### Decidir lo más tarde posible

El desarrollo de software está siempre asociado con cierto grado de incertidumbre, los mejores resultados se alcanzan con un enfoque basado en opciones por lo que se pueden retrasar las decisiones tanto como sea posible hasta que éstas se basen en hechos y no en suposiciones y pronósticos inciertos. Cuanto más complejo es un proyecto, más capacidad para el cambio debe incluirse en éste, así que debe permitirse el retraso de los compromisos importantes y cruciales. El enfoque iterativo promueve este principio: la capacidad de adaptarse a los cambios y corregir los errores, ya que un error podría ser muy costoso si se descubre después de la liberación del sistema.

Un enfoque de desarrollo de software ágil puede llevarles opciones rápidamente a los clientes, lo que implica, retrasar algunas decisiones cruciales hasta que los clientes hayan reconocido mejor sus necesidades. Esto también permite la adaptación tardía a los cambios y previene las costosas decisiones delimitadas por la tecnología. Esto no significa que no haya planificación involucrada en el proceso, por el contrario, las actividades de planificación deben centrarse en las diferentes opciones y se les adapta a la situación actual; así como, se deben clarificar las situaciones confusas estableciendo las pautas para

una acción rápida. Evaluar las diferentes opciones es eficaz tan pronto como queda claro que ellos no son libres, pero proporcionando la flexibilidad necesaria para una tardía toma de decisiones.

### Reaccionar tan rápido como sea posible

En la era de la rápida evolución tecnológica, no es el más grande quien sobrevive, sino el más rápido. Cuanto antes se entrega el producto final sin defectos considerables más pronto se pueden recibir comentarios y se incorporan en la siguiente iteración. Cuanto más cortas sean las iteraciones, mejor es el aprendizaje y la comunicación dentro del equipo. Sin velocidad, las decisiones no pueden ser postergadas. La velocidad asegura el cumplimiento de las necesidades actuales del cliente y no lo que éste requería para ayer. Esto les da la oportunidad de demorarse pensando lo que realmente necesitan, hasta que adquieran un mejor conocimiento. Los clientes valoran la entrega rápida de un producto de calidad.

La ideología de producción *Just In Time* podría aplicarse a programas de desarrollo, reconociendo sus necesidades específicas y el ambiente. Lo anterior se logra mediante la presentación de resultados , la necesidad de dejar que el equipo se organizarse y dividiendo las tareas para lograr el resultado necesario para una iteración específica.

Al principio, el cliente dispone los requisitos necesarios. Esto podría ser simplemente presentar los requisitos en pequeñas fichas o historias y los desarrolladores estimarán el tiempo necesario para la aplicación de cada tarjeta. Así, la organización del trabajo cambia en sistema autopropulsado: cada mañana durante una reunión inicial cada miembro del equipo evalúa lo que se ha hecho ayer, lo que hay que hacer hoy y mañana y pregunta por cualquier nueva entrada necesaria de parte de sus colegas o del cliente. Esto requiere la transparencia del proceso, que es también beneficioso para la comunicación del equipo.

Otra idea clave del Sistema de Desarrollo de Producto de la Toyota se establece a base de diseño. Si un nuevo sistema de frenos es necesario para un coche, por ejemplo, tres equipos pueden diseñar soluciones al mismo problema. Cada equipo aprende sobre el ambiente del problema y diseños de una posible solución. Cuando una solución se considera irrazonable, se desecha. Al final de un periodo, los diseños sobrevivientes se comparan y se elige uno, quizá con algunas modificaciones basadas en el aprendizaje de los demás, un gran ejemplo de compromiso aplazado hasta el último momento posible. Las decisiones en el software también podrían beneficiarse de esta práctica para minimizar el riesgo provocado por un solo gran diseño realizado por adelantado.

### Potenciar el equipo

Ha habido una creencia tradicional en la mayoría de las empresas acerca de la toma de decisiones en la organización: los administradores dicen a los trabajadores cómo hacer su propio

trabajo. En una técnica llamada Work-Out, los roles cambian: a los directivos se les enseña a escuchar a los desarrolladores, de manera que éstos puedan explicar mejor qué acciones podrían tomarse, así como ofrecer sugerencias para mejoras. Los directores de proyecto más experimentados simplemente han declarado la clave de éxito de los proyectos: "Buscar la buena gente y dejarles hacer su propio trabajo".

Otra creencia errónea ha sido considerar a las personas como recursos. Las personas podrían ser los recursos desde el punto de vista de una hoja de datos estadísticos, pero en el desarrollo de software, así como cualquier organización de negocios, las personas necesitan algo más que la lista de tareas y la seguridad de que no será alterada durante la realización de las tareas. Las personas necesitan motivación y un propósito superior para el cual trabajar: un objetivo alcanzable dentro de la realidad con la garantía de que el equipo puede elegir sus propios compromisos. Los desarrolladores deberían tener acceso a los clientes; el jefe de equipo debe proporcionar apoyo y ayuda en situaciones difíciles, así como asegurarse de que el escepticismo no arruine el espíritu de equipo.

## Crear la integridad

El siempre exigente cliente debe tener una experiencia general del sistema. A esto se le llama percepción de integridad: ¿cómo es publicitado, entregado, implementado y accedido? ¿Qué tan intuitivo es su uso? ¿Precio? ¿Qué tan bien resuelve los problemas?. Integridad Conceptual significa que los componentes separados del sistema funcionan bien juntos, como en un todo, logrando equilibrio entre la flexibilidad, mantenibilidad, eficiencia y capacidad de respuesta. Esto podría lograrse mediante la comprensión del dominio del problema y resolviéndolo al mismo tiempo, no secuencialmente. La información necesaria es recibida por los pequeños lotes, no en una vasta cantidad y con una preferible comunicación cara a cara, sin ninguna documentación por escrito. El flujo de información debe ser constante en ambas direcciones, a partir del cliente a los desarrolladores y viceversa, evitando así la gran y estresante cantidad de información después de un largo periodo de desarrollo en el aislamiento.

Una de las maneras más saludables hacia una arquitectura integrante es la refactorización. Cuantas más funcionalidades se añaden a las del sistema, mas se pierde del código base para futuras mejoras. Así como en la Programación extrema (XP), la refactorización es mantener la sencillez, la claridad, la cantidad mínima de funcionalidades en el código. Las repeticiones en el código son signo de un mal diseño de código y deben evitarse. El completo y automatizado proceso de construcción debe ir acompañada de una suite completa y automatizada de pruebas, tanto para desarrolladores y clientes que tengan la misma versión, sincronización y semántica que el sistema actual. Al final, la integridad debe ser verificada con una prueba global, garantizando que el sistema hace lo que el cliente espera que haga. Las pruebas automatizadas también son consideradas como parte del proceso de producción y, por tanto, si no agregan valor deben considerarse residuos. Las pruebas automatizadas no deberían ser un objetivo,

sino, un medio para un fin; específicamente para la reducción de defectos.

Véase todo el conjunto

Los sistemas de software hoy en día no son simplemente la suma de sus partes, sino también el producto de sus interacciones. Los defectos en el software tienden a acumularse durante el proceso de desarrollo por medio de la descomposición de las grandes tareas en pequeñas tareas y de la normalización de las diferentes etapas de desarrollo. Las causas reales de los defectos deben ser encontradas y eliminadas. Cuanto más grande sea el sistema, más serán las organizaciones que participan en su desarrollo y más partes son las desarrolladas por diferentes equipos y mayor es la importancia de tener bien definidas las relaciones entre los diferentes proveedores con el fin de producir una buena interacción entre los componentes del sistema.

La manera de pensar ofrecida Lean tiene que ser bien entendida por todos los miembros de un proyecto antes de aplicarlo de manera concreta en una situación de la vida real.

"Piensa en grande, actúa en pequeño, equivócate rápido; aprende con rapidez" estas consignas resumen la importancia de comprender el terreno y la idoneidad de implementar los principios Lean lo largo del proceso de desarrollo de software. Sólo cuando todos los principios de Lean se aplican al mismo tiempo, combinado con un fuerte "sentido común" en relación con el ambiente de trabajo, hay una base para el éxito en el desarrollo de software.

# Consideraciones de contexto

Al momento de plantear una implementación de alguna metodología ágil o, al menos, sus principios generales, es un factor fundamental y determinante el de contexto.

¿Qué referencia hacemos cuando apuntamos hacia el contexto?

En un sentido amplio se entiende al contexto como al entorno, ambiente o medio donde se hará uso de las metodologías. Lo conforma las herramientas, procesos, actividades, resultados y personas que intervienen en las tareas del “objetivo”, del grupo donde se traducirán las prácticas ágiles.

Es así que se condicionará de diversas maneras la teoría y la práctica de las ágiles según el contexto particular de implementación.

## ¿Dónde pueden aplicarse las metodologías ágiles?

Si bien las metodologías ágiles están enfocadas en procesos de desarrollo de software, como ya hemos visto, pueden ser implementadas en otras áreas afines y, llegado el punto, hasta áreas completamente dispares.

Son una manera de organizar el trabajo y a un grupo de personas para lograr un objetivo común.

En nuestro caso particular vamos a centrar las consideraciones a lo estrictamente Software o afín.

## Consideraciones de la organización

Extrañamente (en comparación con las metodologías tradicionales) las metodologías ágiles nacen desde la base, desde los desarrolladores y llevan su impacto hacia todas las áreas de la organización, al menos que estén involucradas con el cliente y con el producto. De esta manera se desarrollaron en el tiempo.

Ahora bien, si pensamos en querer implementar metodologías ágiles en nuestro trabajo, y este es de desarrollo de software o relacionado con la informática, es deseable o panorama preferido tener el acuerdo, participación y entendimiento de los directivos de la organización que nos incluye.

No solo lo pensemos en cuestiones de conocimiento y entendimiento de las metodologías, es claro que se debe entender lo que se hace y con gran importancia en los estratos más “elevados”, sino que también debemos saber y entender que el alcance se extenderá hacia el cliente, intentando incluirlo en el proceso (en el mejor de los casos) o modificando el sistema de “delivery” o entrega (en los casos donde no se lo pueda contar como parte del equipo).

De esta forma la organización se verá transformada en diversos aspectos y es necesario tener visualizado, antes de comenzar una “agilización”, el objetivo y hasta dónde vamos a querer extender el alcance de la metodología.

Puede darse el caso que se defina adoptar algunas buenas prácticas de las metodologías pero continuar las relaciones con los clientes sin modificar, como también con las gerencias y hasta con otros grupos de trabajo. La flexibilidad, lo permita la teoría o no, siempre va a estar dictada por el grupo que lleve adelante el proceso.

Es también punto a considerar las normas de calidad o de seguridad con las que la organización esté comprometida.

## Consideraciones de equipo

Si bien uno puede gestionar su propio trabajo como individuo bajo el concepto de las metodologías ágiles, utilizando prácticas y herramientas, a la hora de pensar en abrir juego a nivel de equipo es fundamental que todos se acoplen a la utilización.

Difícil sería sincronizar trabajos y obtener los resultados ágiles reales si dentro del mismo equipo se desfasan las metodologías principalmente por los tiempos y las maneras de entregar valor que tendrá el equipo, sumado esto al juego de roles que debe darse.

Reduciéndonos explícitamente a programación podría

reconocerse una implementación de metodologías por parte de los desarrolladores pero que deberán convenir con el resto del grupo para que pueda ser consensuada y entendida la postura de trabajo.

### Consideraciones de cliente

El caso particular del cliente puede entenderse de varios modos. Suele ser más beneficioso contar con su apoyo y voluntad a entender los procesos y actividades del Software, pero en la realidad esto rara vez sucede y varía mucho también con el "contexto de cliente", que se refiere a la vinculación que quien solicita un servicio/producto software tiene con el área o la temática, pudiendo ser por interés o por afinidad de tareas.

En prácticas de negocio y de industria no sería conveniente insistir a un cliente de una metodología si este no posee interés en el tema, o el hecho de atender sus actividades ya es una demanda elevada que le exige una respuesta asesorada de software con bajo nivel de intervención.

Podemos notar que la tendencia de las metodologías ágiles está haciendo un fuerte espacio para una disciplina de **Ingeniería Social**, y posee una fuerte necesidad de fortalecer el espacio de toma de requerimientos, la cual histórica y actualmente ha sido de las más desatendidas y origen de errores y desvíos en los proyectos.

## Herramientas y técnicas

Existen una serie de herramientas y técnicas que rodean a las metodologías ágiles, las cuales pueden ser muy interesantes incorporar y ayudar a la adopción y comprensión de estas metodologías.

Algo importante en esto es que las ágiles se apoyan en herramientas informáticas y en técnicas de comunicación que las metodologías tradicionales intentaron mantener al margen para fortalecer su generalidad.

### Gestión de proyecto

La gestión de proyectos consta de llevar planificación y control de todas las actividades y personas que intervienen en la concreción de un objetivo común, como ya se ha nombrado.

Es una tarea clave en el desarrollo de Software y tiene sus particularidades ya que exige una diversa serie de conocimientos y responsabilidades a la o las personas que lo tomen en responsabilidad.

Existen para estos casos herramientas que apoyan estas tareas.

### Gestores de proyecto

Existen diversas opciones tanto comerciales como de licencia libre, podemos identificar:

- Redmine
- Trac
- JIRA
- Microsoft Project
- IBM Rational Team Concert
- Assembla
- Trello
- Alfresco

### Social coding

Las herramientas de social coding están más orientadas a desarrolladores y buscan la facilidad para compartir código, mejorarlo y darle crecimiento en comunidad. Podemos encontrar:

- Github
- Gitlab
- Gitorious
- Sourceforge
- Google code

## Desarrollo

Las herramientas de desarrollo y las técnicas conforman también la base de las metodologías ágiles, teniendo las siguientes opciones:

### TDD

De las siglas en inglés *Testing Driven Development (Desarrollo guiado por pruebas)*, esta metodología plantea centrar el desarrollo en las pruebas. Y no sólo eso, si no que pretende crear y ejecutar primero las pruebas antes que el código que ellas probarán.

¿Tiene esto sentido?

A un primer vistazo parece no tenerlo, pero una vez incorporada la metodología y practicada, cobra importante relevancia.

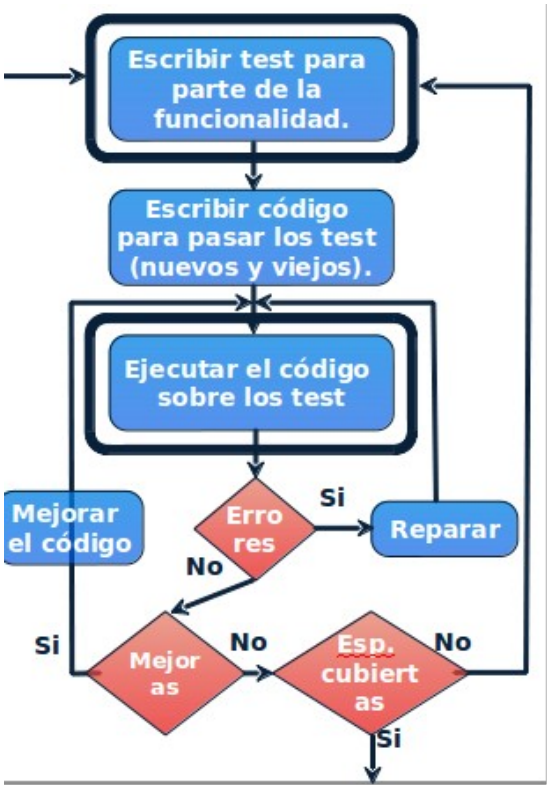
La intención es generar primero un requerimiento, del que se parte para generar una serie de casos de prueba, los suficientes como para asegurarnos una cobertura plena de lo que las funcionalidades deberían responder y abarcar. Luego de esto, el desarrollador toma un requerimiento, selecciona los casos de prueba correspondientes y comienza a escribir su prueba unitaria,

la porción de código que se ejecutará automáticamente para probar el código funcional del sistema, desde lo más general hacia lo más particular. Paso siguiente es ejecutar esa prueba unitaria y, como es de esperarse, fallará al no tener una implementación del código. Esa falla, nos dictamina el *pequeño paso* que debemos dar en codificación, guiando al desarrollo a través de la prueba. Se construye la solución, también general, para que esa prueba sea exitosa y se vuelve a correr el test, verificando su éxito. Es este el momento de la refactorización, que será la que determine el nivel de cobertura que se tendrá sobre las pruebas y el nivel de reutilización que pueda tener lo que estamos construyendo.

Resumiendo los pasos:

1. Elegir un requisitos
2. Escribir una prueba
3. Verificar que la prueba falla
4. Escribir la implementación
5. Ejecutar las pruebas automatizadas
6. Eliminación de la duplicación o refactorización
7. Actualizar la lista de requerimientos

Gráficamente:



## BDD

El Desarrollo guiado por comportamiento (Behavior Driven Development) es un proceso de desarrollo de software basado en TDD, el cual combina las técnicas y principios generales de TDD con ideas del Diseño guiado por el dominio y el análisis + diseño orientado a objetos, para proveer a los desarrolladores y analistas de negocio herramientas y procesos compartidos para colaborar en el desarrollo de software, con un objetivo de entregar software "que importe".

A pesar de que BDD es principalmente una idea sobre cómo el desarrollo de software debe ser gestionado por intereses de negocio y técnicos, la práctica de este asume el uso de software especializado y herramientas desarrolladas particularmente para este proceso de BDD. Estas herramientas sirven a sumar automatización a un lenguaje común conocido como "ubiquitous language", el cual debe ser fácilmente interpretado por analistas de negocio y desarrolladores.

## Integración contínua

La integración continua es un modelo informático propuesto inicialmente por Martin Fowler que consiste en hacer *integraciones automáticas* de un proyecto lo más a menudo posible para así poder detectar fallos cuanto antes. Entendemos por integración la compilación y ejecución de tests de todo un proyecto.

El proceso suele ser, cada cierto tiempo (horas), descargarse las fuentes desde el gestor de versiones (por ejemplo CVS, Git, Subversion, Mercurial o Microsoft Visual SourceSafe o Team Foundation Source Control) compilarlo, ejecutar tests y generar informes.

Para esto se utilizan distintas aplicaciones que se encargan de controlar las ejecuciones, apoyadas en otras herramientas que se encargan de realizar las compilaciones, ejecutar los tests y realizar los informes.

A menudo la integración continua está asociada con las metodologías de programación extrema y desarrollo ágil.

Tiene como ventajas:

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

### Jenkins

**Jenkins** es un software de Integración continua open source

escrito en Java. Está basado en el proyecto Hudson y es, dependiendo de la visión, un fork del proyecto o simplemente un cambio de nombre.

Jenkins proporciona integración continua para el desarrollo de software. Es un sistema corriendo en un servidor que es un contenedor de servlets, como Apache Tomcat. Soporta herramientas de control de versiones y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como scripts de shell y programas batch de Windows. Liberado bajo licencia MIT, Jenkins es software libre.

Web: http://jenkins-ci.org/

## Gestión de la configuración

Se denomina Gestión de la Configuración al conjunto de procesos destinados a asegurar la calidad de todo producto obtenido durante cualquiera de las etapas del desarrollo de un Sistema de Información (S.I.), a través del estricto control de los cambios realizados sobre los mismos y de la disponibilidad constante de una versión estable de cada elemento para toda persona involucrada en el citado desarrollo. Estos dos elementos (control de cambios y control de versiones de todos los elementos del S.I.) facilitan también el mantenimiento de los sistemas al proporcionar una imagen detallada del sistema en cada etapa del desarrollo. La gestión de la configuración se realiza durante todas las fases del desarrollo de un sistema de información, incluyendo el mantenimiento y control de cambios, una vez realizada la puesta en producción.

### Elementos de configuración de Software

Según la interfaz Gestión de la Configuración definida en MÉTRICA v3, los elementos de configuración del software incluyen:

- Ejecutables.
- Código Fuente.
- Modelos de datos.
- Modelos de procesos.
- Especificaciones de requisitos.
- Pruebas.

Y para cada uno de estos elementos se almacenará al menos:

- Nombre.
- Versión.
- Estado.
- Localización.

# Bibliografía


1. Wikipedia, la enciclopedia libre
2. http://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software
3. http://es.wikipedia.org/wiki/Scrum
4. http://es.wikipedia.org/wiki/Programaci%C3%B3n_extrema
5. http://es.wikipedia.org/wiki/Planning_poker
6. http://es.wikipedia.org/wiki/Lean_Software_Development
7. http://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas
8. http://en.wikipedia.org/wiki/Behavior-driven_development
9. http://es.wikipedia.org/wiki/Gesti%C3%B3n_de_la_configuraci%C3%B3n