# TDD

# 1. Metodologias Agiles

Cuando se habla de **metodologías ágiles**, es porque en los Proyectos que se desarrollan se cumple que

- no se terminan a tiempo
- cuestan más que lo presupuestado originalmente
- tienen problemas de calidad serios (aparecen demasiados **bug**)
- generan menor valor que el esperado por el cliente (cuando el cliente lo tiene no cubre sus necesidades en ese momento)

En estos proyectos, se emplean metodologias que se emplean en el mundo de la ingeniería, con procesos predefinidos con documentación muy precisa, y una detallada planificación inicial, que debe seguirse estrictamente.

En estos proyectos se emplea un **Ciclo de vida en Cascada** que se basa en la predictibilidad, en saber que hacer previamente y definir una serie de pasos concretos para conseguirlo.

Esta métodologia tiene varios problemas

- Al comienzo del proyecto, cuando menos conocimiento se tiene, se toman las decisiones de mayor relevancia.
- Al haber tomado las decisiones del analisis al comienzo del proyecto, no hay mucha capacidad de cambio a lo largo del mismo para adecuarse a las necesidades del cliente.
- Las estimaciones estarán, lo mas seguro, mal ya que cuando se hicieron, se tenia poca información.
- Si no se guarda especial atención a la calidad del software, se puede producir **Codigo spaguetti** (codigo con gran complejidad ciclomatica) que es dificil de mantener.
- Cuando se empiezan a acumular retrasos, normalmente se recurre a añadir mas gente al proyecto, lo cual esta demostrado que no significa mas trabajo sacado adelante, normalmente significa meter juniors con poco conocimiento y que añaden mas problemas.

El termino Ágil asociado al desarrollo del Software, indica flexibilidad y respuesta rápida a los cambios.

En las Metodologias Agiles, se plantea en cambio un **Ciclo de vida iterativo**, es decir, en lugar de realizar una unica entrega, se han de realizar entregas durante todo el proyecto, e **incremental**, es decir, que cada entrega incrementa el valor del proyecto.

En 2001 se funda la **The Agile Alliance**, que se encarga de difundir el Manifiesto Ágil.

## 1.1. Manifiesto Ágil (Valores)

Valorar al individuo y al equipo de desarrollo por encima de los procesos a seguir y las herramientas empleadas.

Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.

Valorar el software que funciona, por encima de la documentación sin un objetivo claro.

No producir documentos a menos que sean necesarios de forma inmediata para tomar un decisión importante. Los nuevos integrantes, se acoplarán entendiendo el código e interaccionando con el equipo.

La colaboración con el cliente más que la negociación de un contrato.

El cliente se verá involucrado en el día a día del proyecto.

Responder a los cambios más que seguir estrictamente un plan.

La planificación no debe ser estricta puesto que hay muchas variables en juego, debe ser flexible para poder adaptarse a los cambios que puedan surgir.

# 1.2. Manifiesto Ágil (Puntos)

1. La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.
2. Dar la bienvenida a los cambios (Los cambios son positivos. Permiten aprender más y logran una mayor satisfacción del cliente)
3. Entregas continuas de software (semanas o meses). La entrega no será documentación.
4. Interacción continua entre el Cliente y el equipo de desarrollo.
5. Equipo motivado con confianza y apoyo.
6. El medio para difundir la información es la palabra, no el documento.
7. El software funcional mide el progreso.
8. Velocidad de desarrollo sostenible.
9. Atención continua a la calidad y el buen diseño. Código claro y robusto.
10. Simplicidad en el código. Lo simple es fácil de adaptar.
11. El equipo decide la arquitectura, el diseño y la organización.
12. El equipo debe recapacitar sobre mejoras en su organización y adaptarse.

# 2. Programacion Extrema (XP)

Un tipo de **Metodología Ágil** especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes donde existe un alto riesgo técnico.

Hace hincapie en

- Realimentación continua entre el cliente y el equipo de desarrollo.
- Comunicación fluida entre todos los participantes
- Simplicidad en las soluciones implementadas
- Coraje para enfrentar los cambios.

## 2.1. Caracteristicas XP

- Historias de usuario.
- Roles.
- Proceso (Fases).
- Prácticas.

### 2.1.1. Historias de Usuario

Tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales.

Son dinámicas y flexibles, en cualquier momento pueden romperse, reemplazarse por otras más específicas o generales, añadirse nuevas o ser modificadas.

Cada una es lo suficientemente comprensible y delimitada para que se pueda implementar en unas semanas.

Debe existir al menos una historia de usuario por cada característica importante.

Se asignarán una o dos historias por programador al mes.

Son descompuestas en tareas de programación y asignadas a los programadores para ser implementadas durante una iteración.

Su contenido no esta definido, podría ser:

- Nombre.
- Descripción.
- Estimación de esfuerzo en días.

- Pruebas de aceptación.

### 2.1.2. Roles

- Programador: Produce el código y sus pruebas.
- Cliente: Escribe las historias de usuario, las prioriza y establece las pruebas de aceptación.
- Tester: Ayuda a escribir las pruebas de aceptación, y las ejecuta comprobando el avance.
- Tracker: Realiza el seguimiento de las iteraciones, perfila las estimaciones, buscando el cumplimiento de las iteraciones.
- Coach: Responsable del proceso XP, provee guías XP.
- Consultor: eersona externa al equipo que es un experto de un tema especifico.
- Gestor: Coordinador entre Cliente y equipo.

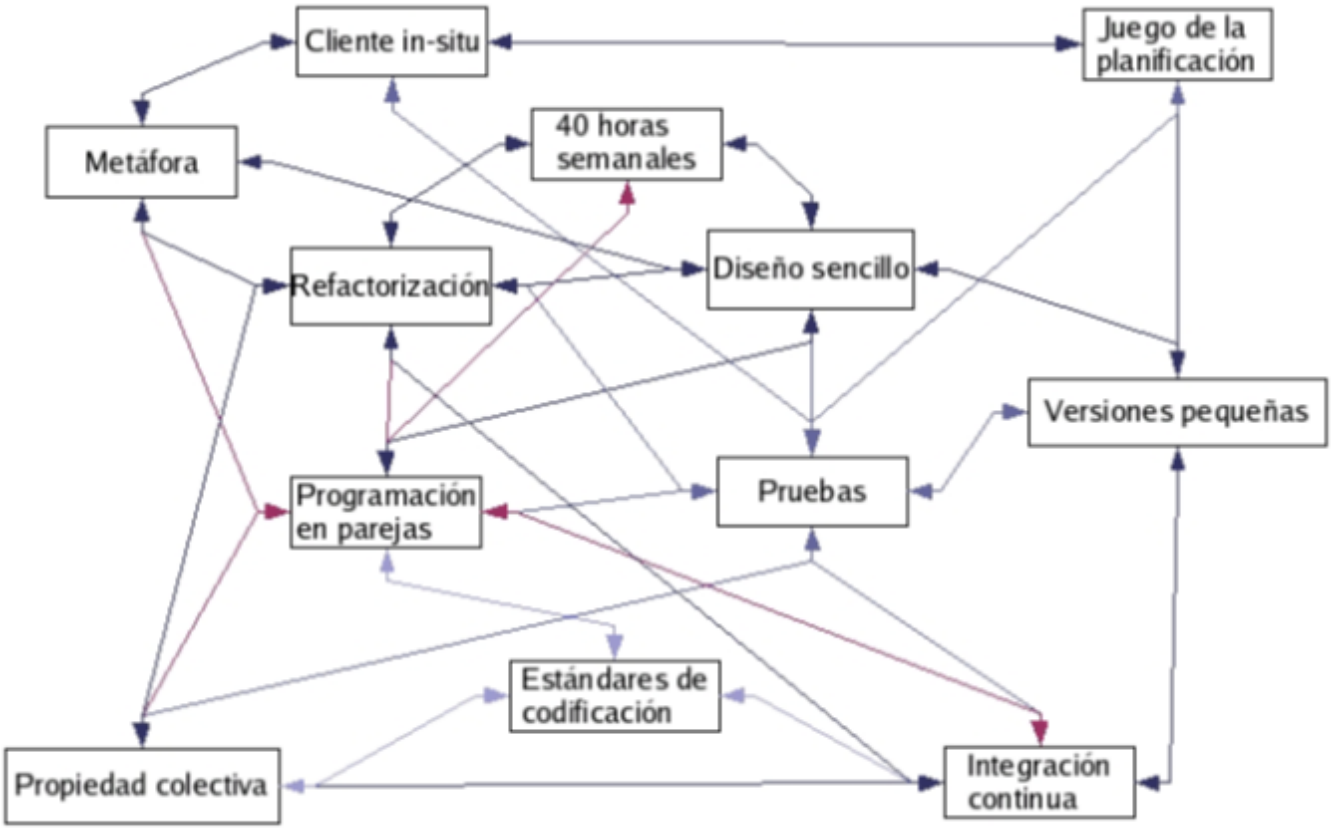
### 2.1.3. Proceso (Fases)

- Exploración. Planteamiento a grandes rasgos de las historias de usuario. Exploración de las tecnologías realizando un prototipo.
- Planificación de la Entrega (Release). El cliente establece la prioridad de cada historia. Los programadores realizan la estimación de las historias. Se determina que historias entran en la entrega y el cronograma de forma conjunta entre cliente y equipo. Se ha de enfrentar lo de mas valor y riesgo antes. Las entregas deberían durar máximo 3 meses. Las estimaciones se miden en puntos, siendo un punto 1 semana de trabajo normal de una persona. Las historias deberían durar de 1 a 3 puntos.
- Iteraciones. Son pequeñas entregas de no mas de 3 semanas, que forman la entrega (release). Se puede intentar realizar en las primeras iteraciones la arquitectura, pero depende del cliente, que buscará maximizar el negocio. Todo el trabajo de la iteración es expresado en tareas de programación. Cada tarea de programación, es asignada a un programador como responsable, pero llevadas a cabo por parejas de programadores. En todas las iteraciones tanto el cliente como el programador aprenden, por lo que tanto las historias de usuario como las estimaciones realizadas para una iteración posterior pueden verse modificadas. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. Los elementos a tener en cuenta son:
  - velocidad del proyecto.
  - pruebas de aceptación no superadas.
  - tareas no terminadas.
- Producción. Se realizaran pruebas adicionales, entre ellas las de rendimiento. Las nuevas propuestas de mejoras y nuevas funcionalidades, se documentan y se dejan para la fase de mantenimiento.
- Mantenimiento. Se han de simultanear el soporte para la versión en producción y las nuevas iteraciones que mejoran el producto, por lo que la velocidad de desarrollo puede verse afectada.

- Muerte del Proyecto. Se produce cuando o bien no hay mas historias de implementar y no hay presupuesto para mantenerlo o bien se abandona por no generar los beneficios esperados.

### 2.1.4. Prácticas

- El juego de la planificación.
- Entregas pequeñas.
- Diseño simple.
- Pruebas.
- Refactorización.
- Programación en parejas.
- Propiedad colectiva del código.
- Integración continua.
- 40 horas/semana.
- Cliente in-situ.
- Estándares de programación.



# 3. Test Driven Development

Un tipo de metodología ágil de desarrollo, que se basa en tres principios.

- Codificar las pruebas antes que el código que prueban.
- Codificar el mínimo código, en el mínimo tiempo posible, que consiga que se pasen las pruebas (sin importar demasiado el estilo o la arquitectura)
- Refactorizar el código creado adecuándolo a la arquitectura y ahora si aplicando un estilo adecuado.

El Mantra del TDD sobre el que se basa el ciclo de TDD es: **rojo-verde-refactor**.

- Rojo: escribe un test rápidamente, pequeño, que quizá ni funcione ni compile.
- Verde: haz que el test funcione rápidamente, cometiendo los errores de diseño que sean necesarios en el proceso. Tu obsesión: la barrita verde que significa que el test ha pasado.
- Refactor: elimina errores de diseño introducidos.

Lo ideal es que en hacer los tres pasos de TDD no se tarde más de media hora.

Una indicación clara de que estamos haciendo pasos muy grandes es que empiecen a fallarnos test inesperadamente.

Una cosa es ser consciente de que hemos hecho una modificación que hace que falle un test que ya funcionaba y otra es que cuando creemos que ya hemos acabado, un test nos falle sin esperarlo.

## 3.1. Ciclo de Vida

- Elegir un requisito: Se elige de una lista el requerimiento que se cree que nos dará mayor conocimiento del problema y que a la vez sea fácilmente implementable.
- Escribir una prueba: Se comienza escribiendo una prueba para el requisito. Este paso fuerza al programador a tomar la perspectiva de un cliente considerando el código a través de sus interfaces.
- Verificar que la prueba falla: Si la prueba no falla es porque el requerimiento ya estaba implementado o porque la prueba es errónea.
- Escribir la implementación: Escribir el código más sencillo que haga que la prueba funcione. Se usa la metáfora "Déjelo simple" ("Keep It Simple, Stupid" (KISS)).
- Ejecutar las pruebas automatizadas: Verificar si todo el conjunto de pruebas funciona correctamente.
- Eliminación de duplicados: Refactorizar para eliminar código duplicado. Los cambios serán progresivos, validando frecuentemente con las pruebas.

- Actualización de la lista de requisitos: Se actualiza la lista de requisitos tachando el requisito implementado. Asimismo se agregan requisitos que se hayan visto como necesarios durante este ciclo y se agregan requerimientos de diseño (P. ej que una funcionalidad esté desacoplada de otra).

## 3.2. Ventajas

- Código limpio que funciona (Clean code that works).
- Evita incluir arquitecturas innecesarias.
- No es necesario dedica tiempo posterior al desarrollo para la implementación de las pruebas.
- Cambio de punto de vista: El programador no realiza las pruebas para validar que su código funciona, hace un código que pase las pruebas.