UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE EXPLOTACIÓN Y PROSPECCIÓN DE MINAS

MASTER INTERUNIVERSITARIO EN DIRECCIÓN DE PROYECTOS

TRABAJO FIN DE MASTER

Identificación y valoración de técnicas ágiles de gestión de proyectos software

Autor: Pedro Sáez Martínez
Director: Joaquín Villanueva Balsera

16 de Julio de 2013

Resumen

En este texto se acerca al lector al mundo de la gestión ágil de proyectos software, ofreciéndole una serie de información y datos con el objetivo de ayudarle a la hora de elegir un modelo de ciclo de vida, y una metodología ágil (en el caso de usar modelo ágil) para su proyecto.

Para poner en contexto al lector, se explican algunos conceptos claves en los que se basa el documento y que si bien son ampliamente utilizados, muchas veces no se usan de manera adecuada e incluso existe confusión sobre ellos.

A continuación se sitúa al lector explicándole el porqué de la necesidad de mejora en la gestión de proyectos software, comentando el alto índice de fracasos que hay en el desarrollo software, y haciéndole entender que la elección de un modelo y una metodología es un proceso crítico y básico para el éxito de un proyecto.

Con todo esto y una vez ya preparado para avanzar al siguiente paso, se explican los modelos de ciclo de vida más importantes e influyentes para el desarrollo software, con diagramas y datos que nos ayudarán a comprender el fin de cada uno de estos. A partir de ahí aquí el documento se adentra un poco más informando sobre las metodologías ágiles más relevantes y las técnicas ágiles más útiles y utilizadas.

Por último y gracias a comparativas, tablas y árboles de decisión, se consigue lograr el fin declarado al principio, una elección o al menos una sugerencia para nuestro lector sobre qué modelo y metodología son más adecuadas para su proyecto.

Junto con este documento se acompaña una tabla Excel que ayudará a realizar una mejor elección de la metodología.

Este documento puede ser útil para un director de proyecto que esté iniciando un proyecto y se encuentre ante la tesitura de elegir una metodología a seguir o unas técnicas a usar, también puede ser útil para una organización que quiere buscar aquel modelo de ciclo de vida que mejor se adapta a sus características, sus clientes y la tipología de sus proyectos, sin olvidar que quizás pueda ser útil como base para aquellas personas dedicadas a la formación, ya sea en instituciones públicas o privadas, y que quieran avanzar más en el tema de la gestión ágil de proyectos.

Índice General

Índice de Figuras

Capítulo 1. Introducción

1.1 Justificación del Trabajo

La Ingeniería Informática, puede que por ser la más moderna de las ingenierías, sea también la menos organizada en cuanto a gestión de proyectos. No es raro encontrarse proyectos que tras invertir una cantidad de dinero y tiempo considerables, dejan el producto a medias o con grandes deficiencias, o simplemente que no llegan a nada. Si se piensa en otros campos de la ingeniería, en los que tienen procedimientos establecidos, normas y modos de hacer más coordinados y controlados, el éxito aumenta o al menos la incertidumbre en estos proyectos es menor.

Esto no quiere decir que los ingenieros informáticos se hayan olvidado de la importancia de una gestión adecuada de un proyecto, pero muchas veces no se le da el valor que merece, y simplemente se limitan a aprender a hacer una buena toma de requisitos y un presupuesto, centrándose más en tareas de diseño y programación.

Aun así, según va madurando la profesión, y los propios ingenieros informáticos están haciéndose con los puestos de dirección de las empresas de IT, puestos antes copados por perfiles de otras carreras, el "problema" de la gestión de proyectos se ha convertido en un tema que les preocupa y mucho, y en base a ello aplican mucho esfuerzo, tiempo y dinero en buscar la forma de dirigir adecuadamente su barco hacia su objetivo.

Pero de momento la mayoría de la información existente o que se ha aplicado se basa en modelos que han triunfado en otras ingenierías, pero que al ser trasladados al mundo del software, fracasan o tienen poco éxito.

Al no existir un modelo exitoso a seguir, muchos equipos de desarrollo se limitan a abandonarlos, pero sin utilizar otras metodologías de desarrollo. Esto lleva de vuelta a un escenario donde la ausencia de una metodología dispara las probabilidades de fracaso de un proyecto.

Las metodologías ágiles surgieron como una posible solución a este problema ya que están orientadas a proyectos software donde la incertidumbre suele ser muy alta tanto en tiempo como en costes, los requisitos cambian constantemente y las tecnologías evolucionan a una velocidad de vértigo que hace que la mayoría de las veces no puedas usar unos proyectos como referencia de otros.

Y ante todo este panorama, cuando un director de proyectos software, o una empresa de IT tiene que elegir qué modelo de ciclo de vida quiere para sus proyectos y que metodología va seguir, se encuentra ante una ingente cantidad de información, con pros y contras para cada modelo y cada metodología. La moda les lleva a muchos a decir "Modelo Ágil", e inmediatamente dicen "Scrum", y esta situación es la que se intenta evitar con este documento.

En este documento se quiere explicar que modelos de ciclo de vida existen, cuales son más adecuados en cada caso, y en el supuesto de que para tu proyecto u organización el ágil sea el más correcto, se da una introducción de las distintas metodologías que existen para este modelo (y no sólo Scrum), haciendo una comparativa de ellas en base a varios factores, para finalmente explicar distintas técnicas que se puedan usar con cada una de ellas. Todo esto con el fin de que la próxima vez que alguien se encuentre ante esta tesitura, tenga unas tablas, datos, diagramas e información centralizado en un único documento sobre el que consultar.

1.2 Objetivos del Trabajo

En el mundo IT la gestión de proyectos cada vez tiene más importancia, cada vez se ofertan más puestos para este perfil, buscando gente capacitada a seguir unos procedimientos que aumenten las posibilidades de éxito de un proyecto.

Esto está siendo escuchado por el mundo laboral, y los informáticos se han lanzado a aprender sobre el modelo de gestión del ciclo de vida de un proyecto de moda para el mundo software, el modelo ágil, pero centrándose exclusivamente en su metodología más conocida, Scrum.

Esto ha llevado a muchos a pensar que gestión de proyecto software es igual a modelo ágil y que este es igual a Scrum, o lo que es lo mismo, en los cursos y demás seminarios sobre Scrum, salen con la idea de que si quieren que su proyecto se gestiones adecuadamente, lo único que tienen que hacer es aplicar Scrum.

Esto es un error, y está basado en el desconocimiento existente en la materia. El objetivo de este trabajo no es otro que el aportar un poco de información sobre este tema para ayudar a aquellos que se estan adentrando en el mundo de la gestión de proyectos y que han mostrado interés por las metodologías ágiles.

De forma enumerada lo resumiría en:

- Introducir al lector en el mundo de la gestión de proyectos software. Mostrarle la situación actual en la que nos encontramos. Sus retos y sus oportunidades.
- Analizar los modelos existentes, para evaluar cuál sería la mejor alternativa, desde el punto de vista de gestión, a la hora de encarar un proyecto software
- Explicar y comparar las metodologías ágiles existentes dentro del modelo ágil y las técnicas que se pueden usar con cada una de ellas.

1.3 Alcance del Trabajo

Con este trabajo se busca acercar al lector al mundo de la gestión de proyectos ágil, pero sobre todo el buscar el modelo adecuado y la metodología (en caso de usar modelo ágil) más madura y correcta, a la hora de gestionar un proyecto, de entre las actualmente existentes.

Se entiende que no existe un modelo de ciclo de vida ideal o una metodología ágil única, sino que se busca la que se comporta de mejor modo en cada tipo de proyecto, explicando las razones para usar uno u otro, todo expuesto de una forma entendible y accesible.

Además se dará información sobre algunas técnicas útiles a la hora de usar estas metodologías ágiles.

Queda fuera del alcance de este trabajo el implementar estas metodologías en un proyecto real, o el profundizar demasiado en ellas.

1.4 Preguntas a Resolver

- ¿Qué problemas tienen los modelos tradicionales a la hora de gestionar proyectos software? ¿Qué ofrecen? ¿En que fallan?
- ¿Es el modelo ágil la solución perfecta?
- ¿Qué ventajas y desventajas tiene cada una de los modelos de ciclo de vida más comunes del desarrollo software?
- ¿Qué metodologías hay disponibles si usamos el modelo ágil? ¿Cuál viene mejor usar en cada caso?
- ¿Qué técnicas se recomienda utilizar de cara a facilitar la gestión de proyectos usando estas metodologías ágiles?

1.5 Estructura del Trabajo

Parte 1, “Aspectos Teóricos”. (Cubre el capítulo 2) Esta sección está destinada a describir brevemente aquellos conceptos existentes que se van a usar en la investigación.

Parte 2, “Situación Actual”. (Cubre el capítulo 3) En esta parte se habla un poco sobre la historia de la gestión de proyectos software. Del potencial que tiene y de lo beneficioso o no que puede ser el aplicar una metodología adecuada. Se describen los modelos de gestión del ciclo de vida más comúnmente utilizados en las empresas IT. Además se comentarán las múltiples metodologías ágiles que se podrían usar a la hora de encarar el proyecto y las distintas técnicas aplicables que ya existen y que están apareciendo para facilitar a los gestores y jefes de proyecto las tareas relacionadas con su trabajo diario.

Parte 3, “Acercamiento al problema”. (Cubre el capítulo 4) Se explica el problema que es para quien tiene que decidir la metodología a seguir para gestionar un proyecto, el encontrarse con tantos modelos y metodologías aplicables. Se verán las ventajas y desventajas de cada uno y se compararán en base a distintos factores.

Parte 4, “Metodología a Usar”. (Cubre los capítulos 5 y 6) Aquí se explica cuándo se debe usar un modelo ágil. Además se orienta sobre la metodología a seguir en cada tipo de proyecto y las técnicas que se podrían usar sobre esa metodología.

Parte 5, “Referencias”. (Cubre el capítulo 7) Libros y artículos usados de alguna forma durante el desarrollo del proyecto o su documentación.

Capítulo 2. Aspectos Teóricos

En esta sección se clarificarán algunos de los conceptos más importantes que se van a manejar a lo largo de todo el documento.

2.1 Software

De forma simplificada podríamos definir el software como la información codificada, que es transmita al hardware, para que este la procese y la ejecute.

De un modo más formal podemos encontrar otras definiciones de **software**:

- IEEE Std. 610 define el software como "programas, procedimientos y documentación y datos asociados, relacionados con la operación de un sistema informático"
- Según el Webster's New Collegiate Dictionary (1975), "software es un conjunto de programas, procedimientos y documentación relacionada asociados con un sistema, especialmente un sistema informático".

El software se puede definir como el conjunto de **tres componentes**:

- **Programas** (instrucciones): Este componente proporciona la funcionalidad deseada y el rendimiento cuando se ejecute. Los programas son conjuntos de instrucciones que proporcionan la funcionalidad deseada cuando son ejecutadas por el hardware. Están escritos usando lenguajes específicos que el hardware puede leer, interpretar y ejecutar, tales como lenguaje ensamblador, Basic, FORTRAN, COBOL, C, Java, C#… Los programas también pueden ser generados usando generadores de programas.
- **Datos**: Este componente incluye los datos necesarios para manejar y probar los programas y las estructuras requeridas para mantener y manipular estos datos.
- **Documentos**: Este componente describe la operación y uso del programa. Los documentos son necesarios tanto para que los usuarios sepan usar el programa como para que otras personas encargadas de mantener el software puedan entender el interior del software y modificarlo en el caso de que sea necesario.

Figura 1 – Componentes del Software

Algunas características del software son:

- El software se **desarrolla**, no se fabrica en el sentido clásico. Cada producto software es diferente porque se construye para cumplir los requisitos únicos de un cliente. Cada software necesita, por lo tanto, ser construido usando un enfoque de ingeniería.
- En el software, **el recurso principal son las personas**. No es siempre posible acelerar la construcción de software añadiendo personas porque la construcción de software

requiere un esfuerzo en equipo. El equipo tiene que trabajar de forma coordinada y compartir un objetivo de proyecto común. Se necesita comunicación efectiva dentro del equipo.

- Otra característica del software es que **no se estropea**. Los defectos no detectados harán que falle el programa durante las primeras etapas de su vida. Sin embargo, una vez que se corrigen (suponiendo que no se introducen nuevos errores) los fallos disminuyen o puede que el software se deteriore debido a la inclusión de nuevos errores.

El software se puede clasificar, según su función principal, en tres categorías:

- **Software de sistema:** El software se encarga de gestionar la complejidad de los dispositivos hardware. Los Sistema Operativos serían un tipo de este software.
- **Software de aplicación:** Esta categoría engloba todo aquel software cuyo propósito es ayudar a realizar al usuario una tarea.
- **Software de desarrollo:** Todos aquellos programas que permiten construir programas. Los entornos de desarrollo (IDE) serían un ejemplo de estos programas.

2.2 Ingeniería del Software

La definición de IEEE describe la ingeniería del software como un enfoque sistemático cubriendo los aspectos del desarrollo, operación y mantenimiento. Este enfoque es disciplinado y cuantificable.

La ingeniería de software es una disciplina formada por un conjunto de métodos, herramientas y técnicas que se utilizan en el desarrollo de los programas informáticos (software).

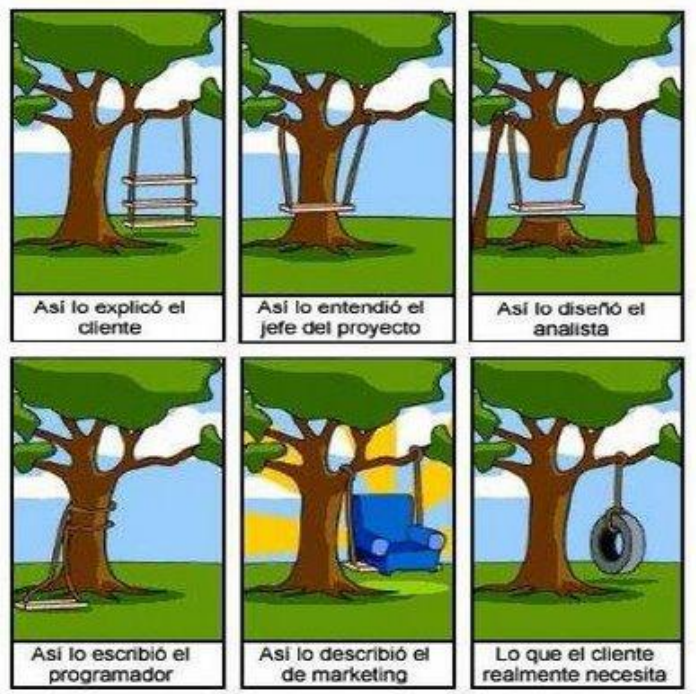


Figura 2 – Problema típico en la Ingeniería de Software

Esta disciplina trasciende de la actividad de programación, que es el pilar fundamental a la hora de crear una aplicación. El ingeniero de software se encarga de toda la gestión del proyecto para que éste se pueda desarrollar en un plazo determinado y con el presupuesto previsto.

La ingeniería de software, por lo tanto, incluye el análisis previo de la situación, el diseño del proyecto, el desarrollo del software, las pruebas necesarias para confirmar su correcto funcionamiento y la implementación y mantenimiento del sistema.

Un campo directamente relacionado con la ingeniería de software y con el que se suele crear confusión entre ambos términos, es la arquitectura de sistemas, que consiste en determinar y esquematizar la estructura general del proyecto, diagramando su esqueleto con un grado relativamente alto de especificidad y señalando los distintos componentes que serán necesarios para llevar a cabo el desarrollo, tales como aplicaciones complementarias y bases de datos. Se trata de un punto fundamental del proceso, y muchas veces es la clave del éxito de un producto informático. Dicho de otra manera la ingeniería de software abarca todos los procesos relacionados para desarrollar programas y dentro de esos procesos se encuentra la arquitectura del software.

2.3 Ciclo de Vida del Proyecto

Todo proyecto de ingeniería tiene unos fines relacionados con la obtención de un producto, proceso o servicio que es necesario generar a través de diversas actividades. Algunas de estas actividades pueden agruparse en fases porque globalmente contribuyen a obtener un producto intermedio, necesario para continuar hacia el producto final y facilitar la gestión del proyecto. Al conjunto de las fases empleadas se le denomina “ciclo de vida”, y a veces también “paradigma”.

Sin embargo, la forma de agrupar las actividades, los objetivos de cada fase, los tipos de productos intermedios que se generan, etc. pueden ser muy diferentes dependiendo del tipo de producto o proceso a generar y de las tecnologías empleadas.

Entre las funciones que debe tener un ciclo de vida se pueden destacar:

- Determinar el orden de las fases del proyecto
- Establecer los criterios de transición para pasar de una fase a la siguiente
- Definir las entradas y salidas de cada fase
- Describir los estados por los que pasa el producto
- Describir las actividades a realizar para transformar el producto
- Definir un esquema que sirve como base para planificar, organizar, coordinar, desarrollar…

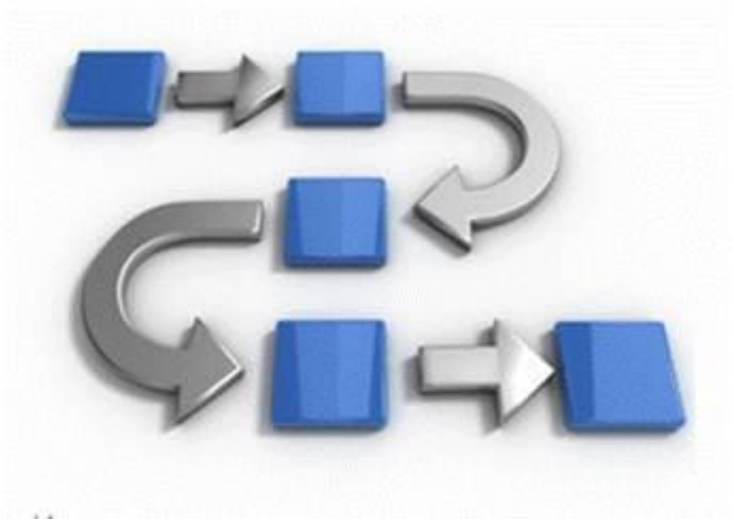


Figura 3 – Diagrama simplificado de concepto de "Ciclo de Vida"

Por lo tanto un ciclo de vida para un proyecto se compone de fases sucesivas compuestas por tareas planificables. Según el modelo de ciclo de vida, la sucesión de fases puede ampliarse con bucles de realimentación, de manera que lo que conceptualmente se considera una misma fase se pueda ejecutar más de una vez a lo largo de un proyecto, recibiendo en cada pasada de ejecución aportaciones de los resultados intermedios que se van produciendo.

Para un adecuado control de la progresión de las fases de un proyecto se hace necesario especificar con suficiente precisión los resultados evaluables, o sea, productos intermedios que deben resultar de las tareas incluidas en cada fase. Normalmente estos productos marcan los hitos entre fases.

Así podemos hablar de:

- Fases → Una fase es un conjunto de actividades relacionadas con un objetivo en el desarrollo del proyecto. Se construye agrupando tareas (actividades elementales) que pueden compartir un tramo determinado del tiempo de vida de un proyecto. La agrupación temporal de tareas impone requisitos temporales correspondientes a la asignación de recursos (humanos, financieros o materiales).
- Entregables → Son los productos intermedios que generan las fases. Pueden ser materiales o inmateriales (documentos, software). Los entregables permiten evaluar la marcha del proyecto mediante comprobaciones de su adecuación o no a los requisitos funcionales y de condiciones de realización previamente establecidos.

No existe una única manera, que sea la mejor, para definir el ciclo de vida ideal de un proyecto. Algunas organizaciones han establecido políticas que estandarizan todos los proyectos con un ciclo de vida único, mientras que otras permiten al equipo de dirección del proyecto elegir el ciclo de vida más apropiado para el proyecto del equipo. Asimismo, las prácticas comunes de la industria a menudo conducen a usar un ciclo de vida preferido dentro de dicha industria.

2.4 Modelo de Ciclo de Vida del Software

La ingeniería del software establece y se vale de una serie de modelos que definen y muestran las distintas etapas y estados por los que pasa un producto software, desde su concepción inicial, pasando por su desarrollo, puesta en marcha y posterior mantenimiento, hasta la retirada del producto. A estos modelos se les denomina "Modelos de ciclo de vida del software". El primer modelo concebido fue el de Royce, más comúnmente conocido como Cascada o "Lineal Secuencial". Este modelo establece que las diversas actividades que se van realizando al desarrollar un producto software, se suceden de forma lineal.

Un modelo de ciclo de vida de software es una vista de las actividades que ocurren durante el desarrollo de software, intenta determinar el orden de las etapas involucradas y los criterios de transición asociados entre estas etapas. Así podríamos resumir sus funciones en:

- Describe las fases principales de desarrollo de software.
- Define las fases primarias esperadas de ser ejecutadas durante esas fases.
- Ayuda a administrar el progreso del desarrollo.
- Provee un espacio de trabajo para la definición de un proceso detallado de desarrollo de software

En cada una de las etapas de un modelo de ciclo de vida, se pueden establecer una serie de objetivos, tareas y actividades que lo caracterizan. Existen distintos modelos de ciclo de vida, y la elección de un modelo para un determinado tipo de proyecto es realmente importante; el orden es uno de estos puntos importantes.

Figura 4 – Representación gráfica de un modelo de ciclo de vida lineal en construcción

Las principales diferencias entre distintos modelos de ciclo de vida están en:

- **El alcance del ciclo** dependiendo de hasta dónde llegue el proyecto correspondiente. Un proyecto puede comprender un simple estudio de viabilidad del desarrollo de un

producto, o su desarrollo completo o en el extremo, toda la historia del producto con su desarrollo, fabricación y modificaciones posteriores hasta su retirada del mercado.

- **Las características (contenidos) de las fases** en que dividen el ciclo. Esto puede depender del propio tema al que se refiere el proyecto, o de la organización.
- **La estructura y la sucesión de las etapas**, si hay realimentación entre ellas, y si tenemos libertad de repetirlas (iterar).

Algunas de las fases típicas que nos encontramos en los distintos modelos del ciclo de vida software son:

- **ANÁLISIS**: En esta etapa se debe entender y comprender de forma detallada cual es la problemática a resolver, verificando el entorno en el cual se encuentra dicho problema, de tal manera que se obtenga la información necesaria y suficiente para afrontar su respectiva solución. Esta etapa es conocida como la del QUÉ se va a solucionar.
- **DISEÑO**: Una vez que se tiene la suficiente información del problema a solucionar, es importante determinar la estrategia que se va a utilizar para resolver el problema. Esta etapa es conocida bajo el CÓMO se va a solucionar.
- **IMPLEMENTACIÓN**: Partiendo del análisis y diseño de la solución, en esta etapa se procede a desarrollar el correspondiente programa que solucione el problema mediante el uso de una herramienta computacional determinada.
- **PRUEBAS**: Los errores humanos dentro de la programación son muchos y aumentan considerablemente con la complejidad del problema. Cuando se termina de escribir un programa, es necesario realizar las debidas pruebas que garanticen el correcto funcionamiento de dicho programa bajo el mayor número de situaciones posibles a las que se pueda enfrentar.
- **DOCUMENTACIÓN**: Es la guía o comunicación escrita en sus diferentes formas, ya sea en enunciados, procedimientos, dibujos o diagramas que se hace sobre el desarrollo de un programa. La importancia de la documentación radica en que a menudo un programa escrito por una persona, es modificado por otra. Por ello la documentación sirve para ayudar a comprender o usar un programa o para facilitar futuras modificaciones (mantenimiento).
- **MANTENIMIENTO**: Una vez instalado un programa y puesto en marcha para realizar la solución del problema previamente planteado o satisfacer una determinada necesidad, es importante mantener una estructura de actualización, verificación y validación que permitan a dicho programa ser útil y mantenerse actualizado según las necesidades o requerimientos planteados durante su vida útil. Para realizar un adecuado mantenimiento, es necesario contar con una buena documentación del mismo.

El orden y la presencia de cada uno de estos procedimientos u otros nuevos en el ciclo de vida de una aplicación dependen del tipo de modelo de ciclo de vida elegido.

2.5 Metodología

Una metodología es un conjunto integrado de técnicas y métodos que permite abordar de forma homogénea y abierta cada una de las actividades del ciclo de vida de un proyecto de desarrollo. Estas se basan en una combinación de los modelos de proceso genéricos (cascada, incremental…). Definen artefactos, roles y actividades, junto con prácticas y técnicas recomendadas. Es decir definen qué hacer, cómo y cuándo durante todo el desarrollo y mantenimiento de un proyecto.

La metodología para el desarrollo de software en un modo sistemático de realizar, gestionar y administrar un proyecto para llevarlo a cabo con altas posibilidades de éxito. Así comprende los procesos a seguir sistemáticamente para idear, implementar y mantener un producto software desde que surge la necesidad del producto hasta que cumplimos el objetivo por el cual fue creado.

Una metodología define una estrategia global para enfrentarse con el proyecto. Entre los elementos que forman parte de una metodología se pueden destacar:

- Fases: tareas a realizar en cada fase.
- Productos: E/S de cada fase, documentos.
- Procedimientos y herramientas: apoyo a la realización de cada tarea.
- Criterios de evaluación: del proceso y del producto. Saber si se han logrado los objetivos.

El modelo agrupa la metodología, es decir, el modelo por ejemplo es el de cascada, pero para cumplir con ese modelo tienes que seguir una serie de pasos, a esos pasos se les llama metodología, en resumen la metodología es la serie de pasos que necesitas realizar para que se pueda decir que estas cumpliendo con el modelo.

2.6 Técnicas

Es el conjunto de instrumentos, herramientas, medios o modos de hacer que utilizamos para llevar a cabo o facilitar nuestro trabajo. Las metodologías suelen definir qué técnicas son convenientes aplicar a la hora de poder cumplir el método que describen.

Figura 5 – EDP, Gantt, CPM… son ejemplos de técnicas de gestión de proyectos

Capítulo 3. Situación Actual

3.1 Fracaso en los Proyectos Software

En diversas organizaciones, se está presentando una situación alarmante con respecto a que la mayor parte de los proyectos informáticos no llegan a buen término, ya sea porque no cumplen con los requisitos planteados o simplemente son cancelados debido a que superan las expectativas de tiempo y/o costo.

La mayor parte de los proyectos de software se desarrollan por equipos de desarrollo de unas cuantas personas hasta grandes grupos de varias decenas de individuos. Son personal altamente especializado y calificado con actividad prácticamente intelectual y creativa. Aun así, la percepción común que se tiene de la industria del software es que es una industria sólida pero sorprendentemente, no se caracteriza por la alta calidad generalizada de sus productos y servicios.

Existen diversos autores que han realizado investigaciones en torno a este tema y la mayoría coinciden en que aproximadamente un 20% de los proyectos informáticos iniciados finalizan de forma exitosa mientras que el restante 80% son finalizados con algún tipo de problema.

El estudio más citado y relevante es el titulado "The CHAOS Report" publicado por el Standish Group. Este grupo se creó en 1985 por una serie de profesionales de West Yarmouth, Massachussets con una visión: obtener información de los proyectos fallidos de IT. El objetivo: encontrar (y combatir) las causas de los fracasos. Con el tiempo, la seriedad y el profesionalismo del Standish Group lo convirtieron en un referente mundial sobre los factores que inciden en el éxito o fracaso de los proyectos de IT. Sus datos dicen ser de más de 50.000 proyectos (aunque no permiten el acceso a esta información).

El Standish Group clasifica los proyectos en tres tipos:

- **Exito** (Successful): El proyecto se completa en tiempo y dentro del presupuesto, con todas las características y funciones (discutible el que se deje fuera otros aspectos como la calidad, el riesgo y la satisfacción del cliente).
- **Cuestionado** (Challenged): El proyecto se completa y es operacional, pero más allá del presupuesto, más allá del tiempo estimado y con pocas de las características y funciones que fueron especificadas inicialmente.
- **Fracasado** (Failed): El proyecto es cancelado antes de completarse

El citado informe tuvo su primera versión en 1994, y nos reveló que ese año el 31 por ciento de los proyectos fueron cancelados. El 53 por ciento registró enormes desvíos en presupuesto y en cronograma. Sólo el 16 por ciento se completó en tiempo y dentro de los costos presupuestados (apenas nueve por ciento en el caso de grandes empresas). Para colmo, de la funcionalidad comprometida sólo se cumplió, en promedio, con el 61 por ciento (42 por ciento en grandes empresas).

En vista de estos fracasos, durante los últimos veinte años, la industria ha invertido varios miles de millones de dólares en el desarrollo y perfeccionamiento de metodologías y

tecnologías (PMI, CMMI, ITIL, SOA, etc.) destinadas a mejorar la administración y productividad de los proyectos de software. Así en el informe del Standish Group del 2011 las cifras eran de 34% de proyectos terminados con éxito, 51% con desviaciones y 15% de fracaso.

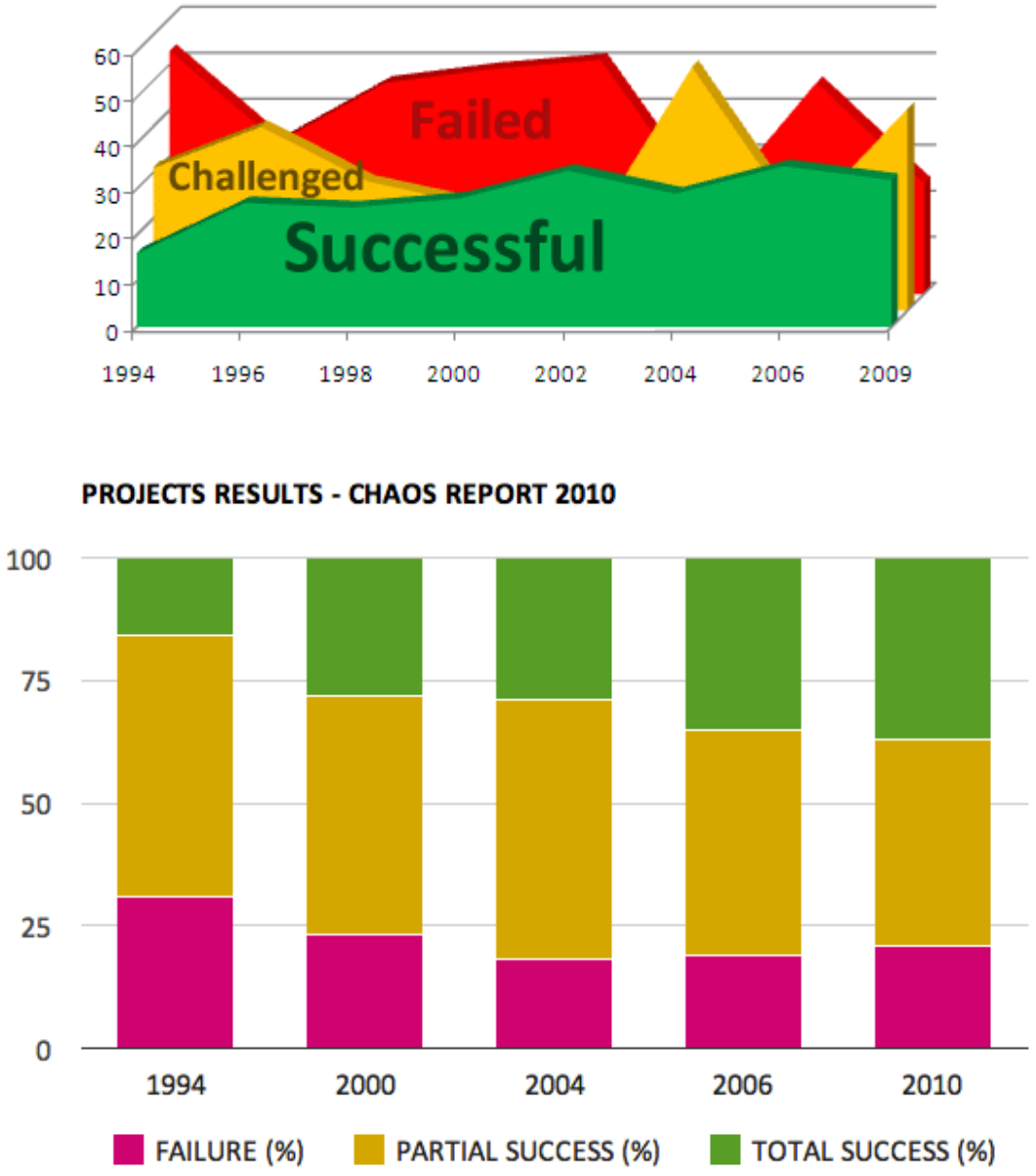


Figura 6 – Evolución de éxito en proyectos según Standish Group

Estos datos significan que los esfuerzos están dando resultado, y que poco a poco se ha ido mejorando, viendo una evolución lenta pero constante hacía mayores ratios de éxitos, pero aun así el número de proyectos no exitosos sigue siendo muy alto.

Este problema llevó a la crisis del software en los 60 y podríamos decir que continúa hasta hoy día, suponiendo perdidas de miles de millones de euros, no sólo en costes directos, sino también en otros como el coste de oportunidad perdido. Esto produce en el sector una sensación de que la crisis del software de los 60 es una enfermedad crónica.

Para busca las razones de tanto fracaso, a finales del año 2001, Daniel Piorum realizó un estudio con aproximadamente 50 responsables de proyectos, con el objeto de analizar las causas que alimentan los fracasos. En dicho estudio encontró tres principales causas que afectan los proyectos de forma negativa, estas son:

- 21 % Cambios en los objetivos definidos a nivel estratégico.
- 31 % No utilización, o mala utilización de metodologías de trabajo.
- 48 % Problemas humanos, de dirección, comunicación y conflictos entre las Personas.

Por su lado, en el informe de Standish, las diez principales causas de los fracasos son las siguientes (por orden de importancia):

1. Escasa participación de los usuarios
2. Requerimientos y especificaciones incompletas
3. Cambios frecuentes en los requerimientos
4. Falta de soporte ejecutivo
5. Incompetencia tecnológica
6. Falta de recursos
7. Expectativas no realistas
8. Objetivos poco claros
9. Cronogramas irreales
10. Nuevas tecnologías

Observemos que de los diez factores mencionados, siete están referidos a factores humanos (1 a 4 y 7 a 9). Como se puede apreciar, las causas que apunta el Standish Group son más detalladas, pero en su gran mayoría podrían ser clasificados en uno de los tres grupos presentados por Piorum. Y podemos ver claramente que los porcentajes más altos se encuentran enmarcados en los temas relacionados con el recurso humano.

Si pones de ejemplo la fabricación de un nuevo producto, gran parte del presupuesto se destina a la investigación y desarrollo del producto que culmina con una serie de modelos, diseños y prototipos, antes de que esté listo para la producción en masa, que evidentemente, tienen un costo para la organización. En cambio, cuando es un proyecto de software cuyo entregable es un software, que solo puede probarse plenamente cuando está terminado, la organización empuja directamente a la producción "sobre la marcha", utilizando, la mayoría de las veces, argumentos como "la falta de tiempo" o que "los detalles se resolverán más delante".

Hoy en día, las empresas están reconociendo gradualmente la necesidad de seguir metodologías claras, establecer rutinas, modos de actuar… y por eso cada vez es más común ver que buscan directores de proyectos profesionales. Pero aun así, aunque algunas metodologías cubren temas de comunicación, manejo de conflictos y negociación, muchas metodologías cometen el error de dar mucha más importancia al contenido herramental por encima del conceptual.

El reto de las nuevas metodologías que surjan es cubrir todos estos puntos, y quizás en ese apartado es donde los modelos ágiles estén sacando mayor ventaja a las tradiciones, y por eso están obteniendo mejores porcentajes de éxito y están ganando tanta fama y tienen tanta repercusión dentro del mundo del software.

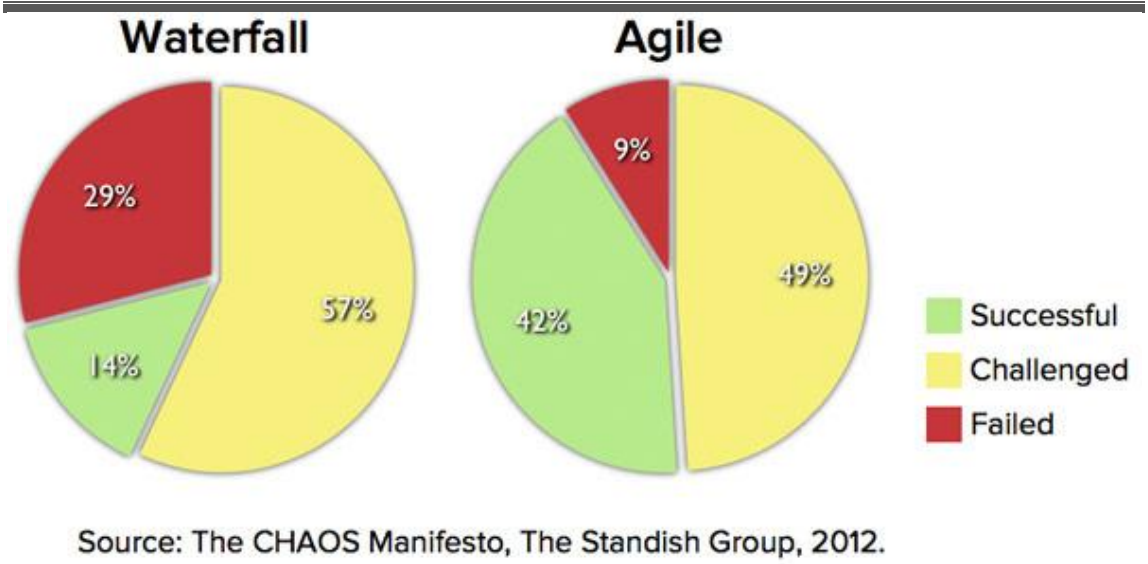


Figura 7 – Éxito de Modelo en Cascada y Ágil según el Standish Group.

El objetivo final para evitar estos fracasos sería lograr que la ingeniería del software fuera tan gestionable, predecible y repetible como las demás, pero hay que recordar que:

- El software "no se ve"
- Los productos software son muy complejos
- Es una disciplina aún joven
- No es, ni será nunca, una ciencia exacta
- Tiene un marcado carácter "sociológico"
- Los programadores no son intercambiables
- La productividad depende fuertemente de las personas.

3.2 Modelos Existentes

En este apartado se enumeran y explican los modelos de ciclo de vida más relevantes y con más impacto en el desarrollo software.

3.2.1 Modelo en Cascada

El **modelo en cascada**, algunas veces llamado el ciclo de vida clásico (porque constituye uno de los primeros modelos de ciclo de vida publicados), sugiere un enfoque sistemático y secuencial hacia el desarrollo del software, que se inicia con la especificación de requisitos del cliente y que continúa con el análisis, diseño, implementación y pruebas para culminar en el soporte del software terminado (mantenimiento)

Este enfoque de desarrollo del software fue introducido por Winston W. Royce en 1970, y se denomina ciclo de vida en "cascada" (aunque Royce no usó el término cascada en este artículo) por la disposición de las distintas fases de desarrollo, en las que los resultados de una fase parecen caer en cascada hacia la siguiente fase. Este modelo ordena rigurosamente las etapas del ciclo de vida del software, de forma que el inicio de cada etapa debe esperar a la

finalización de la inmediatamente anterior. Asociada con cada etapa del proceso existen hitos y documentos, de tal forma que se puede utilizar el modelo para comprobar los avances del proyecto y para estimar cuánto falta para su finalización.

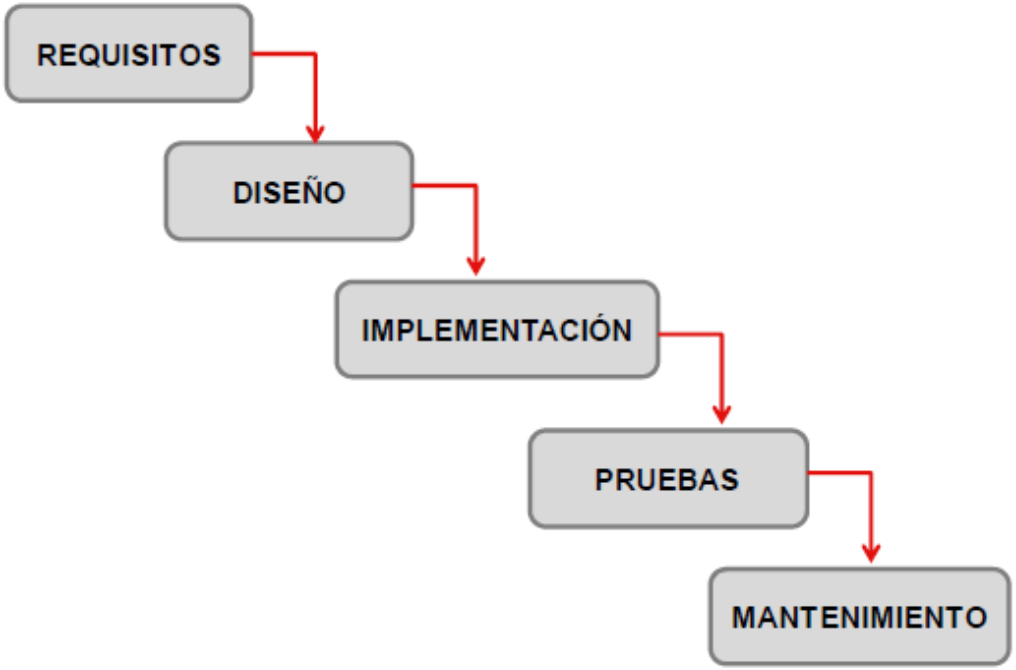


Figura 8 – Modelo de Ciclo de Vida en Cascada

Las principales etapas de este modelo según Sommerville (2005) son:

- **Análisis y definición de requisitos**. Los servicios, restricciones y metas del sistema se definen a partir de las consultas con los usuarios. Se define una especificación del sistema con un catálogo de requisitos, casos de uso, posible prototipado de pantallas e informes y puede que hasta una primera especificación del plan de pruebas.
- **Diseño del sistema y del software**. El proceso de diseño del sistema divide los requisitos en sistemas hardware o software. Establece una arquitectura completa del sistema. El diseño del software identifica y describe las abstracciones fundamentales del sistema software y sus relaciones.
- **Implementación y prueba de unidades**. Durante esta etapa, el diseño del software se lleva a cabo como un conjunto o unidades de programas.
- **Integración y prueba del sistema**. Los programas o las unidades individuales de programas se integran y prueban como un sistema completo para asegurar que se cumplan los requerimientos del software.
- **Funcionamiento y mantenimiento**. El sistema se instala y se pone en funcionamiento práctico. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores del ciclo de vida.

Y sus principios básicos se podrían resumir en:

- Planear un proyecto antes de embarcarse en él.
- Definir el comportamiento externo deseado del sistema antes de diseñar su arquitectura interna.

- Documentar los resultados de cada actividad.
- Diseñar un sistema antes de codificarlo.
- Testear un sistema después de construirlo.

A pesar de su antigüedad y de haber sido ampliamente criticado tanto desde el ambiente industrial como del académico, el ciclo de vida clásico se ha hecho con un lugar importante en el área de la Ingeniería del Software. Proporciona una guía de trabajo en la que se encuentran métodos para el análisis, diseño, codificación, pruebas y mantenimiento. El ciclo de vida en cascada sigue siendo el modelo de proceso más extensamente utilizado por los ingenieros del software, principalmente por su sencillez y facilidad de llevar a cabo. Pese a tener debilidades, es significativamente mejor que un enfoque arbitrario (como el de codificar y corregir) para el desarrollo del software. Muchos de los posteriores modelos de ciclo de vida son, en realidad, modificaciones sobre el modelo clásico, al que se le incorporan iteraciones o nuevas actividades.

Existen muchas variantes de este modelo. En respuesta a los problemas percibidos con el modelo en cascada puro, se introdujeron muchos modelos de cascada modificados. Estos modelos pueden solventar algunas de las críticas del modelo en cascada puro.

Una modificación típica sobre este modelo consiste en la introducción de una revisión y vuelta atrás, con el fin de corregir las deficiencias detectadas durante las distintas etapas, o para completar o aumentar las funcionalidades del sistema en desarrollo. De esta manera, durante cualquiera de las fases se puede retroceder momentáneamente a una fase previa para solucionar los problemas que se pudieran haber encontrado.

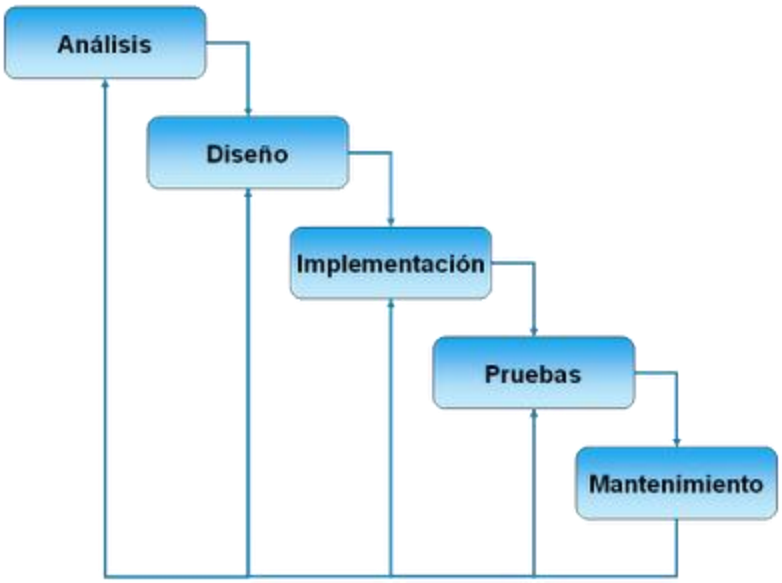


Figura 9 – Modelo de Ciclo de Vida en Cascada Modificado

Otra típica modificación del modelo en cascada es el modelo sashimi (que no creemos con suficiente entidad como para darle un apartado) donde las distintas etapas se solapan lo que implica que se puede actuar durante las etapas anteriores. Por ejemplo, ya que las fases de diseño e implementación se superpondrán en el modelo sashimi, los problemas de implementación se pueden descubrir durante las fases de diseño e implementación del

proceso de desarrollo. Esto ayuda a aliviar muchos de los problemas asociadas con la filosofía del modelo en cascada.

No se habla del **modelo de desarrollo rápido de aplicaciones** porque no es más que una versión a "alta velocidad" del modelo en cascada.

3.2.2 Modelo en V

Este ciclo de vida fue diseñado por Alan Davis, y contiene las mismas etapas que el ciclo de vida en cascada puro pero a diferencia de aquel, busca hacer la actividad de pruebas más efectiva y productiva, mediante la elaboración de los planes (y casos de prueba) a medida que se avanza en el desarrollo del proyecto

El modelo en v se desarrolló para terminar con algunos de los problemas que se vieron utilizando el enfoque de cascada tradicional. Los defectos estaban siendo encontrados demasiado tarde en el ciclo de vida, ya que las pruebas no se introducían hasta el final del proyecto.

El modelo en v dice que las pruebas necesitan empezarse lo más pronto posible en el ciclo de vida y estas actividades deberían ser llevadas a cabo en paralelo con las actividades de desarrollo. Los testers necesitan trabajar con los desarrolladores y analistas de tal forma que puedan realizar estas actividades y tareas y producir una serie de entregables de pruebas.

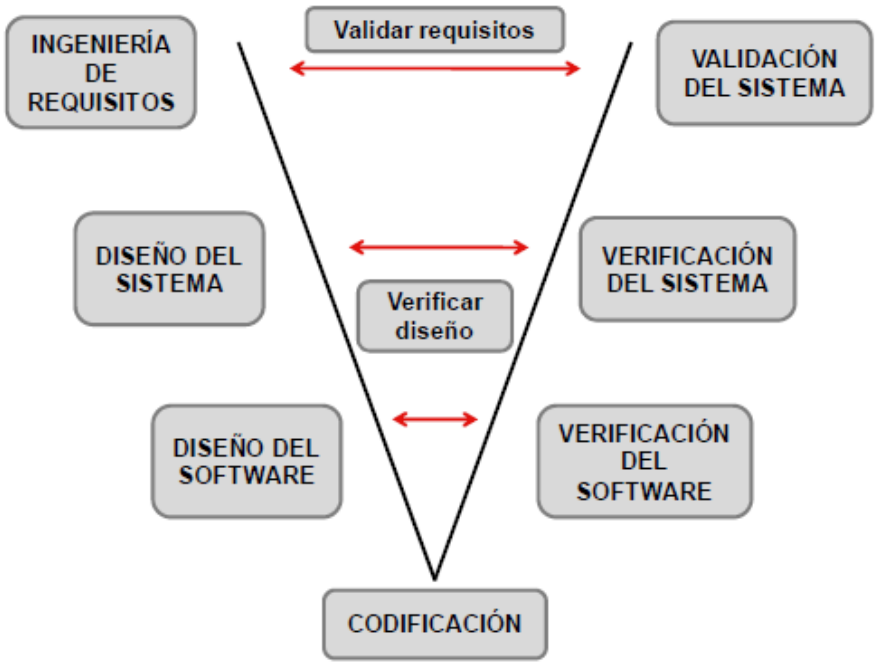


Figura 10 – Modelo de Ciclo de Vida en V

La parte izquierda de la v representa la descomposición de los requisitos y la creación de las especificaciones del sistema. El lado derecho de la v representa la integración de partes y su verificación. V significa "Validación y Verificación". La razón de su forma es que para cada una de las fases de diseño se ha encontrado que hay un homólogo en las fases de pruebas que se correlacionan.

El modelo en V es una variación del modelo en cascada que muestra cómo se relacionan las actividades de prueba con el análisis y el diseño. Como se muestra en la anterior figura, la codificación forma el vértice de la V, con el análisis y el diseño a la izquierda y las pruebas y el mantenimiento a la derecha. Una fase además de utilizarse cómo entrada para la siguiente, sirve para validar o verificar otras fases posteriores.

En la siguiente imagen se puede ver este modelo con mayor detalle:

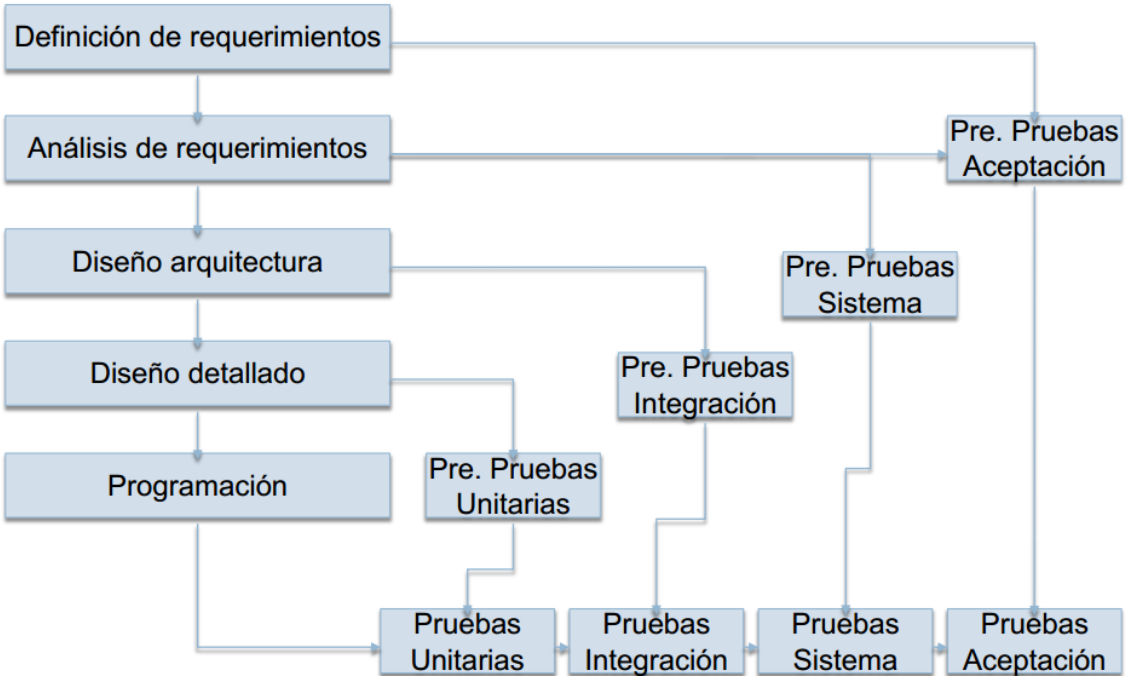


Figura 11 – Modelo de Ciclo de Vida en V detallado

3.2.3 Modelo Iterativo

También derivado del ciclo en cascada puro, este modelo busca reducir el riesgo que surge entre las necesidades del usuario y el producto final por malos entendidos durante la etapa de recogida de requisitos.

Consiste en la **iteración** de varios ciclos de vida en cascada. Al final de cada iteración se le entrega al cliente una versión mejorada o con mayores funcionalidades del producto. El cliente es quien después de cada iteración evalúa el producto y lo corrige o propone mejoras. Estas iteraciones se repetirán hasta obtener un producto que satisfaga las necesidades del cliente.

Comentar que no se explica el **modelo de prototipos** porque se considera una posible fase del iterativo, donde en las etapas iniciales se realizan una serie de prototipos desechables hasta tener claro un mínimo de los requisitos de la aplicación o producto a desarrollar. Ni tampoco se trata el **modelo evolutivo** por razones similares.

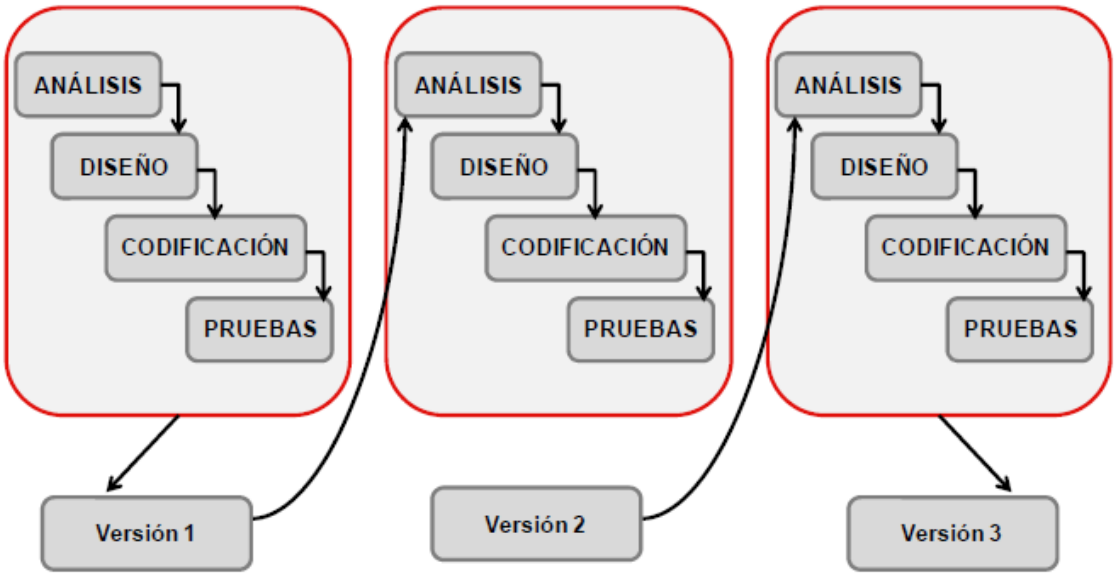


Figura 12 – Modelo de Ciclo de Vida Iterativo

Este modelo se suele utilizar en proyectos en los que los requisitos no están claros por parte del usuario, por lo que se hace necesaria la creación de distintos prototipos para presentarlos y conseguir la conformidad del cliente. Cada iteración es un mini proyecto en cascada auto contenido compuesto de actividades como análisis de requerimientos, diseño, programación y pruebas. Las primeras versiones pueden ser prototipos que se desechan posteriormente.

3.2.4 Modelo Incremental

El modelo incremental se basa en la filosofía de construir incrementando las funcionalidades del programa. Se realiza construyendo por módulos que cumplen las diferentes funciones del sistema. Esto permite aumentar gradualmente las capacidades del software. Facilita el desarrollo, permitiendo a cada miembro del equipo desarrollar un módulo particular (en caso de que sea realizado por un equipo de programadores).

Es similar al modelo con iteraciones, aplicándose un ciclo en cada nueva funcionalidad del programa. Al final de cada ciclo, se le entrega al cliente la versión que contiene la nueva funcionalidad, así se mantiene al cliente en constante contacto con los resultados obtenidos en cada incremento.

Cuando se utiliza un modelo incremental, el primer incremento es a menudo un producto esencial, sólo con los requisitos básicos, pero muchas características suplementarias (algunas conocidas, otras no) no se incorporan. Este modelo se centra en la entrega de un producto operativo con cada incremento. Los primeros incrementos son versiones incompletas del producto final, pero proporcionan al usuario la funcionalidad que precisa y también una plataforma para la evaluación. Como resultado de la evaluación se desarrolla un plan para el incremento siguiente. El plan afronta la modificación del producto esencial con el fin de satisfacer de mejor manera las necesidades del cliente y la entrega de características y funcionalidades adicionales. Este proceso se repite después de la entrega de cada incremento mientras no se haya elaborado el producto completo.

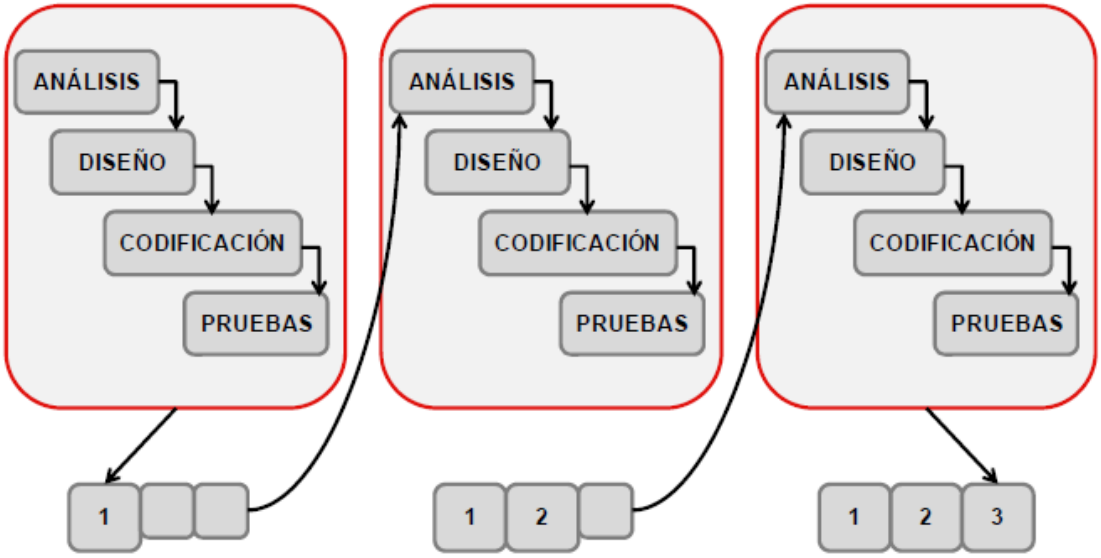


Figura 13 – Modelo de ciclo de vida Incremental

Por ejemplo, un software de procesamiento de texto, desarrollado con el paradigma incremental, en su primer incremento, podría realizar funciones básicas de administración de archivos, edición y creación de documentos; en el segundo incremento, ediciones más sofisticadas, y tendría funciones más complejas de producción de documentos; en el tercer incremento, funciones de corrección ortográfica y gramatical; y en el cuarto, capacidades avanzadas de configuración de página.

Por tanto, el modelo de proceso incremental se asemeja mucho al iterativo, pero a diferencia de este, el modelo incremental se enfoca en la entrega de un producto completamente operacional con cada incremento. Los primeros incrementos son versiones incompletas del producto final, pero proporcionan al usuario la funcionalidad que necesita y una plataforma para evaluarlo.

Figura 14 – Ejemplo comparativo de iterativo (izquierda) e incremental (derecha)

Es decir, en el iterativo el objetivo es poder avanzar en la creación de un software del que no se tienen claros sus requisitos (por ejemplo no se sabe si se quiere que en el cuadro se vean anillos en los dedos), existe incertidumbre en los requisitos. Y sin embargo en el incremental el objetivo es poder abordar un proyecto en base a pequeñas entregar funcionales para que permita abarcar más fácilmente el problema, el cual es conocido, pero se ha decidido dividirlo.

Como resumen entonces se podría decir que un modelo incremental lleva a pensar en un desarrollo modular, con entregas parciales del producto software denominados “incrementos” del sistema, que son escogidos en base a prioridades predefinidas de algún modo. El modelo permite una implementación con refinamientos sucesivos (ampliación y/o mejora). Con cada

incremento se agrega nueva funcionalidad o se cubren nuevos requisitos o bien se mejora la versión previamente implementada del producto software.

3.2.5 Modelo en Espiral

Este modelo fue propuesto por Boehm en 1988 en su artículo "A Spiral Model of Software Development and Enhancement". El modelo se basa en una serie de ciclos repetitivos para ir ganando madurez en el producto final, hasta lograr el objetivo deseado.

No representa al proceso del software como una secuencia de actividades, sino como una espiral. Cada ciclo en la espiral representa una fase del proceso del software. Así el ciclo más interno puede aludir a la Viabilidad del Sistema, el siguiente ciclo a la Definición de Requerimientos, el siguiente ciclo al Diseño del Sistema, y así sucesivamente.

Toma los beneficios de los modelos incremental y por prototipos (derivado del iterativo), pero se tiene más en cuenta el concepto de riesgo que aparece debido a las incertidumbres e ignorancias de los requerimiento proporcionados al principio del proyecto o que surgirán durante el desarrollo. Este sistema es muy utilizado en proyectos grandes y complejos como puede ser, por ejemplo, la creación de un sistema operativo.

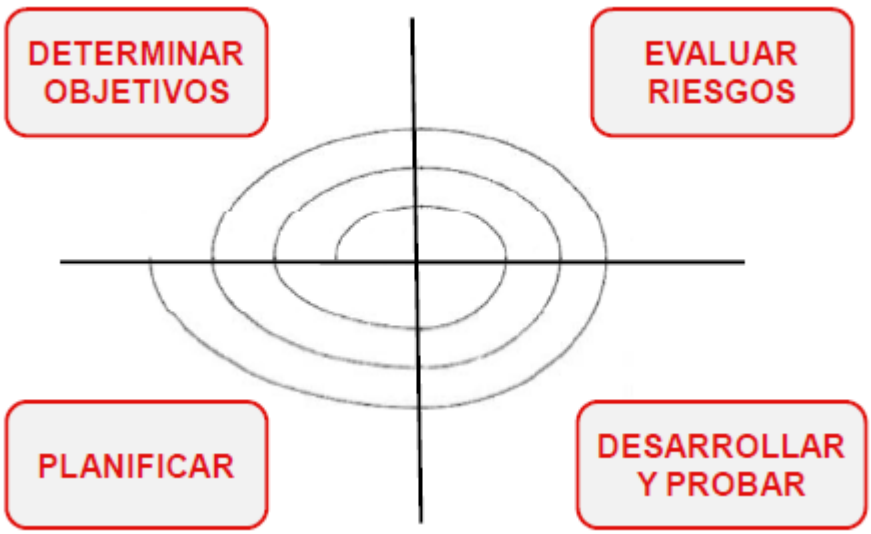


Figura 15 – Modelo de Ciclo de Vida en Espiral

Como ya se ha dicho, este modelo de ciclo de vida en espiral tiene en cuenta fuertemente el riesgo que aparece a la hora de desarrollar software. Para ello, se comienza mirando las posibles alternativas de desarrollo, se opta por la de riesgos más asumibles y se hace un ciclo de la espiral. Si el cliente quiere seguir haciendo mejoras en el software, se vuelven a evaluar las nuevas alternativas y riesgos y se realiza otra vuelta de la espiral, así hasta que llegue un momento en el que el producto software desarrollado sea aceptado y no necesite seguir mejorándose con otro nuevo ciclo.

El porqué de dar tanta importancia a los riesgos, es que una mala gestión de estos origina importantes problemas en el proyecto, que normalmente conllevan pérdidas de tiempo y dinero. Por eso la disminución de riesgos es una actividad clave en la gestión del proyecto. Al

ser un modelo de ciclo de vida orientado a la gestión de riesgos se dice que uno de los aspectos fundamentales de su éxito radica en que el equipo que lo aplique tenga la necesaria experiencia y habilidad para detectar y catalogar correctamente riesgos.

El modelo en espiral se divide en un número de actividades estructurales, también llamadas regiones de tareas, según Sommerville (2005) el ciclo de vida del modelo en espiral se divide cuatro sectores:

1. **Determinar o fijar objetivos**:
    a. Fijar los productos a obtener: requerimientos, especificación, manual de usuario.
    b. Las restricciones.
    c. Identificación de riesgos del proyecto y estrategias alternativas para evitarlos.
    d. Hay una cosa que solo se hace una vez: planificación inicial o previa.
2. **Análisis del riesgo**:
    a. Se estudian todos los riesgos potenciales y se seleccionan una o varias alternativas propuestas para reducir o eliminar los riesgos.
    b. Se definen los pasos para reducir dichos riesgo.
3. **Desarrollar, verificar y validar**:
    a. Dependiendo del resultado de la evaluación de riesgos, se elige un modelo para el desarrollo, que puede ser cualquiera de los otros existentes, como formal, evolutivo, cascada, etc. Así, por ejemplo, si los riesgos de la interfaz de usuario son dominantes, un modelo de desarrollo apropiado podría ser la construcción de prototipos evolutivos. Si el mayor riesgo es la integración de los subsistemas Modelo en Cascada, etc.
    b. Tareas de la actividad propia y de prueba.
    c. Análisis de alternativas e identificación de resolución de riesgos.
4. **Planificar**:
    a. Revisamos todo lo que hemos hecho, evaluándolo y con ello decidimos si continuamos con un nuevo ciclo y planificamos la próxima actividad.

La distinción más destacada entre el modelo en espiral y otros modelos de software es la tarea explícita de evaluación de riesgos. Aunque la gestión de riesgos es parte de otros procesos también, no tiene una representación propia en el modelo de proceso. Para otros modelos la evaluación de riesgos es una subtarea, por ejemplo, de la planificación y gestión global. Además no hay fases fijadas para la especificación de requisitos, diseño y pruebas en el modelo en espiral. Se puede usar prototipado para encontrar y definir los requisitos.

La diferencia entre este modelo y el modelo de ciclo incremental es que en el incremental se parte de que no hay incertidumbre en los requisitos iniciales; en este, en cambio, se es consciente de que se comienza con un alto grado de incertidumbre. En el incremental suponemos que conocemos el problema y lo dividimos. Este modelo gestiona la incertidumbre.

3.2.6 Modelo Ágil

El modelo ágil es un conjunto de métodos de ingeniería del software, que se basan en el desarrollo iterativo e incremental, teniendo presente los cambios y respondiendo a estos mediante la colaboración de un grupo de desarrolladores auto-organizados y multidisciplinares.

Figura 16 – Esquema simplificado del Proceso Iterativo Incremental

Este modelo promueve un proceso de gestión de proyectos que fomenta el trabajo en equipo, la organización y responsabilidad propia, un conjunto de mejores prácticas de ingeniería que permiten la entrega rápida de software de alta calidad, y un enfoque de negocio que alinea el desarrollo con las necesidades del cliente y los objetivos de la compañía.

La definición moderna del desarrollo de software ágil se desarrolló hacia la mitad de los 90 como parte de una reacción contra los métodos denominados “pesados”, como el modelo en cascada. El proceso originado por el uso del modelo en cascada se veía como burocrático, lento, degradante e inconsistente con las formas en las que los desarrolladores de software realizaban trabajo efectivo. Inicialmente, a los métodos ágiles se les llamó métodos ligeros.

En 2001, 17 figuras destacadas en el campo del desarrollo ágil (llamado entonces metodologías de peso ligero) se juntaron en la estación de esquí Snowbird en Utah para tratar el tema de la unificación de sus metodologías. Crearon el manifestó ágil, ampliamente considerado como la definición canónica del desarrollo ágil.

Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto. Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Figura 17 – Reunión de creación de "The Agile Alliance"

Tras esta reunión se creó "The Agile Alliance", una organización sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida fue el Manifiesto Ágil, documento que resume la filosofía "ágil".

En el manifiesto ágil se indican unos puntos que tienen que ser valorados a la hora de utilizar una metodología ágil.

El individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas

El punto clave para el desarrollo de software son las personas. Son ellas las que van a escribir el software, por lo que es más importante centrarse primero en conseguir un buen equipo que será el encargado de llevar el desarrollo a buen puerto. A veces a la hora de iniciar un proyecto de software se le da más importancia a las herramientas y al entorno que a los propios desarrolladores. Este entorno se crea primero, luego se busca a la gente y se le pide que se adapte a él. Es mucho mejor buscar primero a la gente, crear un equipo y luego permitir a este equipo que cree su propio entorno y trabaje sobre él.

Crear software que funcione es más importante que una buena documentación

Puede que la documentación sea algo importante en el desarrollo de software. Pero es importante no olvidarse que estamos desarrollando software, no generando documentación. La mejor documentación del mundo no sirve de nada si el software del que trata no ha sido desarrollado. Una de las normas de las metodologías ágiles es generar la documentación solo cuando es estrictamente necesario para su uso de forma inmediata, especialmente esa documentación interna dedicada a pasar información entre diferente gente del proyecto.

El cliente debe de estar presente en el desarrollo de software

Por muy bueno que sea un contrato donde se especifique que tiene que hacer el software, el cliente es el que más rápido puede identificar si ese software está haciendo lo que él necesita o no. Igual que en el caso de la documentación, recordemos que es lo que estamos haciendo. De nada sirve un software muy bueno, optimizado y con gran funcionalidad si esa funcionalidad no es la que necesita el cliente. Desarrollar algo que el cliente no necesita es una pérdida de tiempo y un gasto que éste difícilmente estará dispuesto a asumir.

Las cosas cambian

Durante el proyecto de desarrollo de software pueden cambiar muchas cosas. Pueden cambiar los requisitos, puede cambiar la tecnología, puede cambiar el equipo que trabaja en él e incluso puede cambiar el cliente. Ser capaz de adaptarse al cambio puede ser un factor clave a la hora de que un proyecto sea un fracaso o un éxito. El tiempo dedicado a una planificación exhaustiva que al final puede no ser útil es tiempo que podría ser invertido en el desarrollo del proyecto.

Siguiendo los valores anteriormente comentados se definieron doce principios. Estos principios son los que diferencian un proceso ágil de un proceso tradicional. Los dos primeros resumen en cierta medida la filosofía del desarrollo ágil, mientras que el resto tienen más relación con el proceso.

1. La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que aporten valor.
2. Los cambios son bienvenidos. Aceptar el cambio es una ventaja competitiva para el cliente.
3. Entregar frecuentemente software que funcione. El periodo puede variar entre unas semanas y unos meses, pero cuanto menor sea el intervalo mejor.
4. La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.
5. Construcción del proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y procurándoles la confianza para finalizar el trabajo.
6. El dialogo cara a cara es la mejor forma para trasmitir información dentro de un equipo de desarrollo.
7. El software que funciona es la principal medida del progreso.
8. Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían mantener un ritmo constante de forma indefinida.
9. La atención continua a la calidad técnica y al buen diseño mejora la agilidad.
10. La simplicidad como arte de maximizar la cantidad de trabajo que no es necesario hacer.
11. Las mejores arquitecturas, requisitos y diseño vienen de equipos que se auto-organizan.
12. En intervalos regulares, el equipo reflexiona respecto a cómo ser más efectivo y actúa en consecuencia.

Como se decía al inicio, este modelo sigue un desarrollo iterativo incremental, esto quiere decir que el proyecto se planifica en diversos bloques temporales llamados iteraciones. Las iteraciones se pueden entender como miniproyectos: en todas las iteraciones se repite un proceso de trabajo similar para proporcionar un resultado completo sobre el producto final, de manera que el cliente pueda obtener los beneficios del proyecto de forma incremental. Para ello, cada requisito se debe completar en una única iteración: el equipo debe realizar todas las tareas necesarias para completarlo (incluyendo pruebas y documentación) y que esté preparado para ser entregado al cliente con el mínimo esfuerzo necesario. De esta manera no se deja para el final del proyecto ninguna actividad arriesgada relacionada con la entrega de requisitos.

En cada iteración el equipo evoluciona el producto (hace una entrega incremental) a partir de los resultados completados en las iteraciones anteriores, añadiendo nuevos objetivos/requisitos o mejorando los que ya fueron completados. Un aspecto fundamental para guiar el desarrollo iterativo e incremental es la priorización de los objetivos/requisitos en función del valor que aportan al cliente.

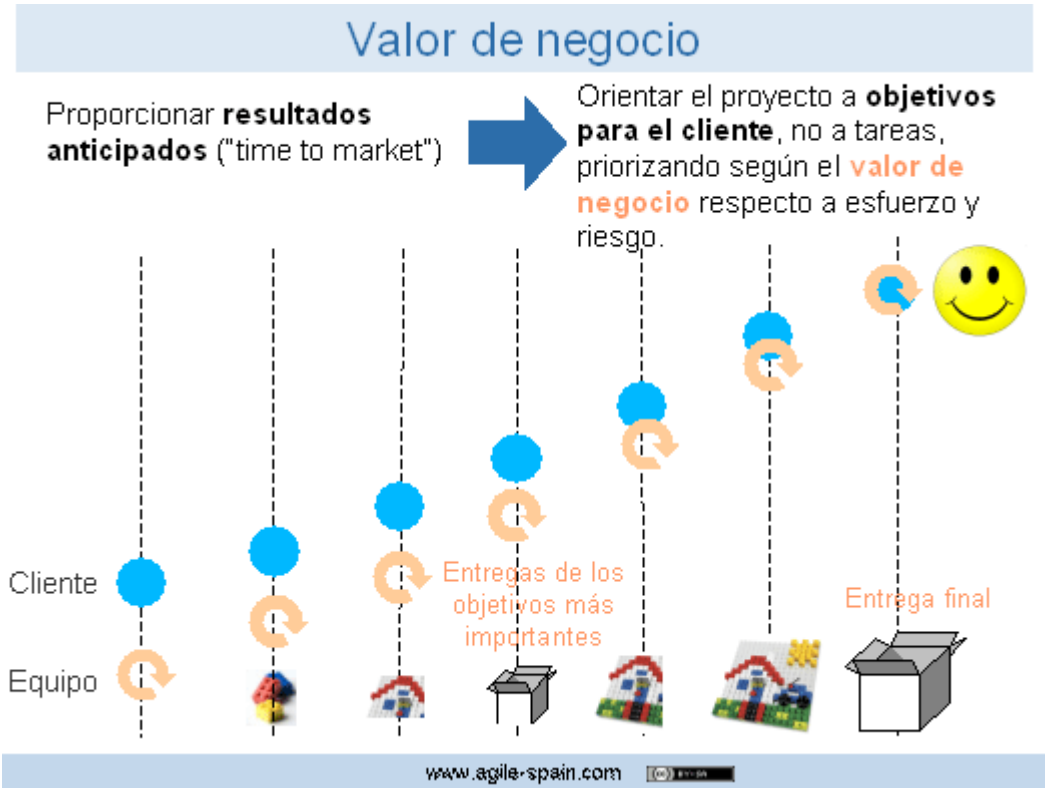


Figura 18 – Evolución del valor de negocio usando el modelo ágil

Es importante recordar que pese a que los principios ágiles se llevan bien con enfoques iterativos e incrementales, por eso su instrumentación en metodologías siguen este tipo de ciclo de vida, la simple aplicación de la misma no quiere decir que el desarrollo sea ágil ya que lo único que hace es fragmentar el proyecto en diferentes entregas. Si el enfoque no es ágil, la aplicación del ciclo de vida iterativo incremental tampoco lo será.

Con todo esto podríamos resumir que las características básicas de los proyectos gestionados con un modelo ágil son:

- **Incertidumbre**: La dirección indica la necesidad estratégica que se desea cubrir (sin entrar en detalles), ofreciendo máxima libertad al equipo de trabajo.
- **Equipos auto-organizados**: No existen roles especializados
- **Autonomía**: Libertad para la toma de decisiones.
- **Auto-superación**: De forma periódica se evalúa el producto que se está desarrollando.
- **Auto-enriquecimiento**: Transferencia del conocimiento.
- **Fases de desarrollo solapadas**: Las fases no existen como tal sino que se desarrollan tareas/actividades en función de las necesidades cambiantes durante todo el proyecto. De hecho, en muchas ocasiones no es posible realizar un diseño técnico detallado antes de empezar a desarrollar y ver algunos resultados. Por otra parte, las fases tradicionales efectuadas por personas diferentes no favorece el trabajo en

equipo y pueden llegar a generar más inconvenientes que ventajas (p.ej. un retraso en una fase, afecta a todo el proyecto).

- **Control sutil**: Establecimientos de puntos de control para realizar un seguimiento adecuado sin limitar la libertad y creatividad del equipo. Así mismo, se recomienda:
    - **Evaluar el ambiente laboral**, siendo fundamental la elección de personas que no generen conflictos.
    - **Reconocer los méritos** mediante un sistema de evaluación justo y entender los errores como puntos de mejora y aprendizaje.
    - **Potenciar la interacción** entre el equipo y el negocio, para que puedan conocer las necesidades de primera mano.
    - **Difusión y transferencia del conocimiento**: alta rotación de los miembros de los equipos entre diferentes proyectos. Por otra parte, potenciar el acceso libre a la información y documentación.

Los proyectos gestionados con metodologías ágiles se inician sin un detalle cerrado de lo que va a ser construido. A nivel comercial, los proyectos pueden ser vendidos como servicios y no como productos.

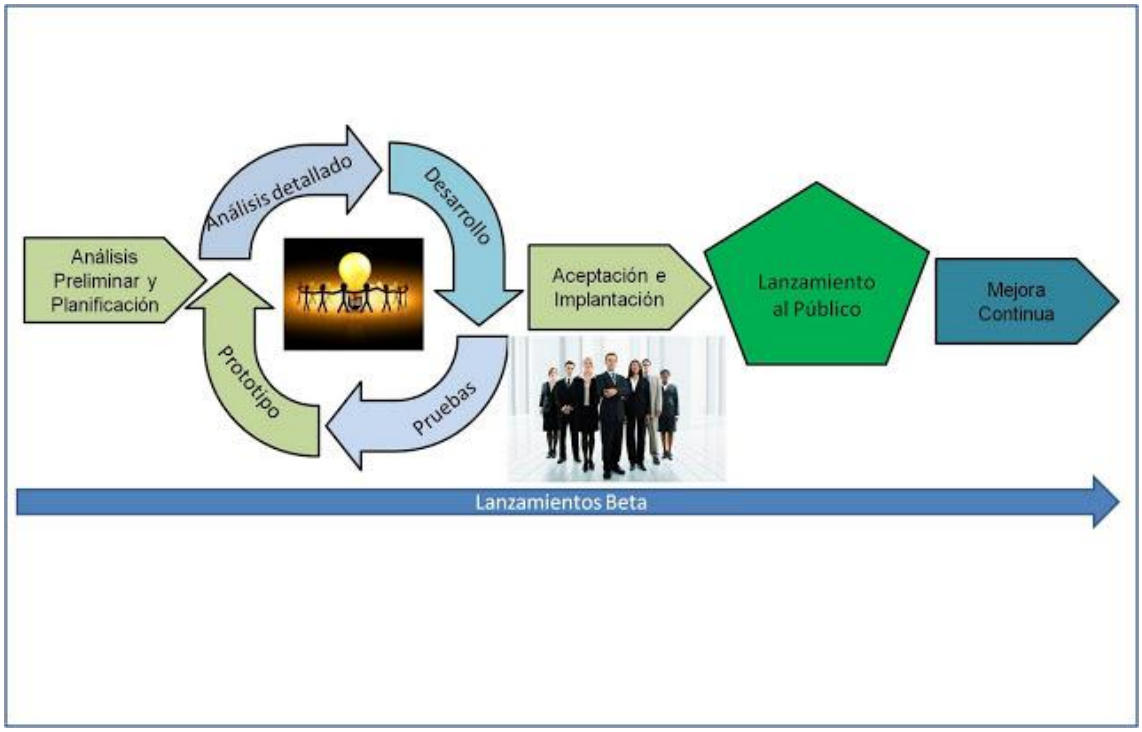


Figura 19 – Proceso de desarrollo de sistema bajo modelo ágil

3.3 Metodologías Ágiles

A continuación se describen brevemente algunas de las metodologías ágiles más destacadas.

3.3.1 Adaptive Software Development

La técnica de Adaptive software Development (ASD) fue desarrollada por Jim Highsmith y Sam Bayer a comienzos de 1990 y publicado en el año 2000. Esta metodología se adapta al cambio en lugar de luchar contra él. Se basa en la adaptación continua a circunstancias cambiantes. En ella no hay un ciclo de planificación-diseño-construcción del software, sino un ciclo especular colaborar-aprender.

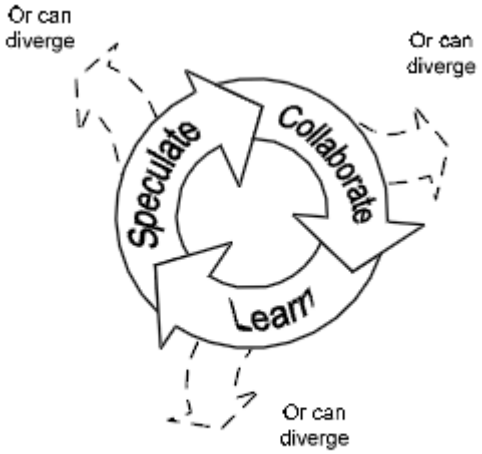


Figura 20 – Ciclo del ASD

Esta metodología hace énfasis en aplicar las ideas que se originaron en el mundo de los sistemas complejos, que giraban en torno a la adaptación continua del proceso al trabajo.

Sus principales características del son:

- Iterativo.
- Orientado a los componentes de software (la funcionalidad que el producto va a tener, características, etc.) más que a las tareas en las que se va a alcanzar dicho objetivo.
- Tolerante a los cambios.
- Guiado por los riesgos
- La revisión de los componentes sirve para aprender de los errores y volver a iniciar el ciclo de desarrollo

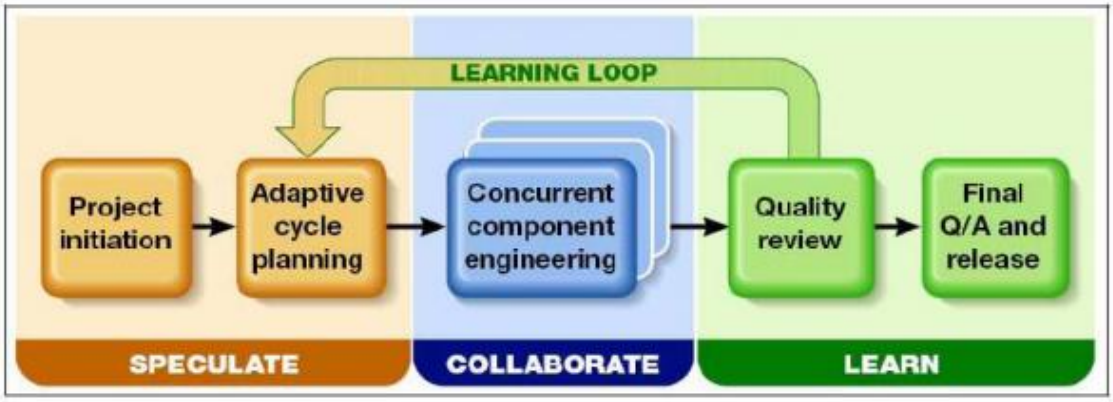


Figura 21 – Actividades del Adaptaive Software Development (ASD)

ASD utiliza un "cambio orientado hacia el ciclo de vida", que tiene tres fases que son: especular colaborar y aprender.

3.3.1.1 Especular

Las tareas de esta fase son;

1. Misión del Proyecto: Una primera fase de iniciación para establecer los principales objetivos y metas del proyecto en su conjunto y comprender las limitaciones (zonas de riesgo) con las que operará el proyecto.
2. Fijación del marco temporal del proyecto: En ASD se realizan estimaciones de tiempo sabiendo que pueden sufrir desviaciones. Sin embargo, estas son necesarias para la correcta atención de los trabajadores que se mueven dentro de plazos de forma que puedan priorizar sus tareas.
3. Determinar número de iteraciones y la duración de cada una.
4. Definición de objetivos de cada iteración.
5. Asignar la funcionalidad de cada iteración.

Estos pasos se pueden volver a examinar varias veces antes de que el equipo y los clientes estén satisfechos con el resultado.

3.3.1.2 Colaborar

Desarrollo concurrente del trabajo de construcción y gestión del producto.

En esta fase del ciclo son revisados a fondo los requerimientos del proyecto. Se define cómo se va a trabajar de acuerdo a las habilidades de los integrantes del grupo.

Es importante que en la fase de colaboración cada uno de los integrantes:

- Realice críticas constructivas del trabajo de los demás en forma anónima.
- Tomar estas críticas para mejorar y no generar resentimientos dentro del grupo.
- Arduo trabajo de los integrantes.
- Tener la habilidad de trabajar en grupo y de la mano de los demás integrantes.
- Comunicar los problemas o preocupaciones que se tiene.

3.3.1.3 Aprender

La idea de ASD es el aprendizaje continuo en cada ciclo. Es un elemento crítico para la eficacia de los equipos. En cada iteración se revisa:

1. **Calidad del producto desde un punto de vista del cliente**. Es la única medida legítima de éxito, pero además, dentro de las metodologías ágiles, los clientes tienen un valor importante.
2. **Calidad del producto desde un punto de vista de los desarrolladores**. Se trata de la evaluación de la calidad de los productos desde un punto de vista técnico. Ejemplos de esto incluyen la adhesión a las normas y objetivos conforme a la arquitectura.
3. **La gestión del rendimiento**. Este es un proceso de evaluación para ver lo que se ha aprendido mediante el empleo de los procesos utilizados por el equipo.
4. **Situación del proyecto**. Como paso previo a la planificación de la siguiente iteración del proyecto, es el punto de partida para la construcción de la siguiente serie de características.

3.3.2 Agile Unified Process

AUP es una versión simplificada de RUP (Rational Unified Process), el cual fue desarrollado por IBM. Describe una manera simple de entender el desarrollo de aplicaciones de negocio usando técnicas ágiles y conceptos heredados del RUP. Sus creadores han tratado de mantenerlo lo más simple posible. El enfoque usa técnicas ágiles incluyendo desarrollo orientado a pruebas, modelado ágil, gestión de cambios ágil y refactorización de bases de datos para mejorar la productividad.

En los proyectos que usan AUP, normalmente se entregan versiones de desarrollo al final de cada iteración. Una versión de desarrollo de una aplicación es una versión que potencialmente puede ser lanzada en producción si pasa la garantía de calidad de pre-producción, supera la fase de pruebas y los procesos de despliegue. En un desarrollo AUP, normalmente la primera versión de producción tarda más que las demás, desarrollándose antes más versiones de desarrollo, pero en las siguientes iteraciones tardará menos en desarrollarse una versión de producción. Un enfoque en las tareas de despliegue ayuda a evitar problemas, y permite aprender de la experiencia a lo largo del desarrollo.

El ciclo de vida de AUP consta de 4 fases: Inicio, Elaboración, Construcción y Transición. Además existen 7 disciplinas que trabajan en cada fase del ciclo de vida.

3.3.2.1 Fases

- **Inicio** → El objetivo es identificar el alcance inicial del proyecto, una arquitectura potencial para el sistema y obtener fondos y aceptación por parte de las personas involucradas en el negocio.
- **Elaboración** → El objetivo es probar la arquitectura del sistema.

- **Construcción** → El objetivo es construir software operativo de forma incremental que cumpla con las necesidades de prioridad más altas de las personas involucradas en el negocio.
- **Transición** → El objetivo es validar y desplegar el sistema en el entorno de producción.

3.3.2.2 Disciplinas

- **Modelado** → Abarca las disciplinas de Modelado de negocio, ingeniería de requisitos, análisis y diseño. El objetivo es entender la organización y el dominio del problema, así como encontrar una solución a éste.
- **Implementación** → El objetivo es transformar el modelo a código ejecutable y realizar pruebas de nivel básico, en particular pruebas unitarias.
- **Test** → El principal objetivo es realizar una evaluación objetiva para asegurar la calidad. Esto incluye encontrar defectos, validar que el sistema funciona como se ha diseñado y verificar que los requisitos se cumplen.
- **Despliegue** → El objetivo del despliegue es planear el desarrollo para la entrega del sistema, y ejecutar este plan para hacer que el sistema esté disponible para el usuario final.
- **Gestión de la configuración** → El objetivo es gestionar el acceso a los elementos del proyecto. Esto incluye gestionar y controlar los cambios que se hagan a estos elementos.
- **Gestión del proyecto** → El objetivo es dirigir las actividades que tienen lugar en el proyecto. Manejar riesgos, dirigir a las personas involucradas (asignando tareas...) y coordinar a las personas y los sistemas involucrados para hacer que el producto esté disponible para su entrega a tiempo.
- **Gestión del entorno** → El objetivo es asegurarse de que los procesos adecuados, guías y estándares, y herramientas, están disponibles para el equipo.

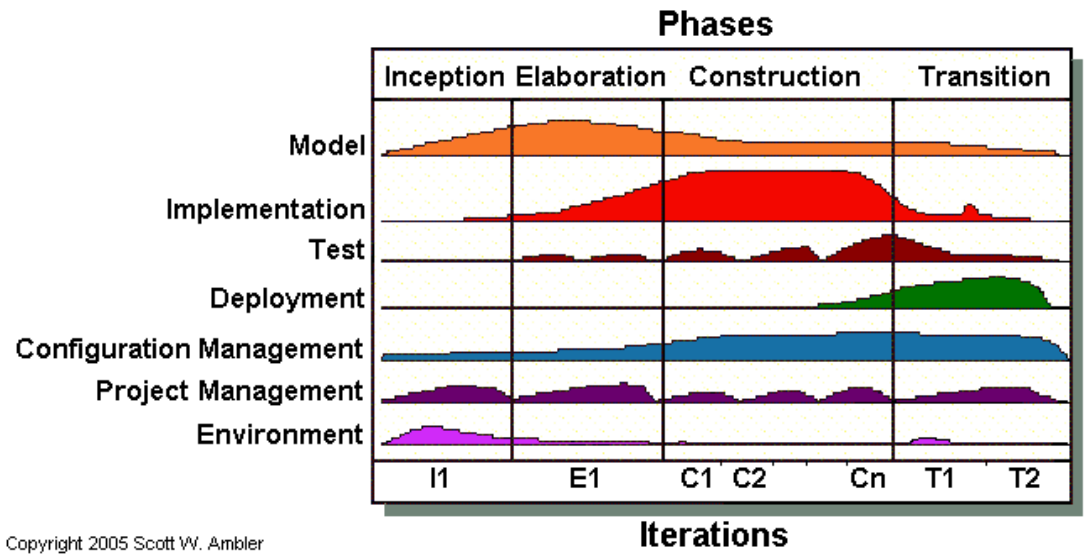


Figura 22 - Carga de trabajo de cada disciplina en cada fase AUP

3.3.2.3 Principios

AUP se basa en las siguientes filosofías:

1. **Los empleados saben lo que están haciendo**. La gente no va a leer documentación del proceso detallada, pero quieren algo de orientación a alto nivel y/o formación de vez en cuando. El producto AUP proporciona enlaces a muchos de los detalles pero no fuerza a ellos.
2. **Simplicidad**. Todo está descrito de forma concisa.
3. **Agilidad**. AUP se ajusta a los valores y principios de desarrollo de software ágil y la Alianza Ágil
4. **Foco en las actividades de alto valor**. El foco está en las actividades que realmente cuentan, no en todas las posibles cosas que pudieran pasar en un proyecto.
5. **Independencia de herramientas**. Se puede usar cualquier conjunto de herramientas. La recomendación es que se usen las herramientas que mejor se adapten al trabajo, que son con frecuencia herramientas simples.
6. Habrá que **adaptar** AUP para cumplir con las necesidades propias.

3.3.3 Crystal

Concebido por Alistair Cockburn (uno de los firmantes del manifiesto ágil), este modelo no describe una metodología cerrada, sino un conjunto de ellas, junto con los criterios para seleccionar y adecuar la más apropiada al proyecto.

El nombre de metodologías Crystal viene de que cada proyecto software puede caracterizarse según dos dimensiones, tamaño o dimensión (número de personas en el proyecto) y criticidad (consecuencia de los errores), al igual que los minerales se caracterizan por dos dimensiones, color y dureza. Y esta es una de las bases de las metodologías Crystal.

Cockburn en las conclusiones de uno de sus artículos, llamado la “Cockburn Scale”, rompe con la antigua idea, que aún se mantiene en muchas empresas, de que existen metodologías de libro que se implantan directamente y por completo. Así expone que no existe una metodología de desarrollo software única, mejor y universal, sino que existe una mejor metodología por tipo de proyecto. Las variables clave a la hora de seleccionar una metodología son el tamaño del equipo, su distribución, la criticidad del proyecto y las prioridades.

La criticidad se puede clasificar en:

- Pérdida de confort o usabilidad
- Pérdidas económicas moderadas
- Pérdidas económicas graves
- Pérdida de vidas humanas

La dimensión, según el número de personas involucradas:

- Clear, para equipos de hasta 8 personas o menos.
- Amarillo, de entre 10 y 20 personas.
- Naranja, para equipos entre 20 y 50 personas.

- Roja, entre 50 y 100 personas.
- etc.

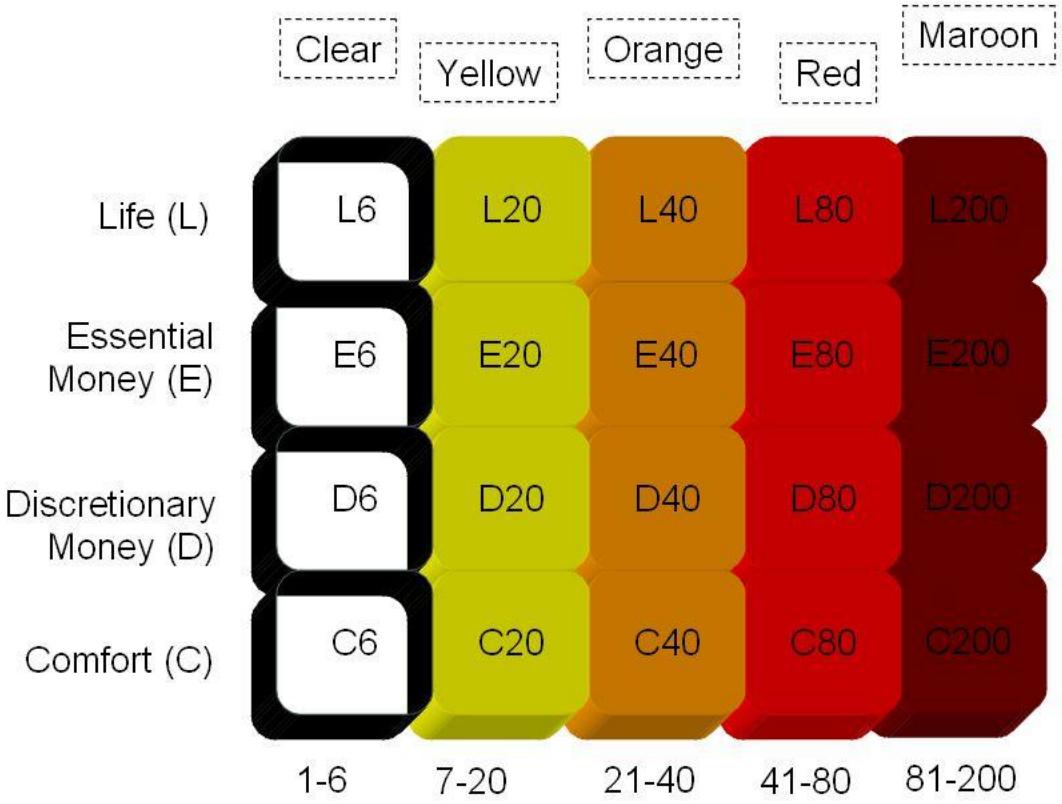


Figura 23 – Metodologías Crystal según criticidad y dimensión

Como se puede ver en la figura anterior, el tamaño del proyecto indica el método a utilizar, por ello estableció una clasificación por colores, por ejemplo Cristal Crear (3 a 8 personas), seguido por Yellow (10 a 20 personas), Crystal Orange (25 a 50 personas), y así sucesivamente hasta azul, mientras que la importancia indica la dureza con que se debe aplicar.

La otra gran clave de metodologías Crystal, común a casi todas las metodologías ágiles, es que lo más determinante para el éxito, o fracaso, de un proyecto son las personas.

Por último comentar las 7 propiedades fundamentales de las metodologías Crystal, que son:

- **Entregas frecuentes**, en base a un ciclo de vida iterativo e incremental. En función del proyecto puede haber desde entregas semanales hasta trimestrales.
- **Mejora reflexiva**. Que viene a ser mejora continua. Las iteraciones ayudan a ir ajustando el proyecto, a ir mejorándolo.
- **Comunicación osmótica**. Que el equipo esté en una misma ubicación física, para lograr la comunicación cara a cara.
- **Seguridad personal**. Todo el mundo puede expresar su opinión sin miedos, teniéndosele en cuenta, considerándose su opinión, etc.
- **Enfoque**. Períodos de no interrupción al equipo (2 horas), objetivos y prioridades claros, definiendo así tareas concretas.

- **Fácil acceso a usuarios expertos**. Las Crystal (a diferencia de otras como XP) no exigen que los usuarios estén continuamente junto al equipo de proyecto (no todas las organizaciones pueden hacerlo), sí que, como mínimo, semanalmente debe haber reuniones y los usuarios deben estar accesibles.
- **Entorno técnico con pruebas automatizadas**, gestión de la configuración e integración continua.

3.3.4 Dynamic Systems Development Method

A continuación hablaremos de la metodología ágil que más se aproxima a los métodos tradicionales, su implantación incluso permitiría alcanzar un nivel 2 de madurez según CMMI.

Dynamic Systems Development Method (DSDM) es la única de las metodologías aquí planteadas surgida de un Consorcio, formado originalmente por 17 miembros fundadores en Enero de 1994. El objetivo del Consorcio era producir una metodología de dominio público que fuera independiente de las herramientas y que pudiera ser utilizado en proyectos de tipo RAD (Rapid Application Development).

El Consorcio, tomando las best practices que se conocían en la industria y la experiencia traída por sus fundadores, liberó la primera versión de DSDM a principios de 1995. A partir de ese momento el método fue bien acogido por la industria, que empezó a utilizarlo y a capacitar a su personal en las prácticas y valores de DSDM.

La estructura del método fue guiada por estos nueve principios:

1. **El involucramiento del usuario es imperativo** → La participación del usuario es la clave principal para llevar a cabo un proyecto eficiente y efectivo, donde ambos, usuarios y desarrolladores compartan un sitio de trabajo, de tal forma que las decisiones se puedan hacer de la forma más exacta.
2. **Los equipos de DSDM deben tener el poder de tomar decisiones** → Se le deben otorgar poderes al equipo del proyecto para tomar decisiones que son importantes para el progreso del proyecto, sin tener que esperar a aprobaciones de más alto nivel.
3. **El foco está puesto en la entrega frecuente de productos** → Entregar algo suficientemente bueno pronto es siempre mejor que entregar todo perfecto al final. Entregando el producto con frecuencia desde una etapa temprana del proyecto, el producto se puede probar y revisar y el registro de pruebas y el documento de revisión se pueden tener en cuenta en la siguiente iteración o fase.
4. **La conformidad con los propósitos del negocio es el criterio esencial para la aceptación de los entregables** → El principal criterio de aceptación de un entregable es que entregue un sistema que trate las necesidades del negocio actuales. Entregar un sistema perfecto que trate todas las necesidades del negocio posibles es menos importante que centrarse en las funcionalidades críticas.
5. **El desarrollo iterativo e incremental** y conducido por la retroalimentación del usuario es necesario para converger hacia una correcta solución del negocio.
6. **Todos los cambios durante el desarrollo son reversibles**.
7. **Los requerimientos están especificados a un alto nivel** → Se debería hacer una línea base del alcance y los requisitos a alto nivel antes de que el proyecto empiece.

8. **El testing es integrado a través del ciclo de vida** → Las pruebas se llevan a cabo a lo largo de todo el ciclo de vida del proyecto.
9. **Un enfoque colaborativo y cooperativo entre todos los interesados es esencial**.

DSDM consta de tres fases:

- Fase pre-proyecto → En esta fase nos aseguraremos que todo está configurado y preparado correctamente preparado para que el proyecto comience y llegue a ser un éxito.
- Fase de ciclo de vida del proyecto → Está subdividida en 5 etapas: estudio de viabilidad, estudio de negocio, iteración de modelo funcional, iteración de diseño y construcción e implementación.
- Fase post-proyecto → La principal tarea de esta fase es el mantenimiento y corrección de errores.

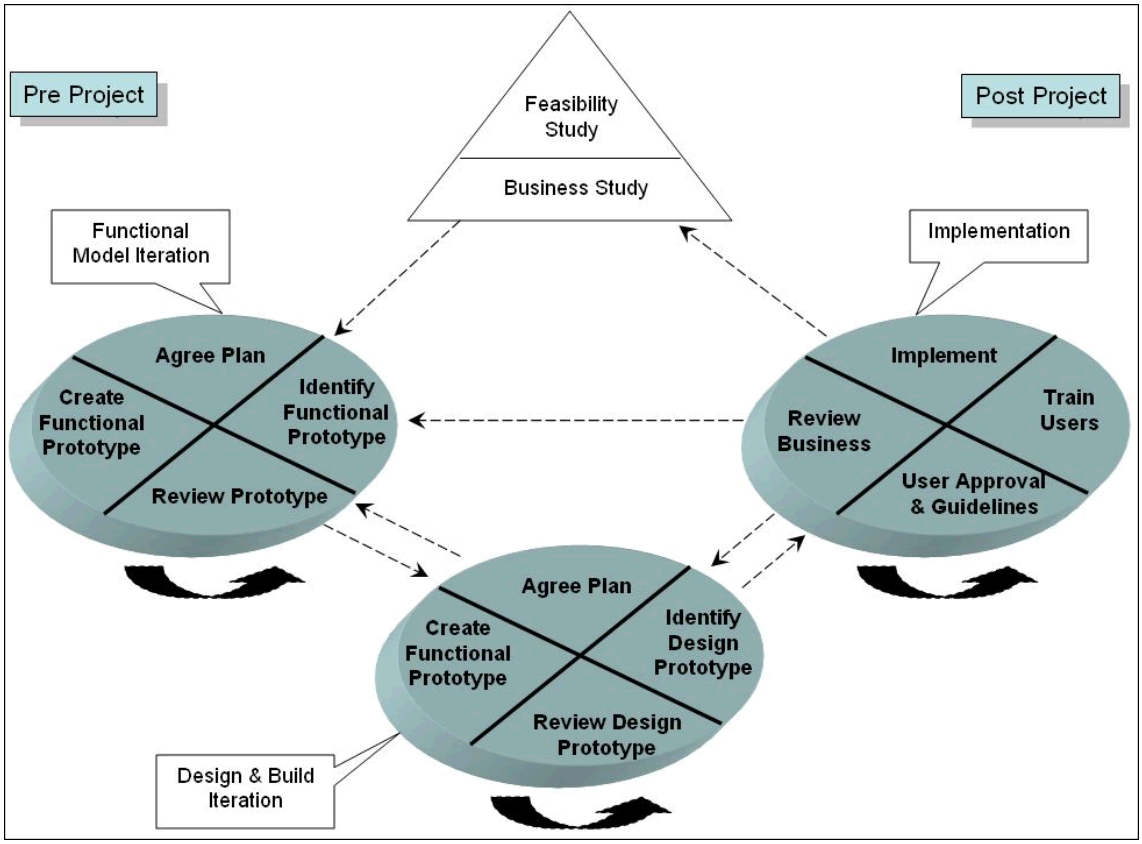


Figura 24 – Metodología DSDM

De las 5 etapas de la fase de ciclo de vida del proyecto, se debe comenzar por:

- **Estudio de viabilidad** → Pequeña fase que propone DSDM para determinar si la metodología se ajusta al proyecto en cuestión.
- **Estudio del negocio** → Se involucra al cliente de forma temprana, para tratar de entender la operatoria que el sistema deberá automatizar. Este estudio sienta las bases para iniciar el desarrollo, definiendo las características de alto nivel que deberá contener el software.

Una vez hechas estás dos etapas anteriores, las siguientes son fases iterativas:

- **Iteración del modelo funcional** → Produce una serie de prototipos increméntales que demuestran la funcionalidad para el cliente. Su propósito durante este ciclo es recopilar requisitos adicionales y producir documentación de análisis.
- **Iteración del diseño y construcción** → Revisa la construcción de prototipos durante la iteración del modelo funcional, en este se diseña el sistema para el uso operacional. En algunos casos, la iteración del modelo funcional y esta, suceden en forma concurrente.
- **Implantación** → Se implantará el sistema en producción previa aceptación del cliente.

El equipo mínimo de DSDM es de dos personas y puede llegar a seis, pero puede haber varios equipos en un proyecto. El mínimo de dos personas involucra que un equipo consiste de un programador y un usuario. El máximo de seis es el valor que se encuentra en la práctica. DSDM se ha aplicado a proyectos grandes y pequeños. La precondición para su uso en sistemas grandes es su partición en componentes que pueden ser desarrollados por equipos normales.

3.3.5 Extreme Programming

Extreme Programming (XP) o metodología de la programación extrema fue ideada por Kent Beck, a partir de un conjunto de buenas prácticas y un modo de realizar el desarrollo de software, fundamentado en entregar constantemente, el mayor valor al cliente.

XP tuvo su origen a finales de 1980. A partir de entonces, sus prácticas fueron perfeccionadas y sus ideas se centraron en el desarrollo de un software que fuera, a la vez, adaptativo y orientado a las personas. El furor de las metodologías ágiles, a finales de la década del 90, tuvo como una de sus mayores protagonistas a XP.

De la mano de Kent Beck, XP ha conformado un extenso grupo de seguidores en todo el mundo, disparando una gran cantidad de libros a los que dio comienzo el Libro Blanco de Kent Beckde 1999, que fue visto como la "Biblia" de XP (Reactualizado en su segunda edición de 2004). Inclusive Addison-Wesley ha creado una serie de libros denominada “The XP Series”.

En la primera edición, XP se definió con cuatro valores, quince principios básicos y doce prácticas. La mayoría del material acerca de XP en la web está basado en la primera edición del Libro blanco. Pero esta ya no es válido porque ahora las doce prácticas han desaparecido. En el nuevo XP, hay cinco valores, catorce principios, trece prácticas primarias y once prácticas como corolario. Las nuevas 24 prácticas apenas son un reflejo de las 12 originales. Dos de ellos, es decir, la metáfora y las normas de codificación, son abandonados.

3.3.5.1 Valores

A continuación se detallan los elementos básicos que Beck considera realmente importantes para desarrollar software exitosamente. Estos valores son la guía para el desarrollo en sí mismo y la inspiración de toda la metodología. Cuatro de ellos son los mismos que en XP original, y se agrega el respeto.

Comunicación:

La mayoría de los problemas y los errores son causados por la falta de comunicación. Por esta razón, la comunicación entre los miembros del equipo y entre el equipo y los clientes se debe desplegar al máximo.

Por otra parte, la comunicación interpersonal más efectiva es la comunicación directa, cara a cara. La otra comunicación es la que se establece entre los artefactos y las personas que los leen: deben ser fácil de leer y debe estar actualizada.

Por ejemplo para los programadores el código comunica mejor mientras más simple sea. 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 cómo 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.

Simplicidad:

Este es el más intelectual de los valores XP. Se dice: "Haz la cosa más simple que pueda funcionar". Sin embargo, la simplicidad - no simplista - es muy difícil. Se requiere experiencia, ideas y trabajo duro. La simplicidad favorece la comunicación, reduce la cantidad de código y mejora de la calidad.

Así, 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 disminuyen 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 mientras más grande se haga el proyecto, todo el equipo conocerá más y mejor el sistema completo.

Feedback o retroalimentación:

Siempre se debe ser capaz de medir el sistema y saber cómo de lejos está de las características necesarias.

Las herramientas de feedback fundamentales son el contacto cercano con el cliente y la disponibilidad de un conjunto de pruebas automatizadas o testeos, que se desarrollan al mismo tiempo que se desarrolla el sistema mismo. También contribuye a la simplicidad. La

simplicidad a menudo se obtiene mediante una práctica de tipo "prueba y error" o " try-and-error". Por otra parte, cuanto más simple es un sistema, más fácil es obtener retroalimentación o feedback sobre él.

Coraje:

Todas las metodologías y los procesos son herramientas para manejar y reducir nuestros miedos. Cuanto más miedo tenemos en un proyecto software, más grandes y más pesadas son las metodologías que necesitamos.

La comunicación, la sencillez y la retroalimentación permiten hacer frente con coraje, incluso grandes cambios en los requisitos aunque conlleven la reconstrucción profunda del sistema. Es un valor que, por sí solo, es peligroso, pero con otros valores, es una poderosa herramienta para hacer frente a los cambios.

Por ejemplo para los gerentes la programación en parejas puede ser difícil de aceptar, parece como si la productividad se fuese a reducir a la mitad ya que sólo la mitad de los programadores está escribiendo código. Hay que ser valiente para confiar en que la programación por parejas beneficia la calidad del código sin repercutir negativamente en la productividad. La simplicidad es uno de los principios más difíciles de adoptar. Se requiere coraje para implementar las características que el cliente quiere ahora sin caer en la tentación de optar por un enfoque más flexible que permite futuras modificaciones.

Respeto:

Los últimos cuatro valores implican un quinto: el respeto.

Si los miembros de un equipo no se preocupan por los demás y su trabajo, no existe una metodología que pueda funcionar. Se debe ser respetuoso con los compañeros y sus contribuciones, con la organización, y con las personas cuya vida se ve afectada por el sistema que se está codificando.

3.3.5.2 Principios

En la nueva versión de XP, los principios son el puente entre los valores, que son sintéticos y abstractos, y las prácticas, que indican la forma de desarrollar software.

Los catorce principios de XP son los siguientes:

Humanidad

El software es desarrollado por personas, por tal motivo, el factor humano es la clave principal para entregar software de calidad.

XP tiene como objetivo abordar los objetivos de la gente y sus organizaciones, por lo que puede beneficiar a ambos. Por supuesto, es necesario encontrar un equilibrio entre los objetivos. Si se sobreestiman las necesidades de la gente, ellos no van a trabajar correctamente, con la consecuente baja productividad y pérdidas empresariales. Hasta podría provocar el cierre de la empresa ocasionando gran perjuicio a los trabajadores.

Si se sobreestiman las necesidades de la organización, habrá ansiedad, exceso de trabajo y conflictos. Esto tampoco es bueno para los negocios.

Economía

Al producir software, también se debe producir valor para el negocio.

Dos aspectos de la economía son la clave para XP: valor actual y el valor de las opciones. La primera dice que un dólar hoy, vale más que un dólar mañana, por lo que el desarrollo de software más temprano, hace ganar dinero y hacerlo más tarde hace gastar dinero, cuanto más pronto sea, mayor será la ganancia que dará. Esto está relacionado con el valor de las opciones. Si usted puede aplazar las inversiones de diseño hasta que su necesidad sea obvia, es valioso para el negocio. Prácticas de XP, como el diseño incremental o el pago por uso (pay-per-use) facilita aplazar o diferir las decisiones.

Beneficio mutuo

Cada actividad debe beneficiar a todas las personas y organizaciones interesadas. Este es, quizás, el principio más importante de XP, y el más difícil de cumplir. Siempre hay soluciones fáciles para cualquier problema, donde algunos ganan y otros pierden. A menudo, estas soluciones son atajos tentadores. Sin embargo, son siempre una pérdida neta, debido a que derriba las relaciones y deteriora el medio ambiente de trabajo. Es necesario que se resuelvan más problemas de los que sean creados. Por lo tanto, se necesitan prácticas que sean beneficiosas tanto para la empresa como para el cliente, actualmente y en el futuro.

Auto-similitud

La naturaleza continuamente utiliza estructuras fractales, que son similares a sí mismas, pero en varias escalas. El mismo principio debe aplicarse al desarrollo de software: debemos ser capaces de volver a utilizar soluciones similares, en diferentes contextos. Por ejemplo, un patrón básico de XP es escribir pruebas que fallan, y luego escribir el código que pasa exitosamente el testeo. Esto es cierto en escalas de tiempo diferentes: en un trimestre, se listan los temas a resolver, y luego escribir las historias que los describen; en una semana, se listan historias para poner en práctica, se escriben los tests de aprobación, y, finalmente, escribir el código para que el testeo funcione, en pocas horas, ya se escribieron los tests (pruebas) unitarios, y luego el código capaz de hacer que funcionen.

Mejora

La mejora continua es la clave para XP. La perfección no existe, pero debe esforzarse continuamente por la perfección. Todos los días usted debe esforzarse para actuar de la mejor manera posible, también en pensar qué hacer para realizarlo aún mejor, mañana. En la práctica, en cada iteración del sistema se mejora en calidad y funcionalidades, utilizando la información (feedback) del cliente, a partir de pruebas (testeos) automatizadas y del propio equipo.

Diversidad

Es muy cómodo cuando en un equipo te encuentras que todos sus miembros son de perfiles y caracteres similares, pero estos equipos no son eficaces. Los equipos deben incluir diferentes saberes, habilidades y personalidades, para poder descubrir y resolver problemas.

Por supuesto, ser diferentes conduce a potenciales conflictos, que deben ser gestionados y resueltos. Tener diferentes opiniones y proponer soluciones diferentes es muy útil en el desarrollo de software, siempre y cuando, sean capaces de gestionar los conflictos y elegir la mejor alternativa.

Reflexión

Un equipo eficaz no se limita a hacer su trabajo. Se preguntan cómo están trabajando, y por qué están trabajando de esa manera. Tienen que analizar las razones del éxito - o el fracaso - sin ocultar los errores, hacerlas explícitas y tratando de aprender de ellas. Durante iteraciones trimestral y semanales, se debe tomar tiempo para reflexionar acerca de cómo se está ejecutando el proyecto, y cuáles son las posibles mejoras. Sin embargo, no se debe pensar demasiado.

La ingeniería de software tiene una larga tradición de gente tan ocupada pensando sobre la mejora de procesos, que ya no son capaces de codificar software. La reflexión debe venir después de la acción, y antes de la siguiente acción.

Flujo

Flujo significa desarrollar constantemente software útil, realizando todas las actividades de desarrollo juntas. Las prácticas de XP suponen un flujo continuo de actividades, no una secuencia de fases diferentes, con el lanzamiento del software sólo después de la última. Sólo el flujo continuo permite la retroalimentación (feedback) para asegurar que el sistema está evolucionando hacia la dirección correcta, y evita los problemas relacionados con la integración final.

Oportunidad

Los problemas deben ser vistos como una oportunidad de mejora. Siempre se presentan problemas, pero para alcanzar la excelencia, no se puede solamente corregir los problemas. Es necesario convertirlos en oportunidades de aprendizaje y mejora. Por ejemplo, si no se pueden hacer planes a largo plazo, una solución sería el uso de ciclos de planificación trimestrales, y revisar los planes a largo plazo, cada tres meses. Otro ejemplo, si el trabajo individual producido por un desarrollador presenta muchos errores. En ese caso una solución posible es programación por parejas. Las prácticas de XP son eficaces precisamente porque le dan rumbo a problemas viejos, siempre presentes durante el desarrollo de software.

Redundancia

Problemas críticos y difíciles deben resolverse de varias maneras diferentes. Así, si una solución falla, las otras van a prevenir un desastre. En estos casos, el costo de la redundancia es fácilmente pagado. Los defectos de software deben ser buscados, encontrados y corregidos de muchas maneras, (programación en parejas, pruebas automatizadas, sentarse juntos, la

participación real de los clientes, etc.) Esto es redundante, ya que, muchas veces se encuentran muchos defectos. Sin embargo, la calidad no tiene precio. Por supuesto, la adición de una práctica que sistemáticamente encuentra defectos que ya se encuentran por otras prácticas es una redundancia inútil, que debe ser evitada.

Fallo

No ser capaces de lograr un éxito, es un fallo. Si no se sabe cómo poner en práctica una historia, se debe tratar de implementar de tres o cuatro maneras diferentes. Incluso si todas fallan, se habrá aprendido mucho.

¿Es útil el fracaso? Sí, si enseña algo. Por lo tanto, no se debe temer al fracaso. Es mejor probar algo y fallar, en lugar de demorar demasiado tiempo para una acción, tratando de hacer lo correcto desde el principio.

Calidad

La calidad debe ser siempre la máxima posible. Aceptar una menor calidad no produce un ahorro, ni un desarrollo más rápido. Por el contrario, la mejora de la calidad produce necesariamente una mejora de otras características del sistema, como la productividad y la eficiencia. Por otra parte, la calidad no es sólo un factor económico. Los miembros del equipo deben estar orgullosos de su trabajo porque mejora la auto-estima del equipo y la eficacia. No se debe confundir la calidad con el perfeccionismo, sin embargo.

Si se demora la acción en búsqueda de la máxima calidad, eso no es promover realmente la calidad. Es mucho mejor intentar y fallar, y luego perfeccionar aquellas soluciones imperfectas encontradas.

Pasos de Bebé

Grandes cambios, preparados en un largo período de tiempo y hechos de una vez, son peligrosos. Para proceder de manera iterativa, es mucho mejor hacerlo en pasos de bebé - el menor paso que se puede apreciar en la dirección correcta. Por otra parte, pasos de bebé no significa avanzar lentamente. Un equipo capaz de avanzar con pasos de bebé puede dar un montón de ellos en poco tiempo, volviéndose rápido y con alta capacidad de respuesta. Una de las razones detrás de los pasos del bebé es que un pequeño paso en la dirección equivocada ocasiona pequeños daños, mientras que un gran paso puede dañar severamente el proyecto.

Aceptación de la responsabilidad

La responsabilidad sólo puede ser aceptada. Es fácil ordenar a los desarrolladores "Haz esto", o "Haz aquello", pero no funciona. Inevitablemente, se le pedirá menos de lo que podría lograrse o, probablemente, más de lo que puede ser logrado. De todos modos, la persona que recibe la orden decidirá si tomará la responsabilidad y aceptará la orden, o no aceptar la responsabilidad y empezar a pasar la pelota.

3.3.5.3 Prácticas

El nuevo XP se basa en trece prácticas primarias, y once prácticas finales o corolario.

Las prácticas primarias deben ser aplicadas primero, y cada una de ellas puede producir una mejora en el proceso de desarrollo de software. Las prácticas finales o corolario, por el contrario, requieren tener habilidad con las prácticas primarias, y son difíciles de aplicar sin haber aplicado las primarias.

Las 24 prácticas son muy importantes, y se deben aplicar en su totalidad a fin de obtener todos los beneficios posibles gracias a XP. A continuación se exponen algunas de esas prácticas:

- **El juego de la planificación** → Hay una comunicación frecuente entre el cliente y los programadores. El equipo técnico realiza una estimación del esfuerzo requerido para la implementación de las historias de usuario y los clientes deciden sobre el ámbito y tiempo de las entregas y de cada iteración.
- **Entregas pequeñas** → Producir rápidamente versiones del sistema que sean operativas, aunque no cuenten con toda la funcionalidad del sistema. Esta versión ya constituye un resultado de valor para el negocio. Una entrega no debería tardar más 3 meses.
- **Metáfora** → El sistema es definido mediante una metáfora o un conjunto de metáforas compartidas por el cliente y el equipo de desarrollo. Una metáfora es una historia compartida que describe cómo debería funcionar el sistema (conjunto de nombres que actúen como vocabulario para hablar sobre el dominio del problema, ayudando a la nomenclatura de clases y métodos del sistema).
- **Diseño simple** → Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto.
- **Pruebas** → La producción de código está dirigida por las pruebas unitarias. Éstas son establecidas por el cliente antes de escribirse el código y son ejecutadas constantemente ante cada modificación del sistema.
- **Refactorización (Refactoring)** → Es una actividad constante de reestructuración del código con el objetivo de remover duplicación de código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios. Se mejora la estructura interna del código sin alterar su comportamiento externo.
- **Programación en parejas** → Toda la producción de código debe realizarse con trabajo en parejas de programadores. Esto conlleva ventajas implícitas (menor tasa de errores, mejor diseño, mayor satisfacción de los programadores, etc.).
- **Propiedad colectiva del código** → Cualquier programador puede cambiar cualquier parte del código en cualquier momento
- **Integración continua** → Cada pieza de código es integrada en el sistema una vez que esté lista. Así, el sistema puede llegar a ser integrado y construido varias veces en un mismo día.
- **40 horas por semana** → Se debe trabajar un máximo de 40 horas por semana. No se trabajan horas extras en dos semanas seguidas. Si esto ocurre, probablemente está ocurriendo un problema que debe corregirse. El trabajo extra desmotiva al equipo.
- **Cliente in-situ** → El cliente tiene que estar presente y disponible todo el tiempo para el equipo. Éste es uno de los principales factores de éxito del proyecto XP. El cliente

conduce constantemente el trabajo hacia lo que aportará mayor valor de negocio y los programadores pueden resolver de manera inmediata cualquier duda asociada. La comunicación oral es más efectiva que la escrita.

- **Estándares de programación** → XP enfatiza que la comunicación de los programadores es a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación para mantener el código legible.

3.3.5.4 Elementos de la Metodología

Las historias de usuario

Son la técnica utilizada para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. El tratamiento de las historias de usuario es muy dinámico y flexible. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarlas en unas semanas. Las historias de usuario se descomponen en tareas de programación y se asignan a los programadores para ser implementadas durante una iteración

Roles XP

Los roles de acuerdo con la propuesta de Beck son:

- **Programador**: El programador escribe las pruebas unitarias y produce el código del sistema.
- **Cliente**: Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en apoyar mayor valor al negocio.
- **Encargado de pruebas (tester)**: Ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para las pruebas.
- **Encargado de seguimiento (tracker)**: Proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, para mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.
- **Entrenador (coach)**: Es el responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.
- **Consultor**: Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto, en el que puedan surgir problemas.
- **Gestor (big boss)**: Es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

Proceso XP

El ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos:

1. El cliente define el valor de negocio a implementar.
2. El programador estima el esfuerzo necesario para su implementación.

3. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
4. El programador construye ese valor de negocio.
5. Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. 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. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse de que el sistema tenga el mayor valor de negocio posible.

3.3.5.5 Fases

El ciclo de vida ideal de XP consisten en 6 fases: exploración, planificación de la entrega, iteraciones, producción, mantenimiento y muerte del proyecto.

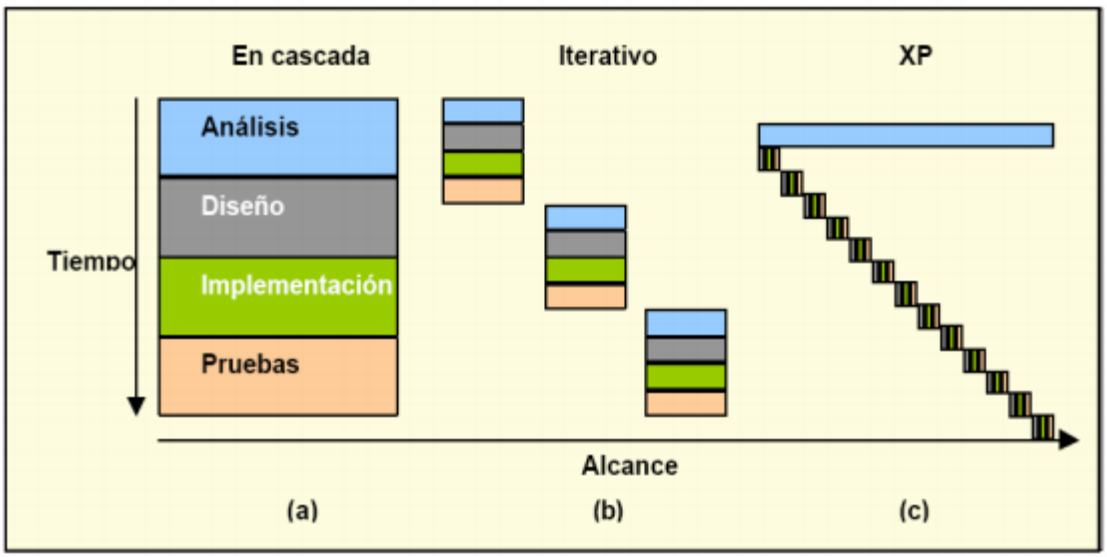


Figura 25 – Iteraciones cortas de XP frente a iterativo o cascada

Fase de exploración

Es la fase en la que se define el alcance general del proyecto. En esta fase, el cliente define lo que necesita mediante la redacción de sencillas "historias de usuario". Los programadores estimas los tiempos de desarrollo en base a esta información. Debe quedar claro que las estimaciones realizadas en esta fase son primarias (ya que están basadas en datos de muy alto nivel), y podrían variar cuando se analicen en más detalle en cada iteración.

Esta fase dura típicamente un par de semanas, y el resultado es una visión general del sistema, y un plazo total estimado.

Fase de planificación

La planificación es una fase corta, en la que el cliente, los gerentes y el grupo de desarrolladores acuerdan el orden en que deberán implementarse las historias de usuario, y,

asociadas a éstas, las entregas. Típicamente esta fase consiste en una o varias reuniones grupales de planificación. El resultado de esta fase es un Plan de Entregas

Fase de iteraciones

Esta es la fase principal en el ciclo de desarrollo de XP. Las funcionalidades son desarrolladas en esta fase, generando al final de cada una un entregable funcional que implementa las historias de usuario asignadas a la iteración. Como las historias de usuario no tienen suficiente detalle como para permitir su análisis y desarrollo, al principio de cada iteración se realizan las tareas necesarias de análisis, recabando con el cliente todos los datos que sean necesarios. El cliente, por lo tanto, también debe participar activamente durante esta fase del ciclo. Las iteraciones son también utilizadas para medir el progreso del proyecto. Una iteración terminada sin errores es una medida clara de avance.

Fase de puesta en producción

Si bien al final de cada iteración se entregan módulos funcionales y sin errores, puede ser deseable por parte del cliente no poner el sistema en producción hasta tanto no se tenga la funcionalidad completa.

En esta fase no se realizan más desarrollos funcionales, pero pueden ser necesarias tareas de ajuste.

3.3.6 Feature Driven Development

Feature Driven Development (FDD) ideada por Jeff De Luca y Peter Coad a mediados de los 90, combina el desarrollo dirigido por modelos con el desarrollo ágil. Se centra en el diseño de un modelo inicial, cuyo desarrollo será dividido en función a las características que debe cumplir el software e, iterativamente, se diseñará cada una de estas características.

Por tanto, cada iteración consta de dos partes, diseño e implementación de cada característica. A diferencia de otras metodologías ágiles no cubre todo el ciclo de vida sino sólo las fases de diseño y construcción. No requiere un modelo específico de proceso y se complementa con otras metodologías. Además, hace énfasis en aspectos de calidad durante todo el proceso e incluye un monitoreo permanente del avance del proyecto, por eso este tipo de metodología está dirigido al desarrollo de aplicaciones con un alto grado de criticidad.

3.3.6.1 Características

- Se preocupa por la calidad, por lo que incluye un monitoreo constante del proyecto.
- Ayuda a contrarrestar situaciones como el exceso en el presupuesto, fallos en el programa o el hecho de entregar menos de lo deseado.
- Propone tener etapas de cierre cada dos semanas. Se obtienen resultados periódicos y tangibles.
- Se basa en un proceso iterativo con iteraciones cortas que producen un software funcional que el cliente y la dirección de la empresa pueden ver y monitorear.

- Define claramente entregas tangibles y formas de evaluación del progreso del proyecto.
- No hace énfasis en la obtención de los requisitos sino en cómo se realizan las fases de diseño y construcción.

3.3.6.2 Procesos

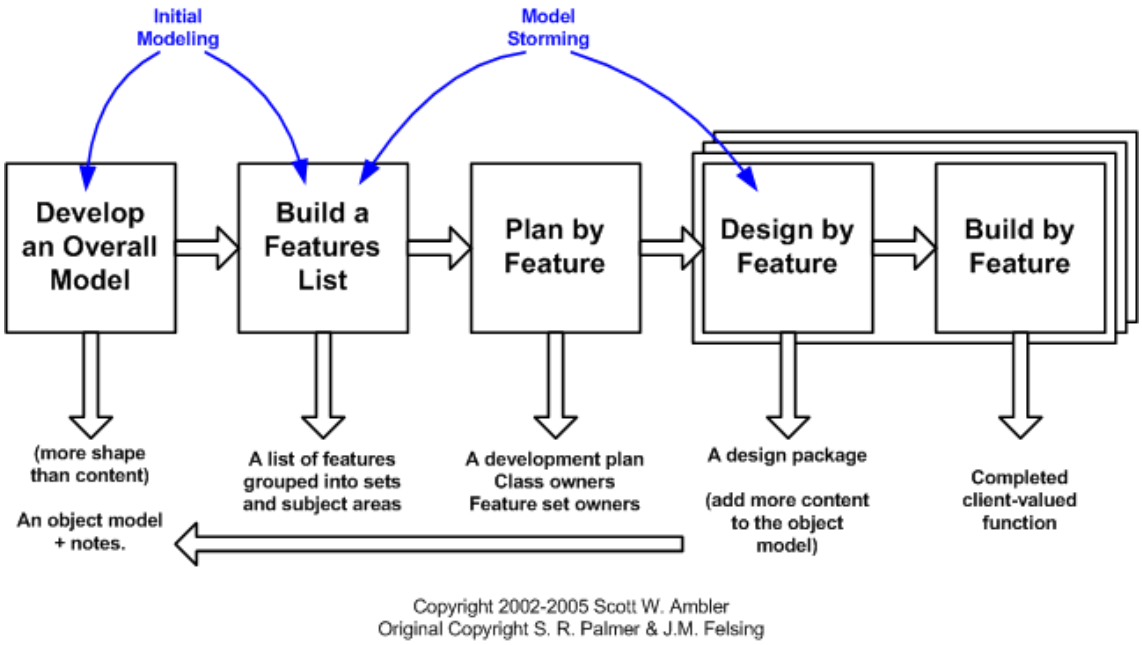


Figura 26 – Procesos Metodología FDD

A continuación se detallan los procesos que forman parte de FDD:

Desarrollar un modelo global

Al inicio del desarrollo se construye un modelo teniendo en cuenta la visión, el contexto y los requisitos que debe tener el sistema a construir. Este modelo se divide en áreas que se analizan detalladamente. Se construye un diagrama de clases por cada área.

Construir una lista

Se elabora una lista que resuma las funcionalidades que debe tener el sistema, cuya lista es evaluada por el cliente. Cada funcionalidad de la lista se divide en funcionalidades más pequeñas para un mejor entendimiento del sistema.

Planear

Se procede a ordenar los conjuntos de funcionalidades conforme a su prioridad y dependencia, y se asigna a los programadores jefes.

Diseñar

Se selecciona un conjunto de funcionalidades de la lista. Se procede a diseñar y construir la funcionalidad mediante un proceso iterativo, decidiendo que funcionalidad se van a realizar en

cada iteración. Este proceso iterativo incluye inspección de diseño, codificación, pruebas unitarias, integración e inspección de código.

Construir

Se procede a la construcción total del proyecto.

3.3.6.3 Roles y Responsabilidades

El equipo de trabajo está estructurado en jerarquías, siempre debe haber un jefe de proyecto, y aunque es un proceso considerado ligero también incluye documentación (la mínima necesaria para que algún nuevo integrante pueda entender el desarrollo de inmediato).

El FDD clasifica sus roles en las siguientes tres categorías:

Roles Clave

- Administrador del proyecto → Quien tiene la última palabra en materia de visión, cronograma y asignación del personal.
- Arquitecto jefe → Realiza el diseño global del sistema. Ejecución de todas las etapas.
- Director de desarrollo → Lleva diariamente las actividades de desarrollo. Resuelve conflictos en el equipo. Resuelve problemas referentes a recursos.
- Programador Jefe → Analiza los requerimientos. Diseña el proyecto. Selecciona las funcionalidades a desarrollar de la última fase del FDD.
- Propietario de clases → Responsable del desarrollo de las clases que se le asignaron como propias. Participa en la decisión de que clase será incluida en la lista de funcionalidades de la próxima iteración.
- Expertos de dominio → Puede ser un usuario, un cliente, analista o una mezcla de estos. Poseen el conocimiento de los requerimientos del sistema. Pasa el conocimiento a los desarrolladores para que se asegure la entrega de un sistema completo.

Figura 27 – Roles clave FDD

Roles de Soporte

- Administrador de entrega → Controla el progreso del proceso revisando los reportes del programador jefe y manteniendo reuniones breves con él. Reporta al manager del proyecto.
- Abogado/guru de lenguaje → Conoce a la perfección el lenguaje y la tecnología.
- Ingeniero de construcción → Se encarga del control de versiones de los builds y publica la documentación.
- Herramentista → Construye herramientas ad hoc o mantiene bases de datos y sitios Web.
- Administrador del sistema → Controla el ambiente de trabajo o optimiza el sistema cuando se lo entrega.

Roles Adicionales

- Verificadores.
- Encargados del despliegue.
- Escritores técnicos.

Un miembro de un equipo puede tener otros roles a cargo, y un solo rol puede ser compartido por varias personas.

3.3.7 Lean Software Development

Basándose en los fundamentos de la filosofía Lean para la fabricación, ideada por Taiichi Ohno alrededor de 1956 y que es la esencia del sistema de producción de Toyota (TPS = Toyota Production System) surge una corriente de pensamiento que aplica los principios Lean al desarrollo de Software. Tom y Mary Poppendieck escriben en 2003 el libro que traslada los principios de Lean Production System al mundo del desarrollo de Software.

El lean, y el lean software development, no es realmente una metodología de ingeniería de software en el sentido convencional. Es más una síntesis de principios y una filosofía para construir sistemas de software. Si lean se considera un conjunto de principios más que prácticas, la aplicación de conceptos lean al desarrollo software y la ingeniería software puede ayudar a mejorar la calidad.

3.3.7.1 Características

- Lean es una filosofía y una forma de pensar.
- La filosofía Lean, analiza los procesos de producción y ELIMINA todo lo que no produzca valor para el cliente.
- Lean es un conjunto de conceptos y técnicas pensadas para el aumento de la productividad y la producción con calidad.
- Se trata de eliminar los “desperdicios”, Muda, que nos hacen ser menos productivos en nuestro trabajo.
- Es una filosofía de pensamiento en la que se basaron y se basan los métodos ágiles para promulgar sus principios.

3.3.7.2 Conceptos Clave

Los Jefes de Proyectos han de preocuparse por detectar y trabajar en tres conceptos de la filosofía Lean:

- **MUDA** → Estudiar el flujo con el que trabaja tu equipo para eliminar la Muda (mediante la mejora del flujo se eliminan los desperdicios.)
- **MURA** → Analizar y revisar el trabajo que tiene cada persona del equipo para prevenir la Mura (Mura o sobrecarga de trabajo que crea cuellos de botella que afectan al flujo).
- **MURI** → Conocer los tipos de trabajo que realizan en el equipo para controlar el Muri (Muri o variabilidad del flujo de trabajo que se controla definiendo políticas específicas en función del tipo de actividad que se realiza en el equipo).

3.3.7.3 Principios

Los principios utilizados en la metodología Lean Software Development son:

1. **Eliminar el desperdicio** → Todas las actividades que no crean valor no sirven y deben ser eliminadas. Por ejemplo: cuando se desarrollan tareas que no fueron solicitadas por el cliente, cuando existe una sobre documentación del proyecto, cuando el proceso de desarrollo se cumple sin analizar su nivel de eficiencia o vigencia. Respecto del código se debe tener en cuenta que mayor cantidad de código no siempre es mejor, ya que en general requiere un mayor esfuerzo de testeo y de mantenimiento. Por otro lado todos los errores, bugs y fallos del software son verdadero desperdicio que debe ser minimizado y eliminado.
2. **Construir con calidad** → La calidad debe ser vista tanto desde el proceso como desde el producto. Un proceso que respeta la calidad es aquel que es conocido, entendido y mejorado por sus propios participantes. Para ello se necesita un importante nivel de compromiso y respeto. Para desarrollar productos con calidad se pueden tener en cuenta elementos como:
    a. Técnicas como TDD (Test Driven Development) permiten que usuarios (clientes), programadores y tester definan claramente los requerimientos y confeccionen pruebas de aceptación antes de escribir el código. Ayuda a la comprensión de los programadores y mejora el entendimiento de los requerimientos.
    b. El programador es responsable de su propio desarrollo. No debe esperar a que Testing o QA descubra los errores.
    c. Fomentar el desarrollo de pruebas automatizadas.
    d. Refactorizar el código, buscando que no existan duplicaciones.
3. **Crear conocimiento** → Llegar al conocimiento de lo que necesita el cliente requiere de mucha dedicación y esfuerzo y debe convertirse en el aspecto principal a tener en cuenta, ya que el desarrollo de un producto que no es útil termina siendo un desperdicio. El proceso de desarrollo de software es un proceso de aprendizaje: hay que entender qué es lo que el cliente quiere y cómo podemos hacerlo lo mejor posible. El desarrollo iterativo incremental nos permite repetir muchas veces el proceso de aprendizaje para que podamos crear el conocimiento necesario.

4. **Diferir el compromiso** → Lean sostiene que el compromiso (básicamente con los requisitos del cliente) no puede hacerse hasta que los mismos no estén claramente expresados y entendidos. En muchos proyectos se hace un compromiso inicial con requisitos incompletos, inestables e incoherentes, siendo esto un detonante del fracaso del proyecto. No todos los requisitos tienen la misma importancia para el cliente, por ello se recomienda tomar compromiso respecto del análisis de aquellos que se merecen esta inversión de tiempo y esfuerzo. Lo mismo sucede con qué requisitos pueden ser diseñados, codificados y testeados.
5. **Entregar rápido** → El desarrollo iterativo permite realizar entregas rápidas a los clientes, quienes se encuentran con código funcionando desde etapas tempranas. Dicho código debe ser desarrollado con calidad ya que no se puede mantener una velocidad importante de entrega si no se cuenta con calidad y un equipo disciplinado, comprometido y confiable. La entrega rápida permite a la organización ser competitiva respecto a otras, posicionarse en el mercado, y obtener ingresos de manera más temprana.
6. **Respetar a las personas** → Lean se basa en el respecto por las personas. Las mismas son el elemento único y diferenciador por excelencia de la organización. Se busca capacitarlas y hacerlas responsable de los procesos en los que interviene, de modo que si son necesarios cambios y mejoras, cada persona puede colaborar en el desarrollo de las mismas. Las técnicas como el análisis de problemas, y la responsabilidad que se proyecta en todos los actores es esencial para asegurar la participación y respeto de todos los involucrados.
7. **Optimizar el todo** → Lean nos invita a focalizarnos en el proceso completo, es decir todo el flujo de valor, en lugar de hacerlo en cada etapa. El problema con optimizar cada paso es que genera inventarios grandes entre los pasos. En el mundo del software, estos "inventarios" representan al trabajo parcialmente terminado (por ejemplo, requerimientos completos, pero sin diseñar, codificar o probar). Lean demostró que un flujo de "una pieza" (por ejemplo, enfocarse en construir un ítem de manera completa) es un proceso mucho más eficiente que concentrarse en construir todas las partes más rápido.

3.3.8 Kanban

La metodología de desarrollo Kanban (Kanban Software Development) está basada en la metodología de fabricación industrial del mismo nombre. Su objetivo es gestionar de manera general como se van completando tareas, pero en los últimos años se ha utilizado en la gestión de proyectos de desarrollo software.

3.3.8.1 Principios

Visualizar el trabajo y las fases del ciclo de producción o flujo de trabajo

Kanban se base en el desarrollo incremental, dividiendo el trabajo en partes. Una de las principales aportaciones es que utiliza técnicas visuales para ver la situación de cada tarea, y que se representa en pizarras llenas de post-it.

El trabajo se divide en partes, normalmente cada una de esas partes se escribe en un post-it y se pega en una pizarra. Los post-it suelen tener información variada, si bien, aparte de la descripción, debieran tener la estimación de la duración de la tarea.

La pizarra tiene tantas columnas como estados por los que puede pasar la tarea (ejemplo, en espera de ser desarrollada, en análisis, en diseño, etc.).

El objetivo de esta visualización es que quede claro el trabajo a realizar, en qué está trabajando cada persona, que todo el mundo tenga algo que hacer y el tener clara la prioridad de las tareas.

Las fases del ciclo de producción o flujo de trabajo se deben decidir según el caso, no hay nada acotado. En la figura se han puesto un conjunto de fases de ejemplo.

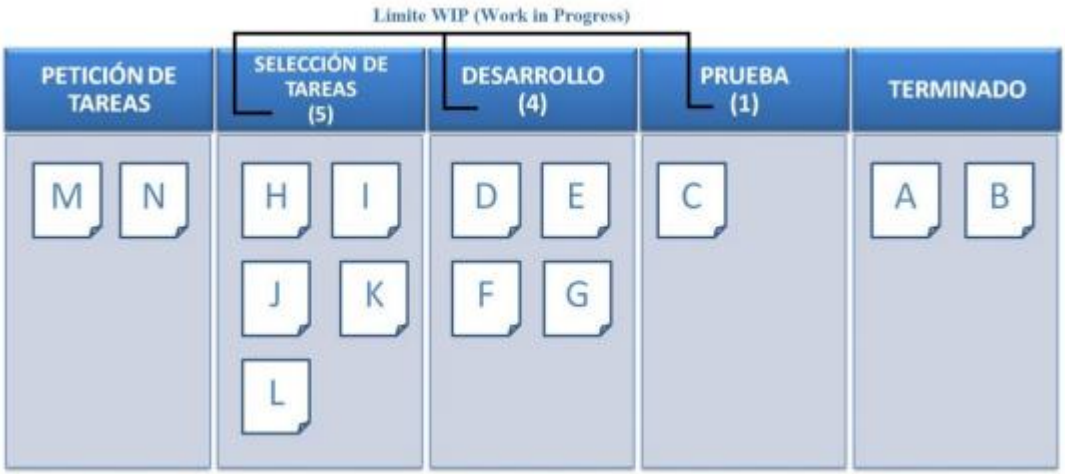


Figura 28 – Muro Kanban

Determinar el límite del trabajo en curso (Work In Progress)

Quizás una de las principales ideas del Kanban es que el trabajo en curso (Work In Progress) debería estar limitado, es decir, que el número de tareas que se pueden realizar en cada fase debe ser algo conocido. Independientemente de si un proyecto es grande o pequeño, simple o complejo, hay una cantidad de trabajo óptima que se puede realizar sin sacrificar eficiencia, por ejemplo, puede ser que realizar diez tareas a la vez nos lleve una semana, pero hacer dos cosas a la vez nos lleve sólo unas horas, lo que nos permite hacer quince tareas en la semana.

En Kanban se debe definir cuantas tareas como máximo puede realizarse en cada fase del ciclo de trabajo (ejemplo, como máximo 4 tareas en desarrollo, como máximo 1 en pruebas, etc.), a ese número de tareas se le llama límite del "work in progress". A esto se añade otra idea tan razonable como que para empezar con una nueva tarea alguna otra tarea previa debe haber finalizado

Medir el tiempo en completar una tarea (Lead time)

El tiempo que se tarda en terminar cada tarea se debe medir, a ese tiempo se le llama "lead time". El "lead time" cuenta desde que se hace una petición hasta que se hace la entrega

Aunque la métrica más conocida del Kanban es el "lead time", normalmente se suele utilizar también otra métrica importante: el "cycle time". El "cycle time" mide desde que el trabajo sobre una tarea comienza hasta que termina. Si con el "lead time" se mide lo que ven los clientes, lo que esperan, y con el "cycle time" se mide más el rendimiento del proceso.

3.3.8.2 Roles

La metodología Kanban no prescribe roles. Tener un papel asignado y las tareas asociadas a dicho papel crean una identidad en el individuo. Por lo tanto, pedir que adopten un nuevo papel o un nuevo puesto de trabajo puede ser entendido como un ataque a su identidad. Habría una resistencia al cambio. Kanban trata de evitar esa resistencia emocional, entiende que la ausencia de papeles es una ventaja para el equipo.

3.3.8.3 Fases

En Kanban no existen unas fases definidas, sino que se habla de un flujo.

No existen iteraciones de tiempo definido, sino que trabaja con cadencias, es decir, se liberan versiones o entregables en base a necesidades o eventos. Tampoco existen unas reuniones diarios, o de inicio de sprint (porque no existen).

Lo extremista que es está metodología en cuanto a lo adaptativa que es y lo poco que define procesos, roles e incluso la forma de hacer el tablero Kanban, hace que en algunos círculos, sobre todo los más cercanos a Scrum, consideren a Kanban una técnica de señalización más que una metodología.

3.3.9 Scrum

La palabra Scrum es un término utilizado en rugby para restablecer la unión después de una interrupción.

Scrum es un proceso ágil que se puede usar para gestionar y controlar desarrollos complejos de software y productos usando prácticas iterativas e incrementales, así define un proceso empírico, iterativo e incremental de desarrollo que intenta obtener ventajas respecto a los procesos definidos (cascada, espiral, prototipos, etc.) mediante la aceptación de la naturaleza caótica del desarrollo de software, y la utilización de prácticas tendientes a manejar la impredictibilidad y el riesgo a niveles aceptables.

Aunque Scrum estaba previsto que fuera para la gestión de proyectos de desarrollo de software, se puede usar también para la ejecución de equipos de mantenimiento de software o como un enfoque de gestión de programas (Scrum de Scrums)

3.3.9.1 Historia

Surge en 1986, nacido de un artículo de la Harvard Business Review titulado "The New New Product Development Game" de Hirotaka Takeuchi e Ikujiro Nonaka, que introducía las mejores prácticas más utilizadas en 10 compañías japonesas. Hirotaka Takeuchi e Ikujiro

Nonaka describieron un enfoque integral que incrementaba la velocidad y flexibilidad del desarrollo de nuevos productos comerciales. Compararon este nuevo enfoque integral, en el que las fases se solapan fuertemente y el proceso entero es llevado a cabo por un equipo multifuncional a través de las diferentes fases, con el rugby, donde todo el equipo trata de ganar distancia como una unidad y pasando el balón una y otra vez.

En 1991, DeGrace y Stahl hicieron referencia a este enfoque como Scrum, un término de rugby mencionado en el artículo de Takeuchi y Nonaka. A principios de 1990s, Ken Schwaber usó un enfoque que guio a Scrum a su compañía, Métodos de Desarrollo Avanzados. Al mismo tiempo, Jeff Sutherland desarrolló un enfoque similar en Easel Corporation y fue la primera vez que se llamó Scrum. En 1995 Sutherland y Schwaber presentaron de forma conjunta un artículo describiendo Scrum en OOPSLA '95 en Austin, su primera aparición pública. Schwaber y Sutherland colaboraron durante los siguientes años para unir los artículos, sus experiencias y las mejores prácticas de la industria en lo que ahora se conoce como Scrum. En 2001, Schwaber se asoció con Mike Beedle para poner en limpio el método en el libro Agile Software Devlopment with Scrum.

3.3.9.2 Operativa

Scrum es un esqueleto de proceso que incluye un conjunto de prácticas y roles predefinidos. Los roles principales en Scrum son el "ScrumMaster" que mantiene los procesos y trabaja junto con el jefe de proyecto, el "Product Owner" que representa a las personas implicadas en el negocio y el "Team" que incluye a los desarrolladores.

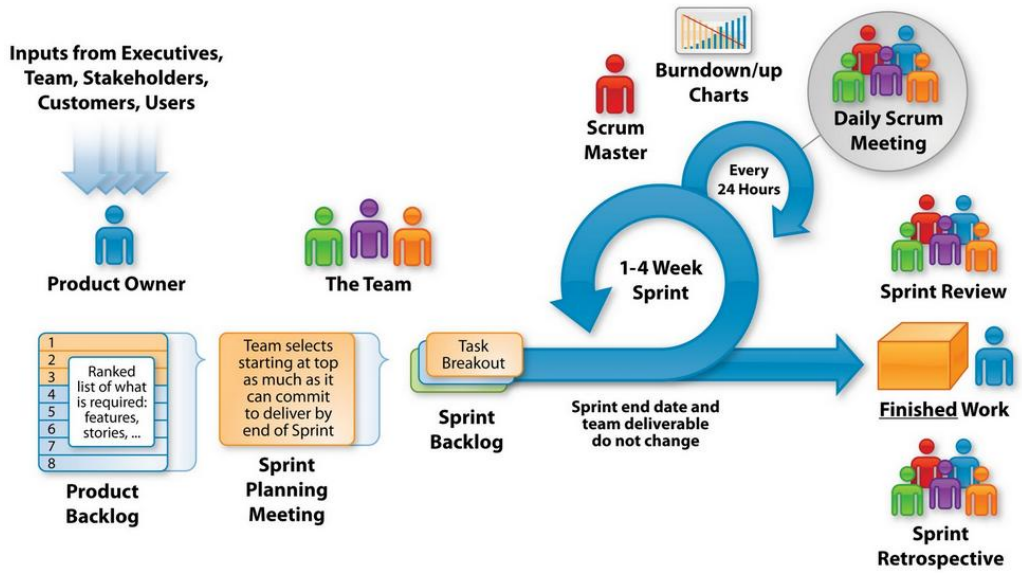


Figura 29 – Flujo Metodología Scrum

Durante cada iteración (sprint- periodos de tiempo), típicamente un periodo de 2 a 4 semanas (longitud decidida por el equipo), el equipo crea un incremento de software operativo. El conjunto de características que entra en una iteración viene del "backlog" del producto, que es un conjunto priorizado de requisitos de trabajo de alto nivel que se han de hacer. Los ítems que entran en una iteración se determinan durante la reunión de planificación de la iteración. Durante esta reunión, el Product Owner informa al equipo de los ítems en el backlog del

producto que quiere que se completen. El equipo determina entonces a cuanto de eso puede comprometerse a completar durante la siguiente iteración. Durante una iteración, nadie puede cambiar el backlog de la iteración, lo que significa que los requisitos están congelados para esa iteración. Cuando se completa una iteración, el equipo demuestra el uso del software.

Scrum permite la creación de equipos con propia organización fomentando la localización conjunta de todos los miembros del equipo y la comunicación verbal entre todos los miembros del equipo y las disciplinas implicadas en el proyecto.

Un principio clave de Scrum es el reconocimiento de que durante un proyecto los clientes pueden cambiar sus pensamientos sobre lo que quieren y necesitan, y de que los desafíos que no se pueden predecir no se pueden tratar fácilmente de una forma predictiva o planificada tradicional. Por esto, Scrum adopta un enfoque empírico, aceptando que el problema no se puede entender o definir completamente, centrándose en cambio en maximizar las habilidades del equipo para entregar rápidamente y responder a los requisitos emergentes.

Una parte muy importante de Scrum son las reuniones que se realizan durante cada una de las iteraciones. Hay distintos tipos:

- Scrum diario: Cada día durante la iteración, tiene lugar una reunión de estado del proyecto. A esta reunión se le domina Scrum
- Reunión de planificación de iteración (sprint): Se lleva a cabo al principio del ciclo de la iteración.
- Reunión de revisión de iteración: Al final del ciclo de la iteración.
- Iteración retrospectiva: Al final del ciclo de la iteración.

Así cada sprint lo podríamos resumir en los siguientes pasos:

- Se desarrolla una reunión al comienzo del sprint (Sprint Planning Meeting), en la cual se desarrolla el Sprint BackLog. Esta reunión suele estar ajustada en el tiempo a 4 horas.
- Cada día se realizan reuniones diarias de 15 minutos (time-boxing) de duración (siempre en el mismo lugar, a la misma hora y de pie) donde se pone de manifiesto qué se ha hecho y qué se va a hacer, así como de los problemas que pueden surgir para alcanzarlo. Se actualiza el Sprint BackLog y el Burndown.
- Otras informativas en la que se revisa el Sprint finalizado (sprint review meeting). Además, se realiza una retrospectiva de los problemas y forma de trabajar (retrospective meeting) planteándose las siguientes cuestiones: qué fue bien, qué cosas no y qué mejoras deberían hacerse para los siguientes sprints.
- Presentación del siguiente Sprint

3.3.9.3 Conceptos Clave

A continuación se enumeran los puntos clave de Scrum.

- Scrum es sencillo, que no quiere decir fácil. Es sencillo de comprender y empezar a utilizar pero es duro cumplir todo lo que nos requiere.

- Se necesita la implicación del cliente. La relación con el cliente es mucho más estrecha y estará al tanto en cada momento de la evolución del proyecto, de los problemas, podrá realizar cuantos cambios desee y solucionará las dudas que puedan surgir.
- El producto final es la medida del progreso.
- En Scrum se mide lo que queda, no lo que se lleva hecho. El objetivo es acabar el sprint a tiempo, con los hitos que se han marcado y con un producto final que se puede enseñar; si se mira atrás será para lamentarse o alabarse.
- El equipo es autosuficiente, se autogestiona, se autodisciplina y responde del proyecto. En Scrum se olvida un poco la idea de jefe: bueno para el empleado y bueno para el responsable. Lo importante es la gente.
- Scrum es incremental e iterativo. Un sprint tras otro y nuevos hitos que cumplir.
- Hay un control diario (Scrum diario). Esto es el corazón de Scrum.
- Sólo el equipo puede manipular la pila del sprint. Nadie ajeno al equipo puede llevar a cabo esta tarea. NUNCA se interrumpe un sprint.
- Sólo el dueño del producto maneja la pila del producto. Él es el que paga y el único que realmente sabe lo que quiere.
- Los sprints tienen una duración fija. Una vez que el equipo defina cuánto dura el sprint (un número de días entre 14 y 28) y se sienta cómodo con él no se debe variar esa cantidad.
- Retrospectivas. Tan sólo se echará la vista atrás una vez acabado el sprint para detectar problemas y errores que han sucedido.
- Es importante definir que significa que una tarea o hito esté terminado. Ej: la ropa estará lavada cuando la hayamos puesto en la lavadora con un programa adecuado, se haya secado, esté planchado y en su armario correspondiente.
- Y por encima de todo... "TIME BOXED". Todo tiene que estar estimado temporalmente y debemos ajustarnos a esas estimaciones.

Es importante tener en cuenta que los conceptos clave de Scrum son, efectivamente CLAVE. Es decir, es posible adaptar Scrum a la empresa o al proyecto, pero si se hiciera una metodología basada en Scrum en la que no se siguieran la mayoría de estos puntos clave, no se estaría haciendo Scrum. Aun así, evidentemente, se puede adaptar la metodología a la empresa o al proyecto.

3.3.9.4 Roles

Scrum define varios roles diferentes. No obstante, todos estos roles, están divididos en dos grupos:

- Grupo "pigs" (cerdos) → Los comprometidos. Son las personas pertenecientes a roles que están comprometidos con el proyecto, y el proceso scrum.
- Grupo "chickens" (gallinas) → Los involucrados. Las personas con un rol gallina, no serán parte del proceso Scrum como tal, pero debe tenérselos en consideración. Un aspecto importante de la aproximación Ágil, es la práctica de involucrar a los usuarios y clientes en el desarrollo, como parte del proceso. Es importante que estas personas se involucren, y provean feedback sobre los productos obtenidos en cada sprint.

Dentro de los "pigs" tenemos:

- **Product owner** → Este rol, representa la voz del cliente (a veces incluso puede ser el mismo cliente, si procede). El cliente debe estar dispuesto a llevar a cabo el proceso scrum. Este personaje, se asegura de que el equipo trabaja en la dirección de negocio adecuada. El product owner, escribe casos de uso, los prioritiza, y los coloca en el product backlog. Debe tener un amplio conocimiento en el área del proyecto a desarrollar, y debe conocer el negocio subyacente.
- **ScrumMaster** → El ScrumMaster, hará que el proceso scrum sea posible. Básicamente, quitará los impedimentos que puedan surgir y que impidan al equipo llegar a su objetivo del sprint. El ScrumMaster no es el líder del equipo, ya que éste se autogestiona, pero actúa de aislante entre el equipo, y las molestias externas que puedan distraer el desarrollo. En equipos inexpertos, el ScrumMáster es absolutamente necesario, y el equipo se apoyará en él durante todo el proceso Cuando el equipo es experto en Scrum, el ScrumMáster se limitará exclusivamente a eliminar los posibles problemas que pueda tener el equipo y que se detecten en las reuniones diarias principalmente.
- **El equipo** → El equipo de trabajo, será el que tiene la responsabilidad de desarrollar el producto. Un equipo, puede componerse habitualmente por entre 5 y 9 personas, guiados por un ScrumMaster. El ScrumMaster, puede guiar a más de un equipo de trabajo. Estas personas, tendrás diferentes habilidades técnicas, que se ajustarán al objetivo de este grupo para este sprint. Los grupos, no tienen por qué ser siempre los mismos, ni del mismo tamaño entre sprints. Dependiendo de lo que requiera el objetivo del sprint, la composición del grupo será diferente.

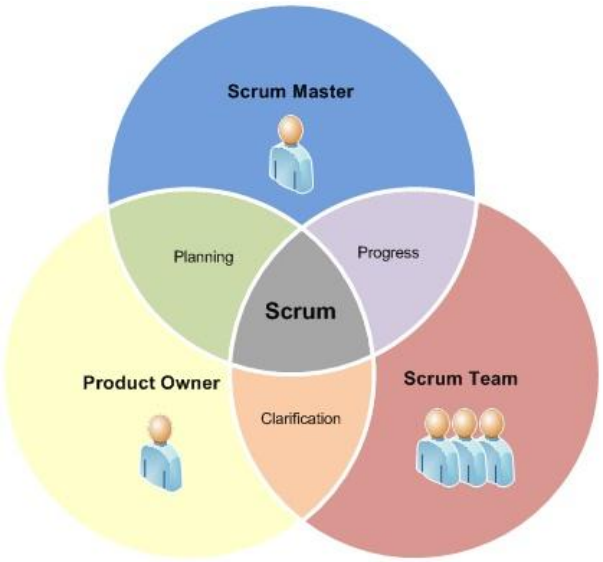


Figura 30 – Roles principales de Scrum

Por otro lado dentro de los "chickens tenemos":

- **Usuarios** → El software se construye con un fin, y ese fin, es satisfacer las necesidades de los usuarios. Si estos no pueden utilizar el software, o este no hace lo que ellos necesitan, entonces no vale para nada.

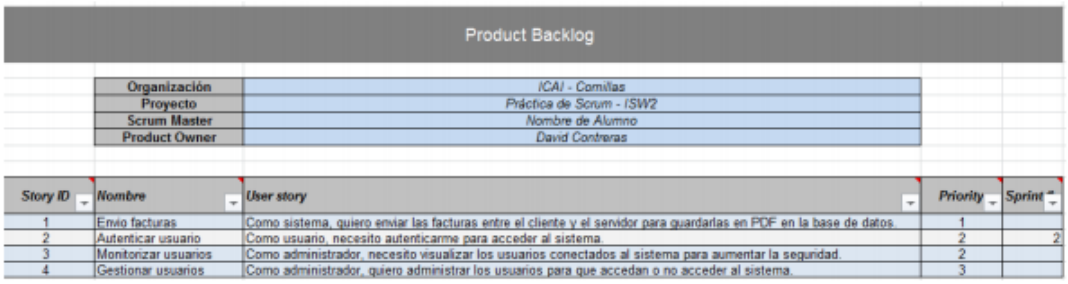
- **Stakeholders** → Estas personas, se encargarán de que el proyecto tenga rendimiento, y sea rentable. No tomarán parte directa en el proceso scrum, salvo en los sprint reviews, en los que darán su opinión sobre si el proyecto cumple sus objetivos.
- **Managers** → Serán los miembros implicados, que establezcan el entorno del producto en las organizaciones en las que se desarrollará.

3.3.9.5 Documentos esenciales

Documentos no convencionales que deben de formar parte del proceso de Scrum.

Pila de producto (Product Backlog)

Se describen de forma reducida todos los requerimientos (user stories) priorizados y con una estimación de tiempos. También se muestra información global del desarrollo. La gestión del Product Backlog se puede llevar a cabo mediante una simple hoja de cálculo o a través de herramientas software más completas.



Product Backlog

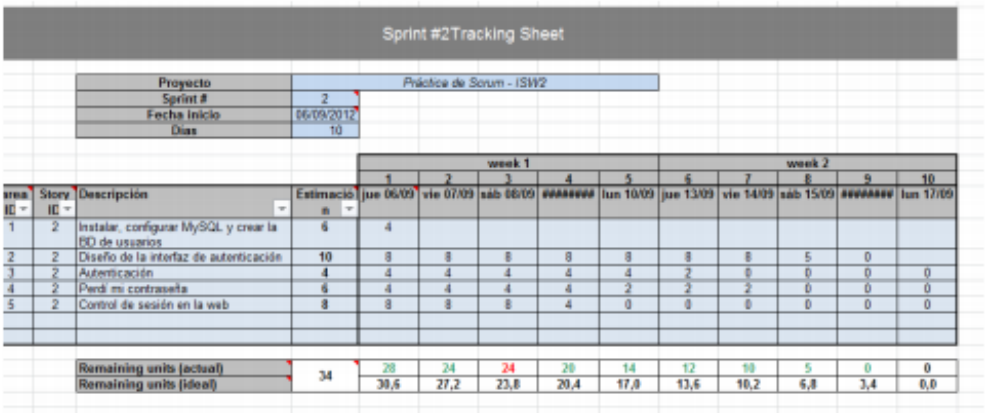
Organización	ICAI - Comillas
Proyecto	Práctica de Scrum - ISW2
Scrum Master	Nombre de Alumno
Product Owner	David Contreras

Story ID	Nombre	User story	Priority	Sprint
1	Envio facturas	Como sistema, quiero enviar las facturas entre el cliente y el servidor para guardarlas en PDF en la base de datos	1	
2	Autenticar usuario	Como usuario, necesito autenticarme para acceder al sistema	2	2
3	Monitorizar usuarios	Como administrador, necesito visualizar los usuarios conectados al sistema para aumentar la seguridad.	2	
4	Gestionar usuarios	Como administrador, quiero administrar los usuarios para que accedan o no acceder al sistema.	3	

Figura 31 – Ejemplo de Product Backlog

Pila de Sprint (Sprint Backlog)

Es un subconjunto de la pila de producto. Un documento detallado de los requerimientos (user stories) de cada sprint. Estos se pueden subdividir en tareas (items) de 4-16 horas. El producto de software resultante debe ser potencialmente usable. Para decidir la carga máxima de trabajo para cada sprint se define una puntuación máxima al equipo (trabajo máximo que puede soportar) y una puntuación a cada uno de los requerimientos. Los primeros sprints a desarrollar suelen abordar las historias de usuario o requerimientos más críticos.



Sprint #2Tracking Sheet

Proyecto	Práctica de Scrum - ISW2
Sprint #	2
Fecha Inicio	06/09/2012
Dias	10

					week 1					week 2				
					1	2	3	4	5	6	7	8	9	10
area ID	Story ID	Descripción	Estimación		jue 06/09	vie 07/09	sáb 08/09	########	lun 10/09	jue 13/09	vie 14/09	sáb 15/09	########	lun 17/09
1	2	Instalar, configurar MySQL y crear la BD de usuarios	6		4									
2	2	Diseño de la interfaz de autenticación	10		8	8	8	8	8	8	8	5	0	
3	2	Autenticación	4		4	4	4	4	4	2	0	0	0	0
4	2	Perdí mi contraseña	6		4	4	4	4	2	2	2	0	0	0
5	2	Control de sesión en la web	8		8	8	8	4	0	0	0	0	0	0
		Remaining units (actual)	34		28	24	24	20	14	12	10	5	0	0
		Remaining units (ideal)			30,6	27,2	23,8	20,4	17,0	13,6	10,2	6,8	3,4	0,0

Figura 32 – Ejemplo de SprintBacklog

User Stories

Los requisitos o los casos de uso en UML, describen de forma muy breve la funcionalidad o capacidades del sistema desde el punto de vista del usuario. Deben ser independientes e indicar:

- ¿Quién es el usuario que se beneficia?
- ¿Qué acción lleva a cabo?
- ¿Cuál es el beneficio?

Siempre se redactan según la siguiente plantilla: As a <type of user>, I want <some goal> so that <some reason>. "Como profesor, quiero listar los alumnos con nota superior a 6 para saber quién libera el examen".

El equipo de proyecto debe decidir la descomposición de la historia de usuario en tareas, estimando la duración de cada una de ellas y la asignación a los miembros del equipo.

Gráfico BurnDown

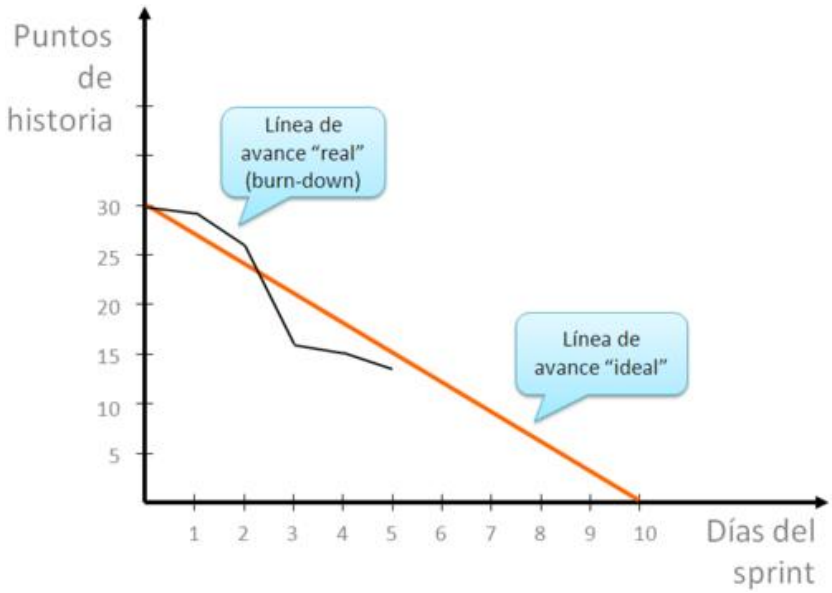


Figura 33 – Gráfico Burndown Scrum

El Gráfico de Burn-Down muestra cuánto le falta al equipo para completar con el compromiso del Sprint. Sobre el eje horizontal se ubican los días hábiles que tiene el Sprint, y sobre el eje vertical la cantidad de puntos que tienen quemar durante ese Sprint.

3.3.10 Scrumban

Scrumban es una metodología derivada y combinación de los métodos de desarrollo Scrum y Kanban.

Características	Scrum	Kanban
TIEMPO POR ITERACIÓN	FIJO	OPCIONAL
EQUIPO	MULTIFUNCIONAL	MULTIFUNCIONAL O ESPECIALIZADO
TAMAÑO DE EQUIPO	Máximo 10 (aprox.)	SIN PRESCRIPCIÓN
DIAGRAMAS	BURNDOWN CHART, ETC.	OPCIONAL
ESTIMACIÓN	LA NECESARIA (PLANNING POKER)	OPCIONAL
SPRINT BACKLOG	RÍGIDA	SE PUEDEN AÑADIR TAREAS
ROLES	SCRUM MASTER, PRODUCT OWNER y EQUIPO	SIN PRESCRIPCIÓN

Figura 34 – Diferencias entre Scrum y Kanban

En ella se pueden usar las mejores prácticas ágiles de cada una. De esta forma tiene origen el Scrum-ban.

Básicamente sigue el flujo de trabajo continuo como lo define Kanban, pero se incluyen elementos de Scrum como, por ejemplo, las reuniones diarias de 15 minutos y pequeñas retrospectivas con el afán de mejorar el proceso.

Así podemos decir que de Scrum toma:

- Roles → Cliente, equipo (con los diferentes perfiles que se necesiten).
- Reuniones → Reunión diaria.
- Herramientas → Pizarra

De Kanban:

- Flujo visual
- Hacer lo que sea necesario, cuando sea necesario y solo la cantidad necesaria.
- Limitar la cantidad de trabajo (WIP)
- Optimización del proceso.

Normas	Scrum	Scrumban
Pizarra / Herramientas	Pizarra Backlogs Gráfica burn-down	Pizarra
Reuniones	Reunión diaria Planificación Retrospectiva	Reunión diaria
Iteraciones	Sí, Sprints	No, flujo continuo
Estimaciones	Sí	No
Esquipo	Multidisciplinar	Puede ser especializado
Roles	Product Owner Scrum Master Equipo	Equipo + otros
WIP (Work In Progress)	Controlado por el contenido del Sprint	Controlado por el estado de la tarea.
Cambios	Se pasan al siguiente Sprint	Se añaden al tablero en la columna "TO DO".
Impedimentos	Solución inmediata	Se evitan.

Figura 35 – Diferencias entre Scrum y Scrumban

Es un modelo de desarrollo especialmente adecuado para proyectos de mantenimiento o proyectos en los que las historias de usuarios (requisitos del software) varíen con frecuencia o en los cuales surjan errores de programación inesperados durante todo el ciclo de desarrollo del producto. Para estos casos, los sprints (periodos de duración constante en los cuales se lleva a cabo un trabajo en sí) de la metodología Scrum no son factibles, dado que los errores/impedimentos que surgirán a lo largo de las tareas son difíciles de determinar y por lo tanto, no es posible estimar el tiempo que conlleva cada historia. Por ello, resulta más beneficioso adoptar flujo de trabajo continuo propio del modelo Kanban.

3.4 Técnicas Ágiles

En este apartado se explican algunas de las principales técnicas usadas en el desarrollo ágil. Vamos a explicar algunas de las más extendidas entre las metodologías.

3.4.1 Diagrama BurnDown

El diagrama de burndown, es una representación gráfica del trabajo por hacer (y por supuesto también del trabajo hecho) en el tiempo. Este diagrama normalmente se utiliza para predecir cuándo se completara un trabajo concreto en base a una temporalidad concreta.

Por medio de este diagrama, se representa el trabajo actual en comparación con el trabajo ideal, y permite de un solo vistazo, detectar si se va adelantado o retrasado respecto al trabajo planificado a realizar durante el Sprint. El diagrama de burndown, es muy utilizado en las metodologías de desarrollo ágil de software (agile), y sobre todo en Scrum.

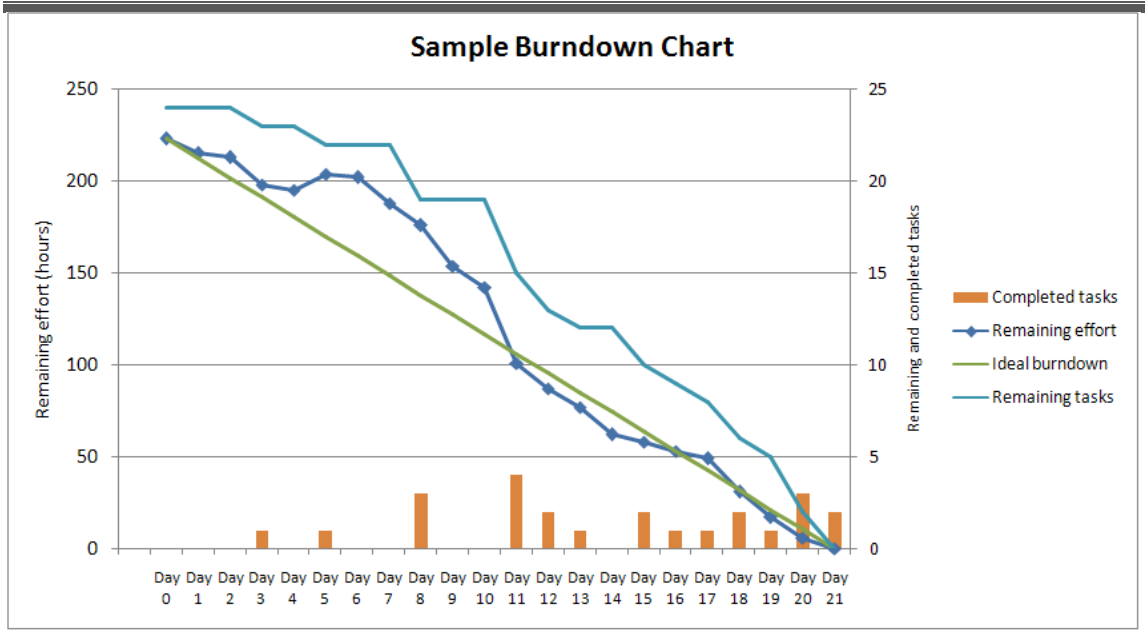


Figura 36 – Ejemplo de Diagrama BurnDown

Usualmente el trabajo remanente (Backlog) se muestra en el eje vertical y el tiempo en el eje horizontal. Por lo que el diagrama representa el trabajo pendiente, útil para predecir el trabajo pendiente.

3.4.2 Estimación Planing Poker

Planning Poker es una técnica de estimación de software puesta en marcha por primera vez por James W. Grenning en un equipo de desarrollo Ágil utilizando XP en 2002.

Los resultados positivos del Planning Poker se derivan de la combinación del uso de opinión experta (quien normalmente realiza la tarea la estima), analogía (puesto que un gran número de historias tienen similitud con el trabajo desarrollado previamente por el equipo) y división del esfuerzo de estimación (en historias de un tamaño adecuado para evitar que la desviación sea elevada).

En las sesiones de Planning Poker debe jugar todo el equipo de desarrollo y aunque determinadas historias no afecten por igual a todos los integrantes del equipo será necesaria la participación activa de todos. El otro role necesario es el moderador que será el encargado de que los participantes lleguen al consenso.

Planning Poker está basado en una lista de características para ser realizadas y una baraja de cartas. La lista de características 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. Un mazo que se encuentra en el mercado utiliza la siguiente secuencia: 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, 100, y, opcionalmente, una tarjeta "?" (inseguro) y una "taza de café" (necesito un descanso).

Figura 37 – Baraja Planing Poker

Las cartas están numeradas de esta forma para explicar el hecho de que, cuanto una estimación es mayor, existe **mayor incertidumbre**. Así, si un desarrollador quiere jugar un 6 se ve obligado a reconsiderar y aceptar que parte de la incertidumbre percibida no existe y jugar un 5, o aceptar una estimación más conservadora de la incertidumbre y jugar un 8.

Cómo se realiza

En la reunión de la estimación a cada estimador se le da un conjunto de las tarjetas. La reunión prosigue de la siguiente manera:

- Un moderador, que no jugará, preside la reunión.
- El desarrollador con más conocimiento de una determinada característica o el moderador proporcionan una breve introducción sobre la misma. El equipo tiene la oportunidad de hacer preguntas y discutir para aclarar los supuestos y riesgos.
- 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, horas... Durante el debate, los números no debe ser mencionados en absoluto.
- Todo el mundo muestra sus tarjetas 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. En este aspecto existen diferentes variantes.
- Se descartan las estimaciones que se alejen mucho de la media aproximada de las estimaciones.
- Se les permite a los estimadores extremos volver a participar en el siguiente descarte con respecto a las mismas tareas tras una nueva discusión de la tarea a realizar.

- Se repite el proceso de cálculo hasta que se alcance un consenso.
- Se puede utilizar un reloj de arena para asegurar que el debate sea estructurado, el moderador o el Scrum Máster podrá en cualquier punto terminar el reloj y cuando se acaba toda discusión debe cesar y otra ronda de póquer se juega.

3.4.3 Estimación Wideband Delphi

Desde un punto de vista ágil, y en base a los principios de las metodologías ágiles, se da una vuelta de tuerca al método Delphi (Juicio del Experto), obteniendo así esta técnica de estimación, basada en los mismos principios que su técnica origen, pero aportando los beneficios obtenidos por la agilidad, es decir, menor carga documental, mayor rapidez y compromiso absoluto.

Cómo se realiza

Los pasos que se realizan en esta reunión, se pueden resumir en los siguientes:

1. Se reúne al grupo de participantes, idealmente de dos a cinco personas. Se contará con personas que hayan trabajado en aplicaciones similares y personas que no, es interesante contar con diferentes perspectivas.
2. Cada persona contará con una descripción general de la cuestión a estimar y quien mejor la conozca expondrá de viva voz sus conocimientos sobre la misma.
3. Cada persona presente en la reunión de estimación realizará y anotará su estimación sin colaborar con los demás. No se mostrará aun esta estimación al resto de participantes.
4. Un facilitador revela los cálculos de manera anónima y seguidamente, tiene lugar un debate sobre las suposiciones en que se basan los cálculos. Quien lo desee puede revelar cuál fue su estimación.
5. El paso 3 se repite hasta que los cálculos converjan. Lo que se pretende es que cada persona aprende de los demás participantes, actualiza sus estimaciones y proporcione una nueva.

Como se puede observar, la técnica está basada en Delphi, pero es mucho más ágil y rápida. No requiere una carga de documentación excesiva, ni requiere un gran número de reuniones. La documentación que entrará en juego con esta técnica será las anotaciones de los expertos y las conclusiones que anota el facilitador.

3.4.4 Integración Continua

Según Martin Fowler: *"La integración continua es una práctica de desarrollo de software en la cual los miembros de un equipo integran su trabajo frecuentemente, como mínimo de forma diaria. Cada integración se verifica mediante una herramienta de construcción automática para detectar los errores de integración tan pronto como sea posible. Muchos equipos creen que este enfoque lleva a una reducción significativa de los problemas de integración y permite a un equipo desarrollar software cohesivo de forma más rápida."*

El proceso de integración continua tiene como objetivo principal comprobar que cada actualización del código fuente no genere problemas en una aplicación que se está desarrollando. La integración continua fue utilizada por IBM para el desarrollo del OS/360 en los años 60.

Los miembros de un equipo de desarrollo integran el programa sobre el que trabajan con frecuencia. La integración continua consiste en activar en cada integración un proceso basado en una plataforma que verifica automáticamente el funcionamiento de la aplicación para que las anomalías sean detectadas lo más pronto posible.

Lo más difícil para el desarrollador es conocer el impacto real de una actualización fundamental sobre todas las funcionalidades de la aplicación. La integración continua permite al desarrollador tener esta visión más global de la aplicación ya que las pruebas de la aplicación se hacen sobre un entorno similar de producción.

A continuación se describen las principales prácticas de integración continua:

- Mantener un único repositorio de código fuente
- Automatizar la construcción del proyecto
- Hacer que la construcción del proyecto ejecute sus propios tests
- Entregar los cambios a la línea principal todos los días
- Construir la línea principal en la máquina de integración
- Mantener una ejecución rápida de la construcción del proyecto
- Probar en una réplica del entorno de producción
- Hacer que todo el mundo pueda obtener el último ejecutable de forma fácil
- Publicar qué está pasando
- Automatizar el despliegue

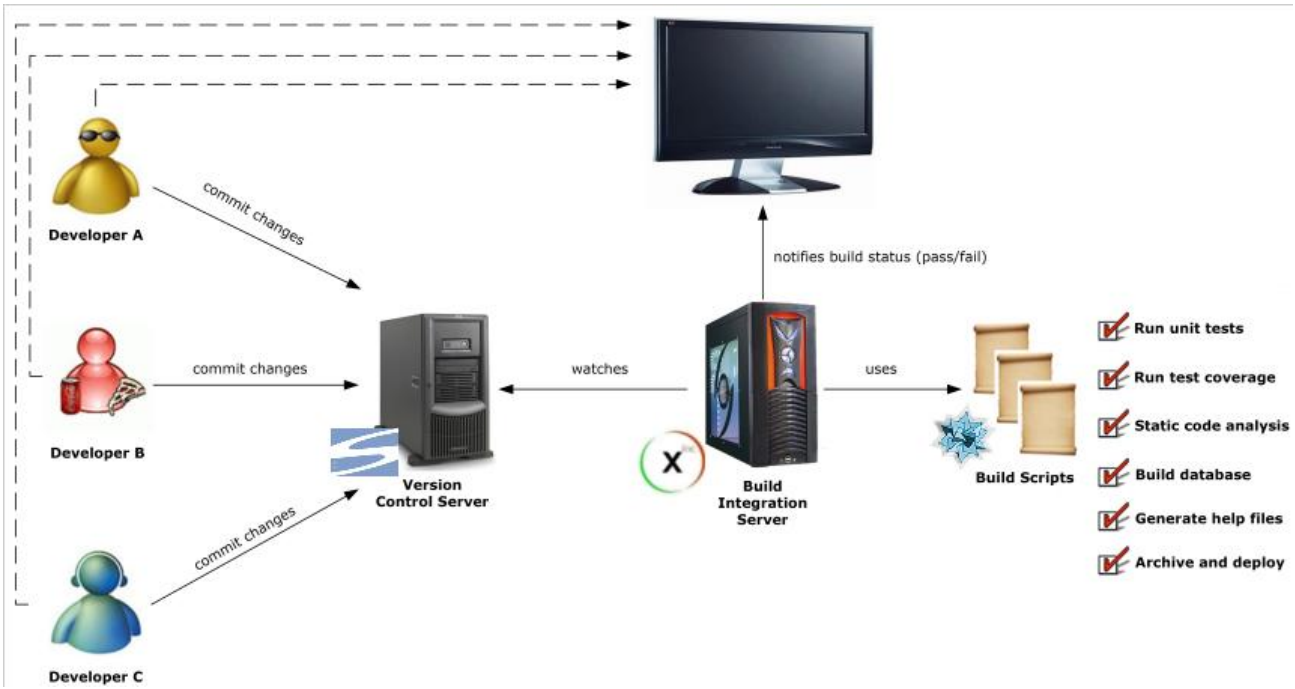


Figura 38 – Esquema de un proceso de integración continua

Algunos términos comunes dentro de la integración continúa:

- **Repositorio** → Es el almacén donde se guardan todos los ficheros y sus versiones. Todas las operaciones registran un punto de referencia en el repositorio. por ejemplo, hacer un check-out (extraer un fichero del repositorio) o un commit (incorporar un fichero al repositorio).
- **Build** → Es el total de etapas necesarias para la compilación, creación de entregables al momento de ejecutar los test (funcional, unitarios, etc.)
- **Commit** → Es la operación que incorpora ficheros al repositorio.
- **Update** → Es la operación de actualizar ficheros del repositorio.
- **Checkout** → Es la operación de extraer uno o varios ficheros del repositorio para una versión concreta.

Así un proceso de integración continua sería como sigue:

1. El desarrollador hace un commit sobre el referencial del administrador de configuración.
2. El servidor de integración continua detecta el commit, hace un checkout ejecuta las operaciones de compilación y de test.
3. En caso de error se genera una notificación para el jefe del proyecto y/o al equipo de desarrollo
4. El desarrollador que cometió el error hace un update del referencial de gestión de configuración y corrige el problema.

Por último remarcar que cuando se trabaja con repositorios, se suele utilizar una estructura formada por tres subdirectorios: trunk, branches y tags:

- **Trunk** → Es el directorio que contiene la última versión del proyecto que se está desarrollando, la versión común a todos los desarrolladores.
- **Branches** →Este directorio contiene varios subdirectorios, uno para cada versión de "Trunk" que se quiera hacer. Es normal que cada desarrollador trabaje sobre su propio subdirectorio en "Branches", y que después ponga todo en común en el directorio "Trunk" con un merge. Cada subdirectorio de "Branches" normalmente se inicializará haciendo una copia de "Trunk".
- **Tags** → En "Tags" se guardan copias de seguridad (snapshots) del proyecto. La única diferencia entre los directorios "Tags" y "Branches" es que nunca se modifican ficheros pertenecientes al directorio "Tags".

3.4.5 Moscow

Es una técnica de priorización utilizada en el estudio de negocio y desarrollo de software con el fin de entender la importancia de los interesados en la entrega de cada requerimiento.

Las categorías a asignar a cada requisito a implementar sería:

- **M-MUST (Debe)** → Define un requisito que debe ser satisfecho para que la solución final sea considerada un éxito.

- **S-SHOULD (Debería)** → Representa un requisito de alta prioridad que debe ser incluido en la solución final de ser posible.
- **C-COULD (Podría)** → Describe un requisito que es deseable pero no necesario. Este puede ser incluido si el tiempo y los recursos lo permiten.
- **W-WON´T** → Representa un requisito que los interesados han acordado que no sea implementado en la versión saliente pero si en el futuro.

Estableciendo estas categorías a nuestros requisitos podremos priorizar mejor a la hora de elegir que funcionalidad debe ser la siguiente en ser implementada.

3.4.6 Programación en Parejas

El "pair programming" apareció a partir de la metodología Extreme Programming, y se basa en que dos programadores trabajan juntos en un solo ordenador. Uno de ellos desarrolla mientras que el otro ayuda y revisa el código del compañero. La persona que está haciendo la codificación se le da el nombre de controlador mientras que a la persona que está dirigiendo se le llama el navegado.

De esta técnica se espera que mejore la productividad, la transferencia de conocimiento y, obviamente, el trabajo en equipo, pero aun así son muchas las dudas, miedo y críticas que levanta esta técnica.

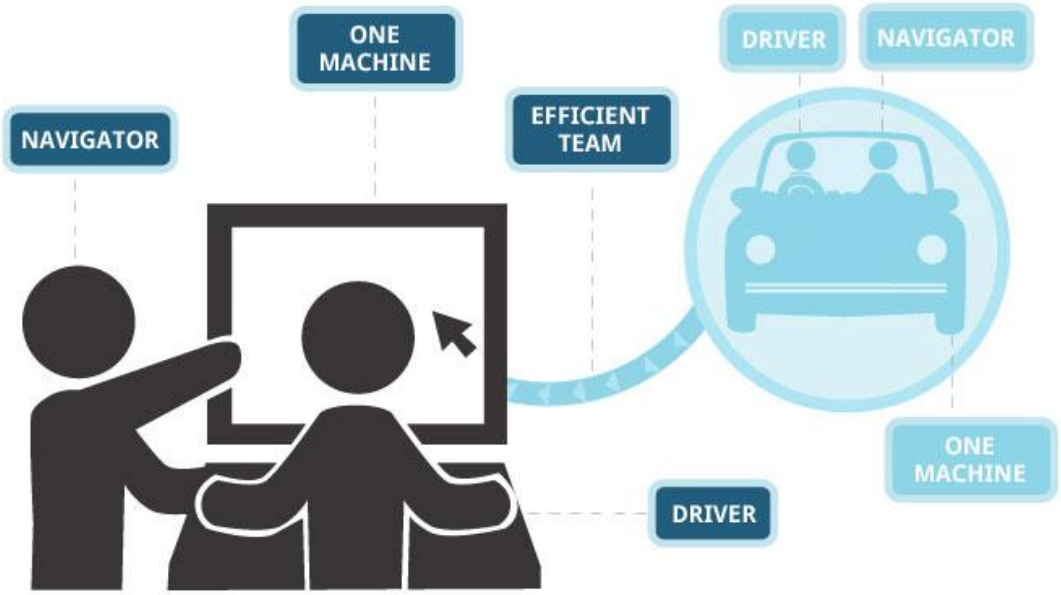


Figura 39 – Programación en parejas similitud con rallyes

A continuación se exponen algunas pautas a seguir para el éxito de esta técnica:

1. **Comenzar con una tarea bien definida** → Cuando los dos se pongan a trabajar deben tener una tarea correctamente detallada y deben tener la seguridad de que van a poder terminarla en una hora o dos

2. **Compartan todo** → Todo se refiere a que los dos programadores funcionen como una sola mente, mientras uno está en el teclado, el otro está revisando todo y dando retroalimentación. Es importante que se intercambien el turno de escribir y revisar para que uno de los dos no se sienta poco importante o fuera del juego.
3. **Hablen todo lo que sea necesario** → Una buena señal es que los dos implicados estén hablando todo el tiempo. Diciendo lo que van a hacer, pidiendo una idea de una implementación, preguntando la mejor manera de hacer algo, dando soluciones alternativas, etc.
4. **No tomarse las cosas muy a pecho** → Cuando una persona tiene el ego muy alto tiende a no aceptar críticas de los demás. También el ego excesivo puede atribuirle una actitud defensiva ante las críticas al programador, o puede pensar que los que se le dice es personal y en su contra.
5. **Celebrar los éxitos** → A medida que se vayan completando las tareas hay que felicitarse, por ejemplo, se resolvió ese problema con la base de datos, chócala!

3.4.7 Programación Lado a Lado

Basado en la técnica de la programación en parejas, esta técnica, nacida de la metodología Crystal, establece proximidad, pero cada quien se enfoca a su trabajo asignado, prestando un ojo a lo que hace su compañero, quien tiene su propia máquina. Esta es una ampliación de la Comunicación Osmótica al contexto de la programación.

3.4.8 Refactoring

Refactorizar (o Refactoring) es realizar una transformación al software preservando su comportamiento, modificando sólo su estructura interna para mejorarlo. El término es de Opdyke, quien lo introdujo por primera vez en 1992, en su tesis doctoral.

La refactorización puede entenderse como un proceso de mantenimiento de un Sistema de Información cuyo objetivo no es ni arreglar errores ni añadir nueva funcionalidad, si no mejorar la comprensión del código a través de su restructuración aplicando, por ejemplo, los Principios de la Orientación a Objetos, algún Patrón de Diseño o simplemente cambiando el algoritmo utilizado o eliminando código muerto, para facilitar así futuros desarrollos, la resolución de errores o la adición de nuevas funcionalidades. No son una optimización del código, ya que esto en ocasiones lo hace menos comprensible, ni solucionar errores o mejorar algoritmos.

Aunque hay varios catálogos de refactorizaciones el más famoso es el de Martin Fowler, que se mantiene en la página www.refactoring.com. Algunos ejemplos de las refactorizaciones que podéis encontrar en este catálogo: Add Parameter, Change Bidirectional Association to Unidirectional, Consolidate Conditional Expression, Extract Class, Introduce Null Object, Move Method, etc.

En la actualidad, los entornos de desarrollo suelen ofrecer herramientas para aplicar algunos métodos de refactorización de manera automática, actualizando correctamente las

dependencias (siempre que sea posible) o marcando los elementos afectados, simplificando en gran medida este proceso.

3.4.9 Reunión - Encuentros Diarios de Pie

Técnica surgida de la metodología Ágil (Scrum), este tipo de reunión (también llamada "Daily Stand up Meeting") es una vuelta de tuerca a las reuniones tradicionales para que realmente sirvan como un mecanismo de colaboración y no como una frustrante obligación y pérdida de tiempo. El objetivo de esta reunión es facilitar la transferencia de información y la colaboración entre los miembros del equipo para aumentar su productividad, al poner de manifiesto puntos en que se pueden ayudar unos a otros.

Cada miembro del equipo inspecciona el trabajo que el resto está realizando (dependencias entre tareas, progreso hacia el objetivo de la iteración, obstáculos que pueden impedir este objetivo) para al finalizar la reunión poder hacer las adaptaciones necesarias que permitan cumplir con el compromiso conjunto que el equipo adquirió para la iteración (en la reunión de planificación de la iteración).

Cada miembro del equipo debe responder las siguientes preguntas en un timebox de cómo máximo 15 minutos:

1. ¿Qué he hecho desde la última reunión de sincronización? ¿Pude hacer todo lo que tenía planeado? ¿Cuál fue el problema?
2. ¿Qué voy a hacer a partir de este momento?
3. ¿Qué impedimentos tengo o voy a tener para cumplir mis compromisos en esta iteración y en el proyecto?

Figura 40 – Ejemplo de Encuentro Diario de Pie

Así se brindan una forma sencilla para que los miembros del equipo se mantengan al tanto de lo que pasa sin tener que pasar mucho tiempo en reuniones o tener que leer y escribir pilas de reportes de estado. Se enfocan en una discusión rápida de progresos, planes y problemas. Esto hace posible a los miembros del equipo obtener actualizaciones oportunas sobre el progreso de otros miembros mientras el administrador de proyecto determina rápidamente sus tareas del día en virtud de los obstáculos del equipo. En esencia, esta técnica es verdaderamente un mecanismo de colaboración para el equipo.

Recomendaciones para que estas reuniones tengan éxito:

- **Diaria** → Si no se puede reunir al grupo todos los días, al menos intentarlo día por medio.
- **De Pie** → Intentarlo, aunque pueda resultar incómodo, precisamente por esta razón las reuniones serán breves... y de mayor provecho.
- **Puntualidad** → Consensuar un único horario de inicio y duración, respetarlo aunque no estén todos los integrantes del equipo, y no permitir que se interrumpa con llamados a celulares o internos de la oficina ni llegada tarde a la reunión.
- **10-15 minutos** → 5 es poco, 10 a 15 está bien, más de 20 ya es una reunión tradicional.
- **Grupo de trabajo** → Es factible con un equipo pequeño y en el que todos compartan la misma problemática de trabajo cotidiana
- **Reunirse en círculo** → Facilita la comunicación, todos pueden además de escuchar ver a la cara al que habla, aumentando en un 100% la comprensión.
- **Temario** → Es conveniente que el coordinador de la reunión pregunte antes los temas a tratar, los exponga al comenzar y no se agreguen nuevos en su transcurso
- **Útiles** → Se puede reunir alrededor de una mesa pero no es necesario, tal vez convenga tener un pizarrón en la misma sala de reunión, un proyector es ya demasiada infraestructura para esta técnica y le restaría agilidad.
- **PPP** → Tratar rápidamente Progresos, Planes y Problemas, darle prioridad a lo importante, no sólo a lo urgente.
- **Marcar claramente el final** → El coordinador de la reunión debe con un gesto propio que identifique dar fin a la reunión, por ejemplo con tres aplausos.
- **"Me gusta"** → El coordinador debe buscar entre los asistentes el "Me gusta" de cada reunión al terminar para mantenerlos y fortalecerlos.

3.4.10 TaskBoard

Esta técnica, que fue popularizada por la metodología Scrum (Scrum Board), permite llevar un control y seguimiento sobre la lista de tareas del proyecto.

Básicamente se trata de una tabla que muestra, al menos, tres columnas para clasificar el trabajo del equipo, las tareas, entre “Pendiente”, “En Progreso”, “Hecho”. El número de columnas puede ser variable. Existen muchas variantes, diferentes plantillas, número de filas, columnas….

Idealmente, el taskboard es un elemento físico, y consistirá de tarjetas de notas o postits (una por cada tarea) y un muro o sitio donde pegarlas, aunque en equipos distribuidos se podrían usar aplicaciones que simulan estas pizarras online.

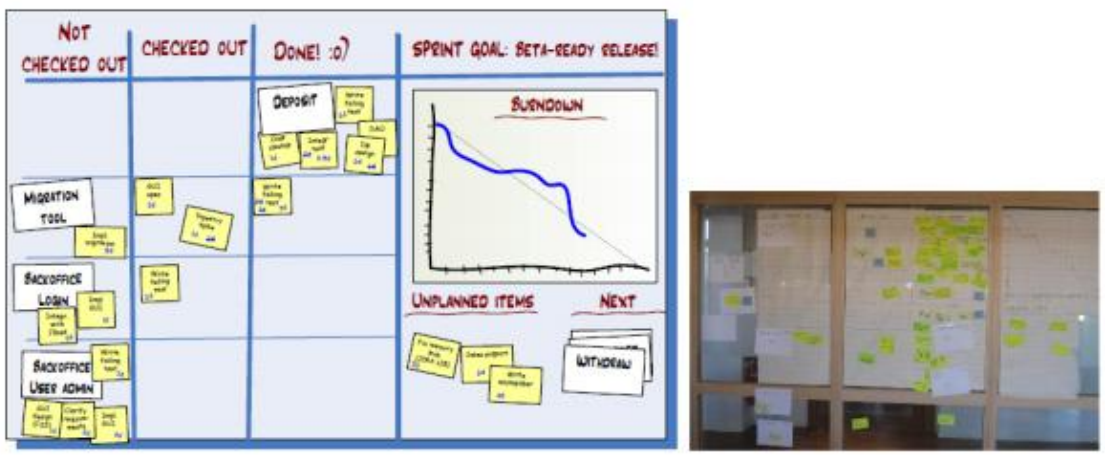


Figura 41 – Ejemplo de TaskBoard en Scrum

A medida que el estado de una tarea vaya cambiado, se irá cambiando su posición en la taskboard. Esta actualización se realiza frecuentemente, normalmente en el daily meeting (técnica comentada en el punto 3.4.8). Y normalmente la pizarra es "reseteada" una vez que la iteración ha terminado.

Con esta técnica conseguimos que:

- La difusión de la información → Todo el mundo sabe el estado de las tareas de cada uno.
- Tener un punto común de atención → Será el punto de reunión del equipo.
- Simplicidad y flexibilidad → Simple de aplicar y fácil de adaptar a las necesidades del equipo, quizás unos postits de distintos colores para marcar prioridades…

Figura 42 – Ejemplo de representación de una tarea en una tarjeta para TaskBoard

3.4.11 Test Driven Development

TDD son las siglas de Test Driver Development un proceso de desarrollo de software cuyos principios son realizar pruebas, codificar y refactorizar el código construido.

TDD se basa en la idea de realizar pruebas unitarias para el código que debemos construir. Pero a diferencia del procedimiento que se usa habitualmente (construir el código y después realizar las pruebas unitarias) TDD establece que primero hay que realizar una prueba y a continuación desarrollar el código que la resuelve.

Ciclo de desarrollo de TDD:

1. Elegir un requisito a desarrollar
2. Crear la prueba o test
3. Ejecutar los tests: falla (ROJO)
4. Crear código específico para resolver el test
5. Ejecutar de nuevo los tests: pasa (VERDE)
6. Refactorizar el código
7. Ejecutar los tests: pasa (VERDE)

El objetivo de esta técnica es realizar un ejercicio previo de análisis, en profundidad, de los requisitos y de los diversos escenarios. Eliminando la mayor parte de variabilidad y encontrado aquellos aspectos más importantes o no contemplados en los requisitos.

El hecho que además solo se implemente el código necesario para resolver un caso de prueba concreto, pasar la prueba, hace que el código creado sea el mínimo necesario, reduciendo redundancia y los bloques de código muerto.

No obstante hay que remarcar que TDD no solo se basa en las pruebas. Una correcta aplicación de la etapa de refactoring hace que nuestro código sea más legible, óptimo y fácil de mantener.

Capítulo 4. Acercamiento al Problema

4.1 Desconocimiento en la Gestión Ágil

Las metodologías ágiles, y en especial Scrum, están siendo un boom en el mundo del software en los últimos años. Gerentes, jefes de proyecto y programadores, toda la profesión, está asistiendo a cursos o ha leído algo sobre esta nueva tendencia con la que dicen se solucionan todos los males del sector.

El problema es que la información que están recibiendo es parcial. En la mayoría de esos cursos se centran en hablar de todo lo malo que tienen otros modelos de ciclo de vida, en explicar el manifiesto ágil y luego exponer Scrum.

Luego está el problema de que en las organizaciones todavía no se atreven a implantar estas metodologías de manera efectiva, sino que se están haciendo pruebas en proyectos no muy grandes.

Y finalmente está el problema de los clientes, estos a los que hay que convencer de que para que el proyecto salga bien, no vale con que paguen, sino que también se necesita su colaboración, y muy cercana.

Por lo tanto se pueden identificar varios problemas:

1. Formación o información parcial recibida por parte de los profesionales de la informática.
2. Organizaciones temerosas de implantar este modelo de ciclo de vida donde el control es menos exhaustivo, al menos en base a documentos.
3. Clientes que pueden llegar a ver excesivo la carga de trabajo que de ellos se necesita.

Los puntos 2 y 3 realmente son un problema temporal, el paso de tiempo, y los buenos resultados que vayan surgiendo en proyectos llevados con estas metodologías ágiles, harán que poco a poco tanto las capas altas de las organizaciones como los clientes, se decidan a usar ágil sin miedo y comprometidos con su forma de actuar.

Pero creo que el gran problema es el punto 1. En cursos o masters como este para el que se presenta el proyecto, apenas se habla de pasada de las metodologías ágiles. En la mayoría de los grados de informática, se sigue dedicando la mayor parte del tiempo a explicar modelos tradicionales, dejando el ágil apartado.

Y luego tenemos la otra cara de la moneda, mucha gente se está formando en metodologías ágiles, sin saber que otros modelos de ciclo de vida existen, cuando es conveniente usar cada uno, y sin ni si quiera saber hay más alternativas ágiles a parte de Scrum o XP.

En este capítulo se mostrarán las ventajas y desventajas que tiene cada uno de los modelos, sin demonizar a ninguno. Y también se comparará cada una de las metodologías ágiles.

4.2 Ventajas y Desventajas de los Modelos Existentes

En el punto 3.2 se han explicado los distintos modelos de ciclo de vida que se tienen disponibles a la hora de elegir un modelo de ciclo de vida, pero con la información allí transmitida no es suficiente. Se hace necesario al menos saber cuáles son las ventajas y desventajas de cada modelo.

Esta información junto con la recopilada en el punto 3.2 será muy útil más adelante, en el punto 5.2, para descubrir qué modelo de ciclo de vida se adapta mejor al proyecto, equipo, cliente y organización en cuestión.

A continuación se enumeran las ventajas y desventajas que tendrá cada uno de los modelos de ciclo de vida.

4.2.1 Modelo en Cascada

Ventajas:

- Ampliamente utilizado y conocido (En teoría).
- Fácil entendimiento e implementación.
- Todo está bien organizado y no se mezclan las fases.
- Identifica entregables e hitos. Se muestra de forma explícita qué productos intermedios se tienen que obtener antes de abordar las siguientes tareas.
- Fácil de gestionar ya que cada fase tiene entregables específicos y un proceso de revisión. Las fases son procesadas y completadas de una vez.
- Orientado a documentos.
- Planificación sencilla.
- Provee un producto con un elevado grado de calidad sin disponer de un personal altamente cualificado.
- Refuerza buenos hábitos: definir antes que diseñar, diseñar antes que codificar.
- Su simplicidad hace que resulte sencillo explicárselo a los clientes que no están familiarizados el proceso software.

Desventajas

- Es rígido, poco flexible y con muchas restricciones
- Necesita conocer todos los requisitos al comienzo del proyecto, lo cual la mayoría de las veces es muy difícil o irreal.
- Los modelos generados (catálogo de requisitos, casos de uso, etc…) serán lo suficientemente grandes como para no poder ser revisados y comprendidos en toda su magnitud, sobre todo por personal no informático.
- Los resultados no se ven hasta en las etapas finales del ciclo.
- No se obtiene feedback de los involucrados hasta casi al final.

- Normalmente crea inseguridad en un cliente que preferiría ir viendo resultados poco a poco.
- Si se han cometido errores y no se detectan en la etapa inmediatamente siguiente, es costoso y difícil volver atrás para realizar la corrección.
- Cualquier error detectado nos trae un retraso y aumenta el coste del desarrollo. Y este va aumentando según avanzamos de fase.
- Genera un alto grado de riesgo e incertidumbre. Dificultad para administrar el riesgo.
- Hacer cambios es difícil y muy costoso.
- La Fase de pruebas es una de las últimas fases, lo cual significa que estamos dejando de descubrir errores que seguramente aparecerán esta fase, la cual ya está muy cercana a la entrega.
- La naturaleza lineal del modelo en cascada conduce a "estados de bloqueo" en los cuales algunos miembros del equipo del proyecto deben esperar a otros para terminar tareas dependientes.

4.2.2 Modelo en V

Este modelo es lo suficientemente parecido al cascada como para tener una ventajas y desventajas prácticamente iguales, pero por su naturaleza de pruebas podemos destacar o añadir alguna ventaja más, pero también alguna desventaja.

Ventajas

- Las pruebas de cada fase ayudaran a corregir posibles errores sin tener que esperar a que sean rectificados en la etapa final del proceso.
- Con las pruebas unitarias y de integración se consigue obtener exactitud en los programas.
- Involucra al usuario en las pruebas.
- Especifica bien los roles de los distintos tipos de pruebas a realizar.

Desventajas

- El modelo no proporciona caminos claros para problemas encontrados durante las fases de pruebas.
- Al encontrarse errores luego de realizar las pruebas se pierde tiempo y dinero, ya que cada prueba se realiza luego de haber terminado la implementación.

4.2.3 Modelo Iterativo

Ventajas

- Una de las principales ventajas que ofrece este modelo es que no hace falta que los requisitos estén totalmente definidos al inicio del desarrollo, sino que se pueden ir refinando en cada una de las iteraciones.
- Permite crear cada vez versiones más completas de software.
- Resolución de problemas de alto riesgo en tiempos tempranos del proyecto.

- Visión de avance en el desarrollo desde las etapas iniciales del desarrollo.
- Menor tasa de fallo del proyecto, mejor productividad del equipo, y menor cantidad de defectos
- El trabajo iterativo deja una experiencia en el equipo que permite ir ajustando y mejorando las planificaciones, logrando menores desvíos en la duración total del proyecto.

Desventajas

- Al no tener todos los requisitos definidos desde el principio, puede que surjan problemas a la hora de establecer la arquitectura.
- Requiere de un cliente involucrado durante todo el curso del proyecto. Hay clientes que simplemente no estarán dispuestos a invertir el tiempo necesario.
- Infunde responsabilidad en el equipo de desarrollo al trabajar directamente con el cliente, requiriendo de profesionales sobre el promedio.
- El no tener en cuenta algunos requisitos, puede hacer que determinados cambios obliguen a cambios profundos en el software, lo cual puede suponer un incremento de costes y tiempo.

4.2.4 Modelo Incremental

Ventajas

- El modelo proporciona todas las ventajas del modelo en cascada realimentado, reduciendo sus desventajas sólo al ámbito de cada incremento.
- Se genera software operativo de forma rápida y en etapas tempranas del ciclo de vida del software.
- Resulta más sencillo acomodar cambios al acotar el tamaño de los incrementos.
- Al ir desarrollando parte de las funcionalidades, es más fácil determinar si los requisitos planeados para los niveles subsiguientes son correctos.
- Reduciendo el tiempo de desarrollo de un sistema (en este caso en incremento del sistema) decrecen las probabilidades que esos requisitos de usuarios puedan cambiar durante el desarrollo.
- Gestión de riesgos simplificada al tratarse de pequeños incrementos, ya que el riesgo de construir un sistema pequeño es menor que el de uno grande.
- Se reduce el tiempo de desarrollo inicial, ya que se implementa la funcionalidad parcial.
- Entrega temprana de partes operativas del software a los usuarios, e incrementos con cierta frecuencia.
- El usuario se involucra más y reduce su incertidumbre.
- Si un error importante es introducido, el incremento previo puede ser usado.
- Los errores de desarrollo introducidos en un incremento, pueden ser arreglados antes del comienzo del próximo incremento

Desventajas

- Difícil de evaluar el coste total.

- Los errores en los requisitos se detectan tarde.
- Requiere de mucha planificación, tanto administrativa como técnica.
- Requiere de metas claras para conocer el estado del proyecto.
- Requiere una experiencia importante para definir los incrementos y distribuir en ellos las tareas de forma proporcionada.
- Cada fase de un incremento es rígido.
- Pueden surgir problemas referidos a la arquitectura del sistema porque no todos los requisitos se han reunido, ya que se supone que todos ellos se han definido al inicio.

4.2.5 Modelo en Espiral

Ventajas

- Puede comenzarse el proyecto con un alto grado de incertidumbre.
- Es posible tener en cuenta mejoras y nuevos requerimientos sin romper con el modelo, ya que el ciclo de vida no es rígido ni estático.
- Se produce software en etapas tempranas del ciclo de vida.
- El análisis de riesgos se hace de forma explícita y clara.
- Agregar un análisis de riesgos para determinar si el proyecto será viable.
- Bajo riesgo de retraso en caso de detección de errores, ya que se puede solucionar en la próxima rama del espiral.
- Incorpora objetivos de calidad.
- Integra el desarrollo con el mantenimiento.
- Como el software evoluciona, a medida que progresa el proceso, el desarrollador y el cliente comprenden y reaccionan mejor ante riesgos en cada uno de los niveles evolutivos.
- El cliente se encuentra involucrado en el proceso del desarrollo del sistema.
- Provee la utilización y creación de prototipos.

Desventajas

- Es un modelo que genera mucho trabajo adicional.
- Dificultad para evaluar los riesgos. Al ser el análisis de riesgos una de las tareas principales exige un alto nivel de experiencia y cierta habilidad en los analistas de riesgos (es bastante difícil).
- Coste económico y temporal alto. Consume muchos recursos.
- Coste temporal que supone cada vuelta del espiral.
- Necesidad de la presencia o la comunicación continua con el cliente.
- Las etapas y sus E/S no están claramente definidas.

4.2.6 Modelo Ágil

Ventajas

- Capacidad de respuesta a cambios a lo largo del desarrollo. Los cambios se perciben como una oportunidad para mejorar el sistema e incrementar la satisfacción del cliente.
- Entrega continúa y en plazos breves de software incremental y funcional lo que permite al cliente verificar in situ el desarrollo del proyecto.
- Los riesgos y dificultades se reparten a lo largo del desarrollo del producto.
- Comunicación directa entre equipo de desarrollo y cliente, lo que lleva a una reducción de malentendidos y a evitar el exceso de documentación.
- Se elimina el trabajo innecesario que no aporta valor al negocio.
- El cliente puede comenzar el proyecto con requisitos de alto nivel, quizás no del todo completos, de manera que se vayan refinando en sucesivas iteraciones. Sólo es necesario conocer con más detalle los requisitos de las primeras iteraciones, los que más valor aportan.
- Feedback al final de cada iteración con el cliente para ir alineándose con las expectativas de este.
- El cliente como máximo puede perder los recursos dedicados a una iteración, no los de todo el proyecto.
- Permite conocer el progreso real del proyecto desde las primeras iteraciones y extrapolar si su finalización es viable en la fecha prevista. El cliente puede decidir repriorizar los requisitos del proyecto, añadir nuevos equipos, cancelarlo, etc.
- Permite mitigar desde el inicio los riesgos del proyecto. Desde la primera iteración el equipo tiene que gestionar los problemas que pueden aparecer en una entrega del proyecto. Al hacer patentes estos riesgos, es posible iniciar su mitigación de manera anticipada.
- Permite gestionar la complejidad del proyecto. En una iteración sólo se trabaja en los requisitos que aportan más valor en ese momento. Se puede dividir la complejidad para que cada parte sea resuelta en diferentes iteraciones.
- Dado que cada iteración debe dar como resultado requisitos terminados, se minimiza el número de errores que se producen en el desarrollo y se aumentar la calidad.

Desventajas

- La comunicación entre los desarrolladores y los clientes durante el desarrollo del proyecto debe ser activa y continua.
- La disponibilidad del cliente debe ser alta durante todo el proyecto dado que participa de manera continua.
- La relación con el cliente ha de estar basada en los principios de colaboración y ganar/ganar más que tratarse de una relación contractual en la cual cada parte únicamente defiende su beneficio a corto plazo.
- Cada iteración ha de aportar un valor al cliente, entregar unos resultados cerrados que sean susceptibles de ser utilizados por él.
- En ocasiones potencia más la acción que la planificación, lo cual puede llevar al caos.

4.3 Comparando las Metodologías Ágiles

En este apartado se hará una comparación entre metodologías ágiles teniendo en cuenta una serie de aspectos o puntos de referencia.

A la hora de comparar las metodologías podríamos seguir varios enfoques:

1. Tabla o framework que evalúe el nivel de cumplimiento de los principios ágiles de cada metodología.
2. Tabla resumen, donde en base a distintas características, se exponga para cada metodología el cómo cumplen esa característica.
3. CheckList con distintas características cruzada con las metodologías indicando que metodología tiene cada característica.

El primer enfoque simplemente valdría para saber cómo de ágil es cada metodología, lo cual queda fuera del alcance de este documento, pero se deja en referencias el informe [Evaluation10] el cual propone una forma de evaluar la agilidad de las distintas metodologías.

Con el segundo enfoque se puede llegar a una visión más subjetiva sobre las diferencias entre metodologías. Y con el tercero se obtienen unos datos más objetivos, pero menos completos.

Así que se ha optado por seguir las dos últimas opciones, creando tanto la tabla resumen como la CheckList.

4.3.1 Tabla Resumen de Metodologías

Lo primero es identificar qué características se incluirán en esta tabla. Y estas son:

1. **Tiempo para cada iteración recomendado** → Se indicará el tiempo recomendado para cada iteración.
2. **Tamaño del equipo** → Número de miembros recomendados para cada tipo de metodología.
3. **Comunicación en el equipo** → Cómo es la comunicación dentro del equipo
4. Tamaño del Proyecto
5. **Involucración del cliente** → Cómo estará involucrado el cliente y cuanto de importante es.
6. **Documentación en el proyecto** → Indicará la necesidad y cantidad de documentación requerida para el tipo de metodología analizada.
7. **Habilidades Especiales** → Se refiere a características o herramientas propias de cada metodología.
8. **Ventajas**
9. **Desventajas**

	ASD	AUP	Crystal	DSDM
Tiempo para cada iteración recomendado	4 a 8 semanas	Primeras iteraciones más tiempo que las demás.	Dependiendo del método de la familia.	80% solución en el 20% del tiempo.
Tamaño del equipo	Equipos pequeños 5 a 9 miembros	Todos los tamaños	Dependiendo del método de la familia	2 a 6 personas. Pero puede haber varios equipos trabajando en distintos módulos.
Comunicación en el equipo	Informal. Cara a cara.	Por medio de comunicación concisa.	Informal. Cara a cara. Periodos de no interrupción. Osmótica: Misma sala.	Basado en documentación. Capacidad del equipo de tomar decisiones importantes sin esperar aprobación de alto nivel.
Tamaño del Proyecto	Proyectos pequeños	Todo tipo de proyectos	Todo tipo de proyectos. Dependiendo del método de la familia	Todo tipo de proyectos. Pero en proyectos grandes deberá ser particionado.
Involucración del cliente	En cada entrega el cliente dará su aportación	Participa activa del cliente en las 4 fases, aunque no necesario día a día.	En cada entrega el cliente dará su aportación. No tiene porque estar accesible constantemente, pero si, al menos, semanalmente.	En cada entrega el cliente dará su aportación (entregas frecuentes)
Documentación en el proyecto	Sólo documentación básica	Documentación de alto nivel, concisa y formación de vez en cuando.	Sólo documentación básica	Abundante
Habilidades Especiales	Ciclo de Aprendizaje	Procesos Definidos	Familia compuesta por varios tipos que se adapta a todo tipo de proyectos y tamaño de equipos.	Prototipado
Ventajas	- Gracias a la colaboración y el aprendizaje, se mejoran mucho los procesos y se solucionan errores. - Muy tolerante al cambio. - Gestión del riesgo.	- El personal sabe lo que esta haciendo: no obliga a conocer detalles (alto nivel) - Simplicidad: apuntes concisos. - Agilidad: procesos simplificados del RUP. - Centrarse en actividades de alto valor: esenciales para El desarrollo. - No fuerza a usar ninguna técnica o herramienta.	Se ajusta a la criticidad o tipo de proyecto y al tamaño del equipo.	- Permite alcanzar nivel 2 CMMI. - Uso de prototipos para obtener requisitos. - Todos los cambios durante el desarrollo son reversibles - Los requisitos están especificados a un alto nivel - El testing está Integrado a través del ciclo de vida. - Desarrollo rápido.
Desventajas	- Errores y cambios que no son detectados con anterioridad afectan la calidad del producto y su costo total.	- El AUP es un producto muy pesado (simplificación de RUP), en relación a otros ágiles. - Como es un proceso simplificado, muchos desarrolladores eligen trabajar con RUP, por tener a disposición mas detalles en el proceso.	Coordinación difícil de equipos grandes.	- Documentación abundante. - Los equipos de DSDM deben tener el poder de tomar decisiones.

Figura 43 – Tabla Resumen Metodologías (ASD, AUP, Crystal, DSDM)

	XP	FDD	LSD
Tiempo para cada iteración recomendado	1 a 6 semanas.	2 días a 2 semanas.	Iteraciones cortas.
Tamaño del equipo	Menos de 20 personas.	Admite muchos miembros.	Todos los tamaños
Comunicación en el equipo	Informal. Cara a cara. Reuniones diarias de pie. Entre artecfactos y personas(código autocomentado, pruebas unitarias)	Basado en documentación.	Por medio de comunicación concisa.
Tamaño del Proyecto	Proyectos pequeños.	Proyectos complejos y críticos.	Todo tipo de proyectos
Involucración del cliente	Cliente involucrado. Comunicación fluída. El cliente decide prioridades y debe estar siempre disponible.	En cada entrega el cliente dará su aportación.	En cada entrega el cliente dará su aportación (entregas frecuentes)
Documentación en el proyecto	Sólo documentación básica. Código autodocumentado.	Existe documentación. La mínima necesaria para que alguien nuevo entienda el sistema.	Sólo documentar lo extrictamente necesario.
Habilidades Especiales	TDD, User Stories, Refactoring, Pair Programming	Diagramas UML.	Muda, Mura y Muri
Ventajas	- El cliente es parte del equipo. - Feedback constante. - Mejores prácticas bien definidas. - Simplicidad. - Potencia la mejora continua. - Da prioridad a la acción. - Propiedad colectiva del código	- Monitoreo constante del avance del proyecto. - Enfasis en la calidad. - Ofrece soluciones a situaciones de riesgo (cambios en requisitos, perdidas de tiempo….)	- Mejora la calidad. - Aumenta la productividad. - Elimina lo que no produzca valor para el cliente.
Desventajas	- Documentación pobre. - Puede llevar a poca disciplina. - La reflexión puede quedar demasiado relegada frente a la acción. - Presencia obligatoria e importante del cliente. - Hacer las cosas simples requiere experiencia. - Requiere coraje para asumir ciertas prácticas.	- Propiedad individual del código. - No cubre todo el ciclo de vida. Necesita de otras metodologías que la cumplimenten.	- Requiere equipo experimentado y responsable en su propio trabajo. - Requiere constantes esfuerzos de revisión y análisis de procesos.

Figura 44 - Tabla Resumen Metodologías (XP, FDD, LSD)

	Kanban	Scrum	Scrumban
Tiempo para cada iteración recomendado	Flujo constante. Se liberan entregables en base a eventos o necesidades	2 a 4 semanas.	Flujo constante. Se liberan entregables en base a eventos o necesidades
Tamaño del equipo	Sin prescripción.	Todos los tamaños (Scrum de Scrums)	No hay ScrumMaster. Equipos pequeños.
Comunicación en el equipo	Informal. Cara a cara. A traves de tablero Kanban. Cada equipo define sus procesos.	Informal. Reuniones diarias de pie. Reuniones de retrospectiva.	Informal. Reuniones diarias de pie. A traves de tablero Kanban. Reuniones de retrospectiva.
Tamaño del Proyecto	Proyectos de corta-media duración.	Todo tipo de proyectos.	Proyectos de corta-media duración.
Involucración del cliente	Sería conveniente una participación activa del cliente.	Cliente fuertemente involucrado a través del rol de ProductOwner.	Cliente involucrado. Comunicación fluída.
Documentación en el proyecto	Evita la documentación. Cada equipo define sus procesos.	Sólo documentación básica.	Sólo documentación básica.
Habilidades Especiales	Tablero Kanban.	Sprint, Product y Sprint BackLog, Scrum Board, Scrum Master, Planning Poker.	Lo mejor de Scrum y de Kanban
Ventajas	- Facilita el que todo el mundo sepa que hacer, quien hace que y como va el proyecto. - Limita el número de tareas a hacer al mismo tiempo. Control del flujo de trabajo. - Fácil de aplicar. - Flexible ante cambios.	- Equipo autoorganizado. - El equipo sabe lo que tiene que hacer cada día. - El cliente sabe lo que se le entrega en cada sprint. - Flexible a cambios. - Desarrolladores tienen autonomía. - Minimiza el trabajo de gestión. - Minimiza el sindrome del estudiante.	Todas las de Kanban, más todo lo que aportan las reuniones de Scrum.
Desventajas	- Se deben definir las fases del ciclo de trabajo. - No define roles, ni fases, ni tampoco profundiza en el tablero Kanban. - Algunos lo ven más como una técnica que como metodología.	- Puede producir stress al sentirse el equipo en un continuo sprint. - Requiere un equipo formado, motivado y con cierta experiencia. - Necesidad de involucración del cliente. - Problemas en equipos distribuídos geográficamente.	- Algunos lo ven como Scrum sólo que usando un tablero Kanban. - No hay una clara definición de roles y fases.

Figura 45 - Tabla Resumen Metodologías (Kanban, Scrum, Scrumban)

Como se puede observar en la tabla, no hay una metodología ideal, cada una tiene sus pros y sus contras, está preparada para un tipo de equipo u otro, es más favorable a unos u otros proyectos… en definitiva, este punto nos permite ver que el tener una metodología por defecto para todos los proyectos que se ejecuten, no es una buena idea, ya que hay metodologías específicas para cada caso.

Sin duda de entre todas la más liviana es Kanban, llegando incluso a poder pensar que es una técnica más que una metodología por la libertad que ofrece, en el otro extremo se tiene a AUP, que aun siendo una simplificación de RUP sigue siendo bastante pesada, y a DSDM que incluso permite alcanzar un nivel 2 CMMI.

También comentar que Scrumban se puede ver que no es más que una combinación de Kanban y Scrum, por lo que también podría ser vista como una simple modificación de los procesos de Scrum.

Por otro lado, las existosas XP y Scrum, pese a ser las dos metodologías ágiles con más seguidores, se puede ver que también presentan números desventajas, y que hay en muchos casos en que no son adecuadas.

4.3.2 CheckList de Metodologías

Para hacer esta CheckList se parte del trabajo ya realizado por Iacovelli [Lacovelli08]. El objetivo del estudio realizado por Iacovelli fue clasificar las metodologías a través de cuatro puntos de vista, cada uno representando un aspecto de las metodologías, y cada punto de vista a su vez se caracteriza por un conjunto de atributos.

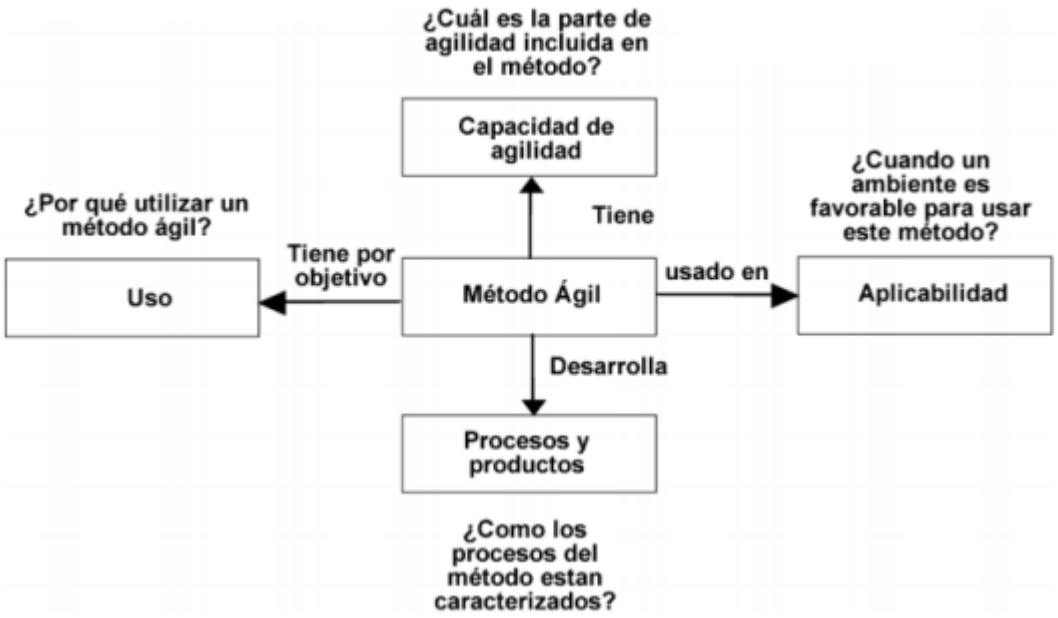


Figura 46 – Cuatro puntos de vista identificados por Iacovelli.

USO

Refleja el por qué utilizar metodologías ágiles.

Los atributos de esta vista tratan de evaluar todos los beneficios que el equipo de desarrollo y el cliente obtienen utilizando este tipo de metodologías: incremento de la productividad, calidad y satisfacción.

Los atributos de este punto de vista son:

- Adaptarse a los entornos turbulentos.
- Satisfacción del usuario final.
- Favorable al offshoring (outsourcing internacional).
- Aumento de la productividad.
- Respeto de un nivel de calidad.
- Respeto de las fechas de entrega.
- Cumplimiento de los requisitos.

CAPACIDAD DE AGILIDAD

Representa cuál es la parte ágil de la metodología.

Los atributos de esta vista representan todos los aspectos del concepto de agilidad y su evaluación refleja que aspectos están incluidos en una metodología.

Apuntar que el atributo colaboración se refiere a la relación que se establece entre los clientes y el equipo de desarrollo.

Los atributos de este punto de vista son:

- Indicadores pueden cambiar
- Colaboración.
- Requisitos funcionales pueden cambiar.
- Recursos humanos pueden cambiar.
- Integración de los cambios.
- Intercambio de conocimientos (bajo, alto).
- De peso ligero.
- Requisito no funcional puede cambiar.
- Centrado en las personas.
- Reactividad (al comienzo del proyecto, cada hito, cada iteración).
- Política de refactoring
- Iteraciones cortas
- Política de pruebas
- Plan de trabajo se puede cambiar.

APLICABILIDAD

El objetivo de esta vista es mostrar el impacto de los aspectos ambientales en el método.

Representa cuando el entorno es favorable para la aplicación de metodologías ágiles. Este aspecto se describe por los siguientes atributos, cada uno correspondiente a una característica del entorno.

- Grado de interacción entre los miembros del equipo (baja, alta).
- Grado de interacción con el cliente (baja, alta).
- Grado de interacción con los usuarios finales (baja, alta).
- Grado de innovación (baja, alta).
- Complejidad del proyecto (baja, alta).
- Riesgos del proyecto (baja, alta).
- Tamaño del proyecto (pequeño, grande).
- Organización del equipo (auto-organización, organización jerárquica).
- Tamaño del equipo (pequeño, grande).

PROCESOS Y PRODUCTOS

La vista de los procesos y productos representa cómo están caracterizados los procesos de la metodología y cuáles son los productos de sus actividades.

Los procesos se componen de dos dimensiones. La primera representa el nivel de abstracción de sus directrices y reglas. La segunda dimensión son las actividades de desarrollo de software cubiertas por las metodologías ágiles.

Por otro lado se listan los posibles productos de las actividades.

En el artículo de Iacovelli se muestran estas 3 dimensiones, como 3 atributos cada uno con un enumerado con posibles valores para cada uno. En este trabajo se ha decidido dividir, tal cual se hace en, esas tres dimensiones en cada uno de los valores de su enumerado para así dejar la tabla con un conjunto de atributos que sólo admiten 2 posibles valores, la razón se explicará en el capítulo 5, pero como adelanto, servirá para hacer nuestra elección de la metodología.

Por tanto los atributos de los procesos y los productos quedan:

Nivel de abstracción de las normas y directrices:

- Gestión de proyectos
- Descripción de procesos
- Normas y orientaciones concretas sobre las actividades y productos

Actividades cubiertas por la metodología:

- Puesta en marcha del proyecto
- Definición de requisitos
- Modelado
- Código
- Pruebas unitarias
- Pruebas de integración
- Prueba del sistema
- Prueba de aceptación
- Control de calidad
- Uso del sistema

Productos de las actividades de la metodología:

- Diseño del modelo
- Código fuente comentado
- Ejecutable
- Pruebas unitarias
- Pruebas de integración
- Pruebas de sistema
- Pruebas de aceptación
- Informes de calidad
- Documentación de usuario

A partir de estos cuatro puntos de vista, y sus atributos, se han generado las tablas con los valores para cada metodología. Estos valores se basan en los del artículo de Iacovelli, aunque algunos valores han sido modificados, y se amplía el número de metodologías.

Aclaración sobre los valores de las tablas:

- V = Verdadero, F = Falso
- A = Alto, B = Bajo
- IP = Inicio del Proyecto, H = Hito, I = Iteración
- G = Grande, P = Pequeño
- J = Jerárquico, AU = Auto-Organizado

			ASD	AUP	Crystal	DSDM	XP	FDD	LSD	Kanban	Scrum	Scrumban
USO	¿Por qué utilizar una metodología ágil?	Respeto de las fechas de entrega	V	V	V	V	F	V	F	F	V	F
		Cumplimiento de los requisitos	V	V	V	V	V	V	V	V	V	V
		Respeto de un nivel de calidad	V	V	F	F	F	V	V	F	F	F
		Satisfacción del usuario	F	F	F	V	V	F	V	F	V	V
		Adaptación a entornos turbulentos	V	F	F	V	V	F	F	V	V	V
		Favorable al off shoring	V	V	V	V	F	F	V	V	F	V
		Aumento de la productividad	F	F	F	V	V	F	V	V	V	V

Figura 47 – Valoración del "Uso" de las metodologías

			ASD	AUP	Crystal	DSDM	XP	FDD	LSD	Kanban	Scrum	Scrumban
CAPACIDAD DE AGILIDAD	¿Cúal es la parte de agilidad incluída en la metodología?	Iteraciones cortas	F	F	V	V	V	V	V	F	V	F
		Colaboración	V	V	V	V	V	F	V	V	V	V
		Centrado en las personas	V	V	V	V	V	F	V	V	V	V
		Política de refactoring	F	V	F	V	V	F	V	F	F	F
		Política de pruebas	V	V	V	V	V	V	V	F	V	V
		Integración de los cambios	V	V	F	V	V	F	V	V	V	V
		De peso ligero	F	F	F	F	V	V	V	V	V	V
		Requisitos funcionales pueden cambiar	V	V	F	V	V	F	V	V	V	V
		Requisito no funcional puede cambiar	V	F	F	F	F	F	F	V	F	V
		Plan de trabajo se puede cambiar	F	F	F	F	V	F	V	V	F	V
		Recursos humanos pueden cambiar	V	F	V	F	V	F	F	V	F	V
		Se pueden cambiar indicadores	F	V	F	F	V	F	V	F	F	F
		Reactividad	H	I	H	I	I	IP	I	H	I	H
		Intercambio de conocimientos	A	B	A	B	A	B	A	A	A	A

Figura 48 – Valoración "Capacidad de Agilidad" de las metodologías

			ASD	AUP	Crystal	DSDM	XP	FDD	LSD	Kanban	Scrum	Scrumban
APLICABILIDAD	¿Cúando un ambiente es favorable para usar este método?	Tamaño del proyecto	P	G/P	G/P	G/P	P	G	G/P	P	G/P	G/P
		Complejidad del proyecto	A	A	A	A	B	A	A	B	A	A
		Riesgos del proyecto	A	A	A	A	B	A	A	B	A	A
		Tamaño del equipo	P	G/P	G/P	G/P	P	G	G/P	P	P	P
		Grado de interacción con el cliente	A	B	B	A	A	B	A	A	A	A
		Grado de interacción con los usuarios finales	B	B	B	A	B	B	B	B	A	B
		Grado de interacción entre los miembros del equipo	A	B	A	A	A	B	A	B	A	A
		Grado de innovación	A	A	B	A	A	B	A	B	A	A
		Organización del equipo	J	J	AU	J	AU	J	AU	AU	AU	AU

Figura 49 – Valoración de la "Aplicabilidad" de las metodologías

			ASD	AUP	Crystal	DSDM	XP	FDD	LSD	Kanban	Scrum	Scrumban
PROCESOS Y PRODUCTOS	¿Cómo están caracterizados los procesos de la metodología y cuáles son los productos de sus actividades?	Nivel de abstracción de las normas y directrices										
		Gestión de proyectos	V	V	V	V	F	V	F	F	V	F
		Descripción de procesos	V	V	V	V	V	V	F	F	V	V
		Normas y orientaciones concretas sobre las actividades y productos	F	V	F	F	V	V	F	F	V	V
		Actividades cubiertas por la metodología										
		Puesta en marcha del proyecto	V	V	F	V	V	V	F	F	V	F
		Definición de requisitos	V	V	F	V	V	V	V	F	V	F
		Modelado	V	V	V	V	V	V	F	F	V	F
		Código	V	V	V	V	V	V	V	V	V	V
		Pruebas unitarias	V	V	V	V	V	V	V	V	V	V
		Pruebas de integración	V	V	V	V	V	V	V	V	V	V
		Prueba del sistema	V	V	V	V	V	V	V	V	V	V
		Prueba de aceptación	V	V	V	V	V	V	V	V	V	V
		Control de calidad	V	V	F	F	F	V	V	F	F	F
		Uso del sistema	F	V	F	V	F	F	F	F	F	F
		Productos de las actividades de la metodología										
		Diseño del modelo	F	V	F	V	F	V	F	F	V	F
		Código fuente comentado	V	V	V	V	V	V	V	F	V	V
		Ejecutable	V	V	V	V	V	V	V	V	V	V
		Pruebas unitarias	V	V	V	V	V	V	V	V	V	V
		Pruebas de integración	V	V	V	V	V	V	V	V	V	V
		Pruebas de sistema	V	V	V	V	V	V	V	V	V	V
		Pruebas de aceptación	V	V	V	V	V	V	V	V	V	V
		Informes de calidad	V	V	F	F	F	V	V	F	F	F
		Documentación de usuario	F	V	F	V	F	V	V	F	V	F

Figura 50 – Valoración de los "Procesos y Productos" de las metodologías

De los valores establecidos en esta CheckList se pueden sacar algunas conclusiones, como son que todas buscan mejorar el cumplimiento de requisitos, dando por bienvenidos en muchos casos los cambios en los requisitos funcionales, no así a los cambios en los requisitos no funcionales, que para la mayoría de las metodologías sigue siendo un problema.

Y si se le da especial importancia al cumplimiento de requisitos, no es así al respecto de las fechas de entrega, donde algunas metodologías como XP, LSD, Kanban o Scrumban flojean.

Como era de esperar, una buena colaboración con el cliente es importante para la mayoría de ellas, excepto para FDD donde la participación del cliente no es tan activa como en el resto. Y también de esperar era que las pruebas fuesen algo básico para todas ellas, y aunque no es contemplado por Kanban, aun así en procesos y productos se ha establecido el valor de las pruebas a verdadero en todos los campos ya que el tener una buena política de pruebas se considera que es algo inseparable del modelo de ciclo de vida ágil.

Finalmente un aspecto que puede llamar la atención, es que cuando se piensa en ágil se suele tener la idea de equipos auto-organizados, cuando en esta checkList se puede ver que casi la mitad de las metodologías exigen un modelo jerárquico.

Capítulo 5. Elección / Propuesta

5.1 Consideraciones Previas

A la hora de encarar un proyecto, como gestores se tendrá que pensar qué modelo de ciclo de vida usar, y una vez elegido este, pensar o discernir sobre que metodología es la más adecuada para el proyecto.

Para tomar estas decisiones se deberá identificar unos aspectos clave en cada proyecto, unas características genéricas que los diferencien a alto nivel para así poder clasificarlos. Los criterios a considerar serían la complejidad del problema, la criticidad del proyecto, el tiempo que disponemos para hacer la entrega final, si el usuario o cliente desea entregas parciales, la disponibilidad del cliente, la comunicación que existe entre el equipo de desarrollo y el usuario, el nivel del equipo, la certeza (o incertidumbre) que se tiene de que los requerimientos datos por el usuario son correctos y completos…

Estos criterios se han agrupado en los siguientes grupos.

5.1.1 Proyecto

El proyecto es el eje central de la actividad, y sobre el caerán la mayoría de los aspectos a considerar, así se tendrá:

- Complejidad técnica → Se valorará la dificultad técnica de la tarea encomendada.
- Criticidad → Se valorará como de importante es entregar el proyecto libre de fallos y perfectamente testeado.
- Tiempo → Se valorará como de factible es entregar el proyecto en tiempo y si la holgura es suficientemente ancha como para cubrir imprevistos.
- Captura inicial de requisitos → Se valorará las posibilidades que hay de hacer una captura completa de requisitos al inicio del proyecto.
- Modificación de requisitos → Aquí se analizará cómo de posible y de frecuente es el que el cliente realice cambios sobre los requisitos a lo largo del proyecto.
- Calidad → La importancia que se debe dar al proyecto a la calidad de cara a un futuro mantenimiento (un buen diseño, código documentado, documentación apropiada…)

5.1.2 Equipo

Uno de los conceptos fundamentales a tener en cuenta, y así es dicho en la mayoría de los libros y cursos de gestión de proyectos, es que el factor humano, el equipo, es básico para que el proyecto llegue a buen puerto. Así en este apartado se tendrá:

- Nivel → Con nivel se quiere decir la experiencia y conocimientos que posee el equipo, ya sea técnicamente o del negocio del problema.
- Tamaño → El número de miembros del equipo afecta a cómo se debe gestionar.

- Dedicación → Cuánto tiempo puede dedicar cada miembro alproyecto. Ya sea a tiempo completo o parcial.
- Situación geográfica → El que el equipo esté distribuido o esté en una misma oficina.

5.1.3 Cliente

El cliente será una parte crucial en el proyecto.

- Disponibilidad → Cómo de disponible y de interesado está el cliente en tener un contacto frecuente con el equipo del proyecto, ya sea a través de su jefe de proyecto o de sus miembros.
- Accesibilidad → Cómo de accesible es, y que nivel de disponibilidad va a tener, nuestro cliente a todos los miembros del equipo (desarrolladores incluidos).

5.2 Modelo a Usar

Antes de leer este documento, podría pensar que unos modelos de ciclo de vida predominan sobre otros, que son más útiles y que su aplicación continuada traerá mayores beneficios a nuestra organización, pero tras analizar cada uno de ellos y exponer sus ventajas e inconvenientes, se puede ver que ninguno es perfecto y que cada uno se adapta a una situación o entorno determinado.

En base al análisis que haya realizado de su proyecto en el punto 5.1, en este punto se ayudará al lector a seleccionar aquel modelo que mejor le convenga. Para ello primero se dará una explicación de para qué tipo de proyectos es adecuado cada modelo, para a continuación presentar un árbol de decisión donde en función de alguno de los parámetros expuestos en el punto anterior podamos llegar hasta nuestro modelo ideal.

5.2.1 Modelo en Cascada

Este modelo es adecuado cuando se disponen de todos los requisitos desde el principio, ya sea porque es un producto no novedoso, o porque es un proyecto con funcionalidades conocidas o porque aun siendo un proyecto complejo sus requisitos son fácilmente capturarles y entendibles. También es común usar este modelo en productos maduros donde los requisitos es difícil que cambien o en equipos débiles.

Con estas condiciones se da el entorno ideal para este modelo, donde en base a unos requisitos estables, podamos hacer unas estimaciones correctas y unos diseños adecuados que se mantengan a lo largo del tiempo.

Además, al estar todo bien definido, de forma línea, y ser tan fácilmente entendible, este modelo podrá ser aplicado por casi cualquier equipo, tenga la experiencia que tenga, y no necesitará del cliente más que en la fase de toma de requisitos, y en la entrega.

5.2.2 Modelo en V

Es un modelo que suele funcionar bien para proyectos pequeños donde los requisitos son entendidos fácilmente

Podemos utilizar este modelo de ciclo de vida en aplicaciones, que si bien son simples, necesitan una confiabilidad muy alta. Un ejemplo claro en el que no se puede permitir el lujo de cometer errores es una aplicación de facturación, en la que si bien los procedimientos vistos individualmente son de codificación e interpretación sencilla, la aplicación en su conjunto puede tener matices complicados.

5.2.3 Modelo Iterativo

Se suele utilizar en proyectos en los que los requerimientos no están claros de parte del usuario, por lo que se hace necesaria la creación de distintos prototipos para presentarlos y conseguir la conformidad del cliente.

Se puede adoptar el modelo mencionado en aplicaciones medianas a grandes, en las que el usuario o cliente final no necesita todas las funcionalidades desde el principio del proyecto y al que no le importa estar involucrado de forma activa en el proyecto. Quizás una empresa que debe migrar sus aplicaciones hacia otra arquitectura, desea hacerlo paulatinamente, es un candidato ideal para este tipo de modelo de ciclo de vida.

5.2.4 Modelo Incremental

El desarrollo incremental es útil sobre todo cuando nos encontramos ante un proyecto con unos requisitos conocidos pero donde el personal necesario para una implementación completa no está disponible. Los primeros incrementos se pueden implementar con menos gente. Si el producto esencial es bien recibido, se agrega (si se requiere) más personal para implementar el incremento siguiente. Además, los incrementos se pueden planear para manejar los riesgos técnicos. Por ejemplo, un sistema grande podría requerir la disponibilidad de un hardware nuevo que está en desarrollo y cuya fecha de entrega es incierta. Sería posible planear los primeros incrementos de forma que se evite el uso de este hardware, lo que permitiría la entrega de funcionalidad parcial a los usuarios finales sin retrasos desordenados.

Este modelo de ciclo de vida no está pensado para cierto tipo de aplicaciones, sino que está orientado a cierto tipo de usuario o cliente. Se puede utilizar este modelo de ciclo de vida para casi cualquier proyecto, pero será verdaderamente útil cuando el usuario necesite entregas rápidas, aunque sean parciales, de un producto operativo.

El modelo Incremental no es recomendable para casos de sistemas de tiempo real, de alto nivel de seguridad, de procesamiento distribuido, y/o de alto índice de riesgos. Difícil de aplicar a los sistemas transaccionales que tienden a ser integrados y a operar como un todo.

Tener en cuenta que para este modelo será necesario un importante labor de planificación y por lo tanto se necesitará un equipo experimentado capaz de repartir correctamente el trabajo entre incrementos.

5.2.5 Modelo en Espiral

Si en el modelo anterior, el incremental, se parte de que no hay incertidumbre en los requisitos iniciales, en este, en cambio, se es consciente de que se comienza con un alto grado de incertidumbre. En el incremental se supone que conocemos el problema y lo dividimos. Este modelo gestiona la incertidumbre.

De forma directa, se podría decir que el modelo en espiral es adecuado para proyectos largos, caros, complicados y con incertidumbre en los requisitos. Es decir, está pensado para proyectos largos y complejos de misión crítica, como puede ser, por ejemplo, la creación de un sistema operativo. Para proyectos pequeños no es muy adecuado porque la carga adicional de trabajo que supone no compensa en un proyecto de poca entidad.

Este modelo requiere de un equipo con la suficiente experiencia como para detectar y evaluar riesgos.

Además agregar que este modelo no llega a ser una alternativa al resto de modelos, sino que se podría situar por encima de ellos como un marco de trabajo, ya que es compatible con el resto de modelos desde el momento en que, en función de los riesgos que se vayan a asumir, en la fase de desarrollo se debe elegir otro modelo para lo que se vaya a desarrollar en ese ciclo.

5.2.6 Modelo Ágil

Este enfoque está mostrando su efectividad en proyectos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad.

Los métodos ágiles fueron pensados especialmente para equipos de desarrollo pequeños, con plazos reducidos, requisitos volátiles y nuevas tecnologías.

Así podríamos resumir que es adecuado cuando el equipo sea pequeño y esté formado mayoritariamente por gente con talento y experiencia (a la cual se quiere mantener contenta), cuando el cliente final esté involucrado y no imponga barreras de comunicación (y se busque su máxima satisfacción), cuando los requisitos sean altamente cambiantes, y no se esté ante un proyecto crítico (ya que no hay gestión de riesgos) ni tampoco sea demasiado grande.

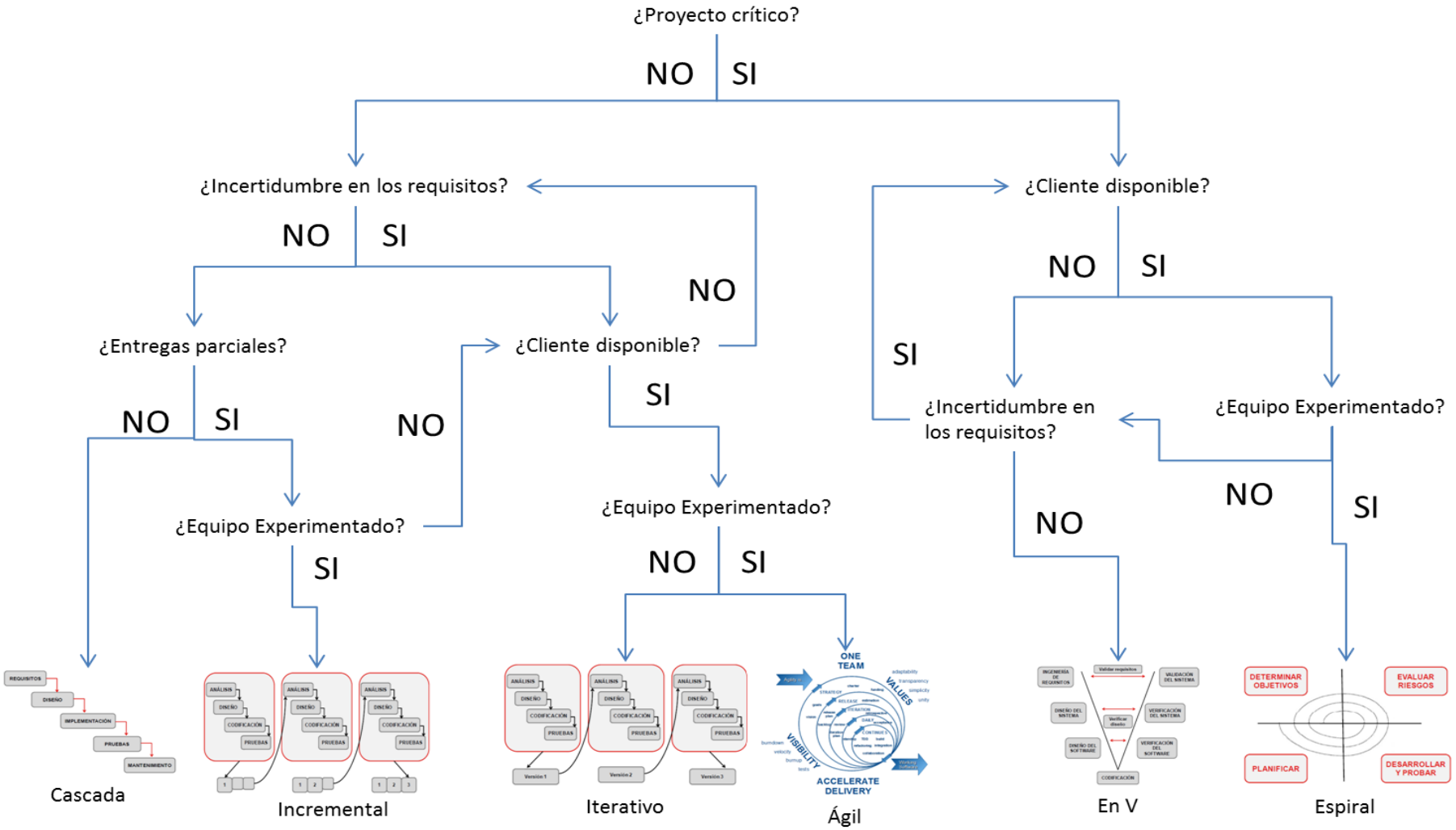


Figura 51 – Árbol de decisión para selección de modelo

5.3 Metodología a Usar

En este estudio, como ya se ha comentado varias veces, sólo nos vamos a centrar en el caso de que el modelo de ciclo de vida adecuado para tu proyecto sea el ágil, en cuyo caso, en este apartado se te ayudará a elegir una de las metodologías ágiles entre las diez analizadas en este documento.

Para tomar esta decisión nos podríamos valer de la información que tenemos recopilada en el punto 4.3.1, pero puede que esta no llegue a ser lo suficientemente objetiva, en cuyo caso nos valdremos otra vez de los datos recabados en la checklist del punto 4.3.2. Con esta información podremos saber si nuestro proyecto se enmarca en una u otra metodología. Para ello nos basamos en otro trabajo [PFG12].

En este trabajo, el cual también usa de base el estudio de lacovelli, se plantean una serie de cuestionarios a rellenar como jefes de proyecto. Estos cuestionarios se basan en dar un valor a cada una de las variables que lacovelli especificó en su artículo. Así se ha modificado ligeramente estos formularios para un mejor entendimiento de estos.

	USO	**Respuesta**
1	Respeto de las fechas de entrega	
2	Cumplimiento de los requisitos	
3	Respeto de un nivel de calidad	
4	Satisfacción del usuario	
4	Adaptación a entornos turbulentos	
5	Favorable al off shoring	
7	Aumento de la productividad	

Figura 52 – Formulario "Uso" de metodologías

	CAPACIDAD DE AGILIDAD	**Respuesta**
1	Iteraciones cortas	
2	Colaboración	
3	Centrado en las personas	
4	Política de refactoring	
4	Política de pruebas	
5	Integración de los cambios	
7	De peso ligero	
8	Requisitos funcionales pueden cambiar	
9	Requisito no funcional puede cambiar	
10	Plan de trabajo se puede cambiar	
11	Recursos humanos pueden cambiar	
12	Se pueden cambiar indicadores	
13	Reactividad (al comienzo del proyecto, cada hito, cada iteración)	
14	Intercambio de conocimientos (Bajo, Alto)	

Figura 53 - Formulario "Capacidad de Agilidad" de metodologías

	APLICABILIDAD	Respuesta
1	Tamaño del proyecto (Pequeño, Grande)	
2	Complejidad del proyecto (Bajo, Alto)	
3	Riesgos del proyecto (Bajo, Alto)	
4	Tamaño del equipo (Pequeño, Grande)	
4	Grado de interacción con el cliente (Bajo, Alto)	
5	Grado de interacción con los usuarios finales (Bajo, Alto)	
7	Grado de interacción entre los miembros del equipo (Bajo, Alto)	
8	Grado de innovación (Bajo, Alto)	
9	Organización del equipo (auto-organización, jerárquica)	

Figura 54 - Formulario "Aplicabilidad" de metodologías

	Nivel de abstracción de las normas y directrices	Respuesta
1	Gestión de proyectos	
2	Descripción de procesos	
3	Normas y orientaciones concretas sobre las actividades y productos	

Figura 55 - Formulario "Nivel de abstracción de las normas y directrices" de metodologías

	Actividades cubiertas por la metodología	Respuesta
1	Puesta en marcha del proyecto	
2	Definición de requisitos	
3	Modelado	
4	Código	
5	Pruebas unitarias	
6	Pruebas de integración	
7	Prueba del sistema	
8	Prueba de aceptación	
9	Control de calidad	
10	Uso del sistema	

Figura 56 - Formulario "Actividades cubiertas por la metodología" de metodologías

	Productos de las actividades de la metodología	Respuesta
1	Diseño del modelo	
2	Código fuente comentado	
3	Ejecutable	
4	Pruebas unitarias	
5	Pruebas de integración	
6	Pruebas de sistema	
7	Pruebas de aceptación	
8	Informes de calidad	
9	Documentación de usuario	

Figura 57 - Formulario "Producto de las actividades de la metodología" de metodologías

Comparando los resultados introducidos en los formularios con la checkList del punto 4.3.2, se identificará la metodología que mejor se adapta a la forma de trabajo de la empresa y al proyecto. La metodología adecuada será la que mayor número de coincidencias tenga con el cuestionario anterior.

Adjunto a este documento se incluye un documento Excel creado para este trabajo que aporta esta misma funcionalidad, donde se muestra la tabla resumen del punto 4.3.1, la CheckList del punto 4.3.2, los formularios aquí presentados, y que una vez rellenos muestran sus resultados con la metodología supuestamente más recomendable.

5.3.1 Ejemplo Práctico

Imaginar que se está a punto de encarar un nuevo proyecto, que ya se tiene claro que el modelo de ciclo de vida que mejor se adapta al proyecto es el ágil, pero se está abrumado ante la cantidad de metodologías ágiles disponibles. En ese caso, rellenando los formularios aquí propuestos, se sugerirá una metodología, que será aquella que más coincidencias tenga con aquello que se busca.

Así que si en esos formularios, se rellanse lo siguiente…

	USO	**Respuesta**
1	Respeto de las fechas de entrega	V
2	Cumplimiento de los requisitos	V
3	Respeto de un nivel de calidad	V
4	Satisfacción del usuario	V
4	Adaptación a entornos turbulentos	V
5	Favorable al off shoring	F
7	Aumento de la productividad	F

	CAPACIDAD DE AGILIDAD	**Respuesta**
1	Iteraciones cortas	V
2	Colaboración	V
3	Centrado en las personas	V
4	Política de refactoring	V
4	Política de pruebas	V
5	Integración de los cambios	V
7	De peso ligero	F
8	Requisitos funcionales pueden cambiar	V
9	Requisito no funcional puede cambiar	F
10	Plan de trabajo se puede cambiar	F
11	Recursos humanos pueden cambiar	V
12	Se pueden cambiar indicadores	V
13	Reactividad (al comienzo del proyecto, cada hito, cada iteración)	I
14	Intercambio de conocimientos (Bajo, Alto)	A

	APLICABILIDAD	**Respuesta**
1	Tamaño del proyecto (Pequeño, Grande)	P
2	Complejidad del proyecto (Bajo, Alto)	B
3	Riesgos del proyecto (Bajo, Alto)	B
4	Tamaño del equipo (Pequeño, Grande)	P
4	Grado de interacción con el cliente (Bajo, Alto)	A
5	Grado de interacción con los usuarios finales (Bajo, Alto)	B
7	Grado de interacción entre los miembros del equipo (Bajo, Alto)	A
8	Grado de innovación (Bajo, Alto)	A
9	Organización del equipo (auto-organización, jerárquica)	AU

	Nivel de abstracción de las normas y directrices	**Respuesta**
1	Gestión de proyectos	V
2	Descripción de procesos	V
3	Normas y orientaciones concretas sobre las actividades y productos	V

	Actividades cubiertas por la metodología	**Respuesta**
1	Puesta en marcha del proyecto	V
2	Definición de requisitos	V
3	Modelado	V
4	Código	V
5	Pruebas unitarias	V
6	Pruebas de integración	V
7	Prueba del sistema	V
8	Prueba de aceptación	V
9	Control de calidad	V
10	Uso del sistema	F

	Productos de las actividades de la metodología	**Respuesta**
1	Diseño del modelo	V
2	Código fuente comentado	V
3	Ejecutable	V
4	Pruebas unitarias	V
5	Pruebas de integración	V
6	Pruebas de sistema	V
7	Pruebas de aceptación	V
8	Informes de calidad	V
9	Documentación de usuario	V

Como veis se han puesto valores en relación a las características que tiene o que se quiere ver en el equipo, el proyecto y el cliente.

Así, los resultados obtenidos serían:

			ASD	AUP	Crystal	DSDM	XP	FDD	LSD	Kanban	Scrum	Scrumban
USO	¿Por qué utilizar una metodología ágil?	Respeto de las fechas de entrega	1	1	1	1	0	1	0	0	1	0
		Cumplimiento de los requisitos	1	1	1	1	1	1	1	1	1	1
		Respeto de un nivel de calidad	1	1	0	0	0	1	1	0	0	0
		Satisfacción del usuario	0	0	0	1	1	0	1	0	1	1
		Adaptación a entornos turbulentos	1	0	0	1	1	0	0	1	1	1
		Favorable al off shoring	0	0	0	0	1	1	0	0	1	0
		Aumento de la productividad	1	1	1	0	0	1	0	0	0	0
			ASD	**AUP**	**Crystal**	**DSDM**	**XP**	**FDD**	**LSD**	**Kanban**	**Scrum**	**Scrumban**
CAPACIDAD DE AGILIDAD	¿Cúal es la parte de agilidad incluída en la metodología?	Iteraciones cortas	0	0	1	1	1	1	1	0	1	0
		Colaboración	1	1	1	1	1	0	1	1	1	1
		Centrado en las personas	1	1	1	1	1	0	1	1	1	1
		Política de refactoring	0	1	0	1	1	0	1	0	0	0
		Política de pruebas	1	1	1	1	1	1	1	0	1	1
		Integración de los cambios	1	1	0	1	1	0	1	1	1	1
		De peso ligero	1	1	1	1	0	0	0	0	0	0
		Requisitos funcionales pueden cambiar	1	1	0	1	1	0	1	1	1	1
		Requisito no funcional puede cambiar	0	1	1	1	1	1	1	0	1	0
		Plan de trabajo se puede cambiar	1	1	1	1	0	1	0	0	1	0
		Recursos humanos pueden cambiar	1	0	1	0	1	0	0	1	0	1
		Se pueden cambiar indicadores	0	1	0	0	1	0	1	0	0	0
		Reactividad	0	1	0	1	1	0	1	0	1	0
		Intercambio de conocimientos	1	0	1	0	1	0	1	1	1	1

			ASD	AUP	Crystal	DSDM	XP	FDD	LSD	Kanban	Scrum	Scrumban
APLICABILIDAD	¿Cúando un ambiente es favorable para usar este método?	Tamaño del proyecto	1	1	1	1	1	0	1	1	1	1
		Complejidad del proyecto	0	0	0	0	1	0	0	1	0	0
		Riesgos del proyecto	0	0	0	0	1	0	0	1	0	0
		Tamaño del equipo	1	1	1	1	1	0	1	1	1	1
		Grado de interacción con el cliente	1	0	0	1	1	0	1	1	1	1
		Grado de interacción con los usuarios finales	1	1	1	0	1	1	1	1	0	1
		Grado de interacción entre los miembros del equipo	1	0	1	1	1	0	1	0	1	1
		Grado de innovación	1	1	0	1	1	0	1	0	1	1
		Organización del equipo	0	0	1	0	1	0	1	1	1	1

			ASD	AUP	Crystal	DSDM	XP	FDD	LSD	Kanban	Scrum	Scrumban
PROCESOS Y PRODUCTOS	¿Cómo están caracterizados los procesos de la metodología y cuáles son los productos de sus actividades?	Nivel de abstracción de las normas y directrices										
		Gestión de proyectos	1	1	1	1	0	1	0	0	1	0
		Descripción de procesos	1	1	1	1	1	1	0	0	1	1
		Normas y orientaciones concretas sobre las actividades y productos	0	1	0	0	1	1	0	0	1	1
		Actividades cubiertas por la metodología										
		Puesta en marcha del proyecto	1	1	0	1	1	1	0	0	1	0
		Definición de requisitos	1	1	0	1	1	1	1	0	1	0
		Modelado	1	1	1	1	1	1	0	0	1	0
		Código	1	1	1	1	1	1	1	1	1	1
		Pruebas unitarias	1	1	1	1	1	1	1	1	1	1
		Pruebas de integración	1	1	1	1	1	1	1	1	1	1
		Prueba del sistema	1	1	1	1	1	1	1	1	1	1
		Prueba de aceptación	1	1	1	1	1	1	1	1	1	1
		Control de calidad	1	1	0	0	0	1	1	0	0	0
		Uso del sistema	1	0	1	0	1	1	1	1	1	1
		Productos de las actividades de la metodología										
		Diseño del modelo	0	1	0	1	0	1	0	0	1	0
		Código fuente comentado	1	1	1	1	1	1	1	0	1	1
		Ejecutable	1	1	1	1	1	1	1	1	1	1
		Pruebas unitarias	1	1	1	1	1	1	1	1	1	1
		Pruebas de integración	1	1	1	1	1	1	1	1	1	1
		Pruebas de sistema	1	1	1	1	1	1	1	1	1	1
		Pruebas de aceptación	1	1	1	1	1	1	1	1	1	1
		Informes de calidad	1	1	0	0	0	1	1	0	0	0
		Documentación de usuario	0	1	0	1	0	1	1	0	1	0
		TOTAL	39	40	32	38	42	32	37	26	41	31

Como se puede ver en las imágenes anteriores, en la tabla se pone un uno en cada campo en el que coincide el valor que le pusimos en el punto 4.3.2 y el valor que se ha establecido en nuestro formulario.

Así, la metodología que más puntos obtiene es XP con 42 puntos, seguida de Scrum con 41 y AUP con 40.

La interpretación de estos resultados no tiene que ser que la única metodología válida para este proyecto es XP, sino que debe ser que "Probablemente la más adecuada sea XP" pero cada uno deberá analizar si realmente es así, y en caso de no serlo, pasar a la siguiente opción con más puntos, en este caso Scrum, y así sucesivamente.

Es decir, este método te da una forma de ordenar las metodologías ágiles disponibles en orden de "Probablemente más adecuada" a "Probablemente menos adecuada", para que a la hora de elegir tu metodología tengas un punto de partida o sepas por cual empezar.

5.4 Resumen

Entonces, ¿Cómo deberíamos leer o utilizar este documento una vez que hayamos entendido que realmente es importante para nuestro proyecto el tener claro qué modelo y metodología usar? Tras una primera lectura del documento, a continuación deberíamos:

1. Releer el punto 3.2, 4.2 y seguir el árbol de decisión del 5.2 para decidir el modelo a usar.
2. En base a esa elección, si se ha elegido modelo ágil, releer del punto 3.3 y 4.3 y usar la Excel que acompaña este documento tal cual se explica en el punto 5.3. Con esto se tendrá una propuesta de metodología ágil.
3. Verificar que esta metodología es adecuada para nuestro proyecto y entorno. En caso de que no sea así, pasar a la siguiente metodología con más puntos.
4. Una vez todo en marcha, siempre tener en cuenta que habrá procesos, que se tendrán que adaptar a nuestro entorno.

Capítulo 6. Conclusiones

Al inicio de la investigación, la idea era encontrar el camino perfecto y claro para que a partir de unas condiciones, como son tipología y prioridades del proyecto a realizar, cliente, equipo disponible, tiempo, organización… poder llegar a elegir el Modelo de ciclo de vida y la metodología que mejor se adapten a nosotros.

Tras documentar en este trabajo sobre el porqué del fracaso de los proyectos software, explicar 6 modelos de ciclo de vida, diez metodologías ágiles y otras tantas técnicas ágiles, ver sus ventajas y desventajas, preparar listas de comprobación, formularios y árboles de decisión, se ha llegado a varias conclusiones:

- Los modelos de ciclo de vida y las metodologías presentados, muchas veces son complementarios entre sí. En muchos casos, los paradigmas pueden y deben combinarse de forma que puedan utilizarse las ventajas de cada uno en un único proyecto.
- No se debe establecer un modelo y una metodología por defecto para todos los proyectos. Cada proyecto es un mundo que depende además de otros importantes factores como son el equipo que lo va a desarrollar o el cliente que lo solicita. Aun así sí que hay forma de aproximar cual es aquel modelo y metodología que mejor se adapta a nuestras necesidades.
- La elección de un modelo y una metodología a seguir es un paso crítico en cualquier proyecto software, y así lo deben entender tanto los jefes de proyecto, como el equipo que lo forman, como la organización para la que trabajan.

Este documento puede ayudar a aportar luz en el inicio del camino de un proyecto software. La información que aquí se recoge, todo en un mismo sitio, es una fuente importante de valor para alguien que se encuentre en la encrucijada de elegir modelo y metodología para su proyecto, cuando seguramente en muchos casos ni si quiera se tenía clara la diferencia entre modelo de ciclo de vida y metodología. Aquí se ayuda a dilucidar muchos conceptos, obtener información importante sobre modelos, metodologías ágiles y sus técnicas, y se da un método a seguir para poder realizar una primera aproximación a tu modelo y metodología adecuada.

El árbol de decisión propuesto y los formularios en combinación con los CheckList darán una aproximación sobre el modelo y metodología a usar, aun así esto no es una ciencia exacta, y por lo tanto los resultados obtenidos a través de estas técnicas siempre deben ser tomados con cautela, teniendo en cuenta que cada organización deberá muchas veces ajustar las metodologías a su propio carácter y forma de hacer las cosas.

Capítulo 7. Referencias Bibliográficas

7.1 Libros y Artículos

[Zavala04] J. Jesús María Zavala Ruiz. "¿Por Qué Fracasan los Proyectos de Software?; Un Enfoque Organizacional". Congreso Nacional de Software Libre. 2004

[CSTIC09] Cristina Rivas Vila, Juan Antonio López, Jose Barato. ¿Por qué fracasan los proyectos Software?". CSTIC. 2009

[Standish94] Standish Group. "The Chaos Report". The Standish Group. 1994

[Standish09] Standish Group. "CHAOS Summary 2009". The Standish Group. 2009

[ComputingNow10] J. Laurenz Eveleens, Chris Verhoef. "The Rise and Fall of the Chaos Report Figures". ComputingNow. 2010

[INTECO09] Laboratorio Nacional de Calidad del Software del INTECO. "Ingeniería del Software: Metodologías y Ciclos de Vida". INTECO. 2009

[Grey11] J.Grey, Prof. H.M. Huisman. "The development of a hybrid agile project management methodology". North-West University. 2011

[KLR08] KLR. "Selecting the Right Project Management Methodology". KLR. 2008

[Palacio07] Juan Palacio. "Flexibilidad con Scrum. Principios de diseño e implantación de campos de Scrum". lulu. 2007

[Varas00] Marcela Varas C. "Gestión de Proyectos de Desarrollo de Software". Universidad de Concepción. 2000

[Tesis08] Pilar Rodríguez González. "Estudio de la aplicación de metodologías ágiles para la evolución de productos software". Universidad Politécnica de Madrid. 2008

[PFG12] María José Pérez Pérez. "Guía Comparativa de Metodologías Ágiles". Universidad de Valladolid. 2012

[Kniberg07] Henrik Kniberg. "Scrum y XP desde las trincheras". InfoQ. 2007. 978-1-4303-2264-1

[Kniberg10] Henrik Kniberg, Mattias Skarin. "Kanban and Scrum - making the most of both". InfoQ. 2010. 978-0-557-13832-6

[Evaluation10] Karla Mendes Calo, Elsa Estevez, Pablo Fillottrani. "A Quantitative Framework for the Evaluation of Agile Methodologies". JCS&T Vol. 10 No. 2. 2010

[Lacovelli08] Adrian Iacovelli, Carine Souveyet. “Framework for Agile Methods Classi¯cation”. Universite Paris. 2008

7.2 Referencias en Internet

[Buonamico13] Damián Buonamico. “Historia de las Metodologías Ágiles en Contexto”. http://www.caminoagil.com/2013/03/historia-de-las-metodologias-agiles-en.html. 2013

[Pmoinformatica13] pmoinformatica. “Una breve historia de las metodologías ágiles”. http://www.pmoinformatica.com/2013/06/una-breve-historia-de-las-metodologias.html. 2013

[Romo10] Ascari Romo. “El Fracaso en los Proyectos de Software”. http://mundobyte-x.blogspot.com.es/2010/03/ultima-revision-17-de-marzo-2010.html. 2010

[Cohn12] Mike Cohn. “Agile Succeeds Three Times More Often Than Waterfall”. http://www.mountaingoatsoftware.com/blog/agile-succeeds-three-times-more-often-than-waterfall. 2012

[JRamon12] Jose Ramón. “Buenas noticias para la gestión de proyectos”. http://informatica.blogs.uoc.edu/2012/10/08/buenas-noticias-para-la-gestion-de-proyectos/. 2012

[JGarzasChaos10] Javier Garzás. “Informe CHAOS: Visión y críticas sobre el éxito de los proyectos software”. http://www.javiergarzas.com/2010/08/proyectos-software-informe-chaos.html. 2010

[Krigsman07] Michael Krigsman. “New IT project failure metrics: is Standish wrong?”. http://www.zdnet.com/blog/projectfailures/new-it-project-failure-metrics-is-standish-wrong/513. 2007

[RubbYie] Rubby Casallas, Andrés Yie. “Ingeniería del Software: Ciclos de Vida y Metodologías”.http://sistemas.uniandes.edu.co/~isis2603/dokuwiki/lib/exe/fetch.php?media=principal:isis2603-modelosciclosdevida.pdf.

[Rivera11] Iván Rivera. “¿Hay futuro para la administración de proyectos en la metodología ágil?”. http://ivanrivera-pmp.blogspot.com.es/2011/08/hay-futuro-para-la-administracion-de.html. 2011

[Kumar 06] Anil Kumar Natogi. “Agility in Project Management. A recipe for turbulent times”. http://www.pmiglc.org/PD/Uploads/Presentations/Anil%20Kumar%20Natogi%20Agility%20in%20Project%20Management%20v2.pdf. 2006

[ExecutiveBrief08] ExecutiveBrief. “Which Life Cycle Is Best for Your Project?”. http://www.projectsmart.co.uk/which-life-cycle-is-best-for-your-project.html. 2008

[Martha 08] Martha. “Ciclos de Vida - Clasificación”. http://ciclosdevida1.blogspot.com.es/2008/09/ciclos-de-vida-clasificacin.html. 2008

[Oriente13] Universidad de Oriente - Venezuela. “Metodologías para el desarrollo de software”.

http://wiki.monagas.udo.edu.ve/index.php/Metodolog%C3%ADas_para_el_desarrollo_de_software#Clasificaci.C3.B3n_de_las_Metodolog.C3.ADas_seg.C3.BAn_el_modelo_de_proceso. 2013

[Madpitbull 13] Madpitbull_99. "Metodologías para el desarrollo de software". http://www.slideshare.net/madpitbull_99/modelos-de-ciclos-de-vida. 2013

[JGarzasIterativoIncremental10] Javier Garzás. "Veterano ciclo de vida iterativo e incremental". http://www.javiergarzas.com/2010/01/veterano-ciclo-de-vida-iterativo-incremental.html. 2010

[JGarzasAgil10] Javier Garzás. "Una metodología ágil no es siempre la mejor opción". http://www.javiergarzas.com/2010/11/contextualizar-metodologia-agil.html. 2010

[Jummp12] Jumpp. "Desarrollo de software. Ágil no es lo mismo que iterativo incremental". http://jummp.wordpress.com/2012/02/26/desarrollo-de-software-agil-no-es-lo-mismo-que-iterativo-incremental/. 2012

[Proyectos12] proyectosagiles.org. "Desarrollo iterativo e incremental". http://www.proyectosagiles.org/desarrollo-iterativo-incremental. 2012

[Sócola12] Daniel Sócola Escobar. "Metodologías Y Ciclos De Vida". http://www.slideshare.net/vdaniel20/metodologas-y-ciclos-de-vida. 2012

[Metricas12] Pmoinformatica.com. "5 métricas de desempeño para proyectos de desarrollo ágil y Scrum". http://www.pmoinformatica.com/2012/08/5-metricas-para-proyectos-de-desarrollo.html. 2012

[Rubio11] Juan Carlos Rubio Pineda. "Seminario de metodologías ágiles, bloque I". http://es.slideshare.net/jcrubio/curso-metod-agiles-b1. 2011

[Social10] Social Media Group. "Pensamiento agil, un estilo de vida!". http://es.slideshare.net/jlema/pensamiento-agil-un-estilo-de-vida. 2010

[Alejandroslide12] Alejandroslide. "Metodologías Agiles de Dirección de Proyectos". http://es.slideshare.net/Alejandroslide/alejandro-gabayort-conferencia-19072011. 2012

[Marble08] marble. "Metodologías ágiles de gestión de proyectos (Scrum, DSDM, Extreme Programming – XP…)". http://www.marblestation.com/?p=661. 2008

[UFPE12] Unioeste/UFPE - Informatics. "Overview Agile Methods". http://www.slideshare.net/ifsse3/overview-agile-methods. 2012

[Dimitri11] Dimitri Ponomareff. "Introducing Agile Scrum XP and Kanban". http://www.slideshare.net/dimka5/introducing-agile-scrum-xp-and-kanban. 2011

[Dimitri12] Dimitri Ponomareff. "Scrum vs Kanban". http://www.slideshare.net/dimka5/scrum-vs-scrumban-8728461. 2012

[Moniruzzaman12] A B M Moniruzzaman. "Comparative study on agile software development". http://www.slideshare.net/mzkhan2000/comparative-study-on-agile-software-development. 2012

[GarridoASD10] Juan Manuel Garrido. "Metodología ágil: ASD (Adaptive Software Development)". http://developerwiki.egafutura.com/glosario/metodologia-agil-asd-adaptive-software-development. 2010

[GarridoCrystal10] Juan Manuel Garrido. "Metodología ágil: Crystal". http://developerwiki.egafutura.com/glosario/metodologia-agil-crystal. 2010

[GarzasCrystal12] Javier Garzás. "Las metodologías Crystal. Otras metodologías ágiles que, quizás, te puedan encajar más que Scrum". http://www.javiergarzas.com/2012/09/metodologias-crystal.html. 2012

[Nancy07] Nancy Morales. "Crystal". http://seminariodeinformatica-1.blogspot.com.es/2007/10/crystal.html. 2007

[GarzasCockBurn11] Javier Garzás. "Una metodología para cada proyecto, o la escala de Cockburn". http://www.javiergarzas.com/2011/06/metodologa-por-proyecto.html. 2011

[GarridoDSDM10] Juan Manuel Garrido. "Metodología ágil: DSDM". http://developerwiki.egafutura.com/glosario/metodologia-agil-dsdm. 2010

[Niñoles12] Pablo Niñoles Aznar. "Scrumban". http://aesmppdf.blogspot.com.es/2012/03/scrumban.html. 2012

[Miñana12] Roberto Miñana. "Kanban para novatos". http://calidadysoftware.blogspot.com.es/2012/07/kanban-para-novatos.html. 2012

[Deseta08] Leonardo Deseta. "Kanban y Scrum". http://www.dosideas.com/noticias/metodologias/184-kanban-y-scrum.html. 2008

[GarzasKanban11] Javier Garzás . "Kanban y Scrum". http://www.javiergarzas.com/2011/11/kanban.html. 2011

[Salazar13] Antonio Fco. Martín Romero, Daniel I. Salazar Recio. "Agile unified process". http://osl2.uca.es/wikiCE/index.php/Agile_unified_process. 2013

[Dasari05] Dasari. Ravi Kumar. "Lean Software Development". http://www.projectperfect.com.au/downloads/Info/info_lean_development.pdf. 2005

[Pavon13] José Carlos Pavón Montañez, Pablo Pérez Luna. "Lean software development". http://osl2.uca.es/wikiCE/index.php/Agile_unified_process. 2013

[Vazquez12] José Vázquez Sánchez. "Aplicando el pensamiento Lean a la Gestión de Proyectos.". hhttp://www.gestiondeproyectosit.es/blogit/2012/03/lean-project-management/. 2012