5. PROBLEMÁTICA DE LOS PROYECTOS SOFTWARE

5.1. Introducción a los proyectos software

5.1.1. ¿Qué es un proyecto software?

Haciendo uso de la definición de proyecto de la guía del PMBOK [32], y adaptándola a un proyecto software, podríamos definirlo como:

"Un proyecto software es un esfuerzo temporal que se lleva a cabo para crear un producto software, servicio TI o resultado único."

¿Pero que es el software? Según la definición del IEEE, *"software es la suma total de los programas de ordenador, procedimientos, reglas, la documentación asociada y los datos que pertenecen a un sistema de cómputo", y "un producto de software es un producto diseñado para un usuario".*

El software puede dividirse en dos grandes categorías:

- **Software de aplicaciones**: se usan para proveer servicios a clientes y ejecutar negocios de forma más eficiente. El software de aplicaciones puede ser un sistema pequeño o uno grande integrado. Como ejemplos de este tipo de software estarían un sistema de cuentas, un sistema de planificación de recursos…
- **Software de sistemas**: el software de sistemas se usa para operar y mantener un sistema informático. Permite a los usuarios usar los recursos del ordenador directamente y a través de otro software. Algunos ejemplos de este tipo de software son los sistemas operativos, compiladores y otras utilidades del sistema.

5.1.2. Ingeniería del software

Los proyectos software tienen características específicas que los hacen diferentes de otros proyectos de ingeniería.

La Ingeniería del Software es la rama de la ingeniería que crea y mantiene las aplicaciones de software usando tecnologías y prácticas de las ciencias de la computación, manejo de proyectos, ingeniería, el ámbito de la aplicación, y otros campos.

¿Por qué el software es diferente a cualquier otro proceso de fabricación? Podríamos identificar los siguientes motivos:

- El software se desarrolla, no se fabrica en el sentido clásico de la palabra. Ambas actividades se dirigen a la construcción de un "producto", pero los métodos son diferentes. Los costes del software se encuentran en la ingeniería, esto implica que los proyectos no se pueden gestionar como si lo fueran de fabricación.
- La juventud de la ingeniería del software con respecto a otras ingenierías, la mayoría del software se construye a medida, en vez de ensamblar componentes previamente creados. Aunque ya se están dando importantes pasos en esta dirección, que facilitaría en gran medida el desarrollo de aplicaciones informáticas.
- 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.
- El software no se estropea, pero se deteriora. Durante su vida, el software sufre cambios (mantenimiento). Conforme se hacen los cambios, es bastante probable que se introduzcan nuevos defectos, lo que hace que el software se vaya deteriorando debido a estos cambios.

5.1.3. Proceso de desarrollo de un producto software

La formalización del proceso de desarrollo se define como un marco de referencia denominado ciclo de desarrollo del software o ciclo de vida del desarrollo del software.

La ISO, en su norma 12207 define al ciclo de vida del software como un marco de referencia que contiene las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto software, abarcando desde la definición hasta la finalización de su uso.

Las etapas principales a realizar en cualquier ciclo de vida son:

- Análisis: Construye un modelo a partir de la toma de los requisitos
- Diseño: A partir del modelo de análisis se deducen las estructuras de datos, la estructura en la que descompone el sistema y la interfaz de usuario.
- Codificación: Construye el sistema. La salida de esta fase es código ejecutable.
- Pruebas: Se comprueba que se cumplen criterios de corrección y calidad.
- Mantenimiento: En esta fase, que tiene lugar después de la entrega se asegura que el sistema siga funcionando y adaptándose a nuevos requisitos.

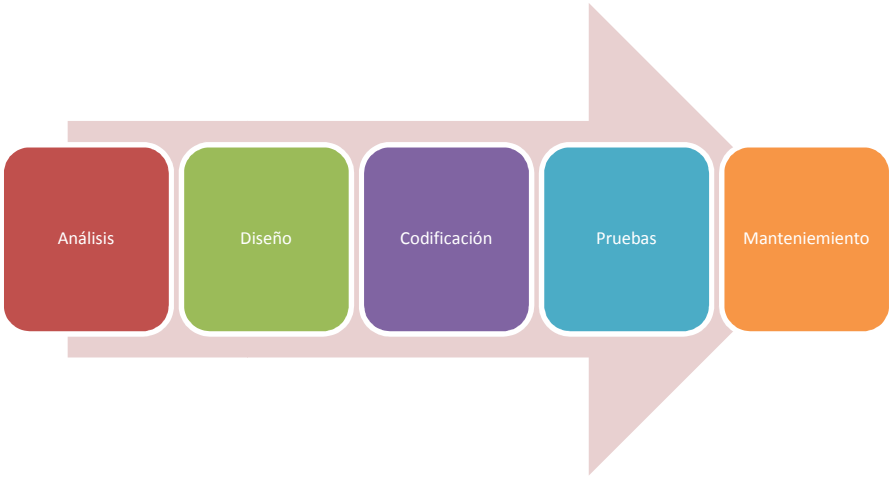

Figura 35 Etapas Ciclo de Vida Software

El ciclo de desarrollo del software es utilizado para estructurar las actividades que se llevan a cabo en el desarrollo de un producto software.

En cada una de las fases suelen participar diferentes recursos humanos que se encargan de cada una de las tareas, por regla general, en un proyecto software suelen participar los siguientes perfiles:

Jefes de Proyecto	•Su rol es el mismo que en cualquier tipo de proyecto
Consultor	•Participan principalmente en la fase de análisis y aportan conocimiento funcional en el ámbito de aplicación del proyecto.
Analistas	•Su participación principal es en la fase de análisis y diseño. •Sus tareas van desde la recolección de requisitos, hasta el diseño lógico de la solución, pasando por las pruebas del sistema.
Analistas programadores	•Pueden participar en la gran mayoría de las etapas del proyecto, aumentando su dedicación en la etapa de diseño (en especial el físico), y en la de codificación.
Programadores	•Su tarea principal es la codificación, y su dedicación mayoritaria en dicha fase. •Pueden participar también la fase de pruebas.

Tabla 15 RRHH Proyectos Software

A continuación se exponen las distintas aproximaciones de desarrollo de software, en función del tipo de ciclo de vida:

5.1.3.1. Aproximación convencional

Fue introducida por Winston Royce en la década de 1970, como una técnica rígida para mejorar la calidad y reducir los costos del desarrollo de software.

Conocido también como **modelo en cascada**, por su filosofía de completar cada paso con un alto grado de exactitud, antes de iniciar el siguiente.

Es el más utilizado por las organizaciones, se estima que el 90% de los sistemas han sido desarrollados con este modelo de ciclo de vida.

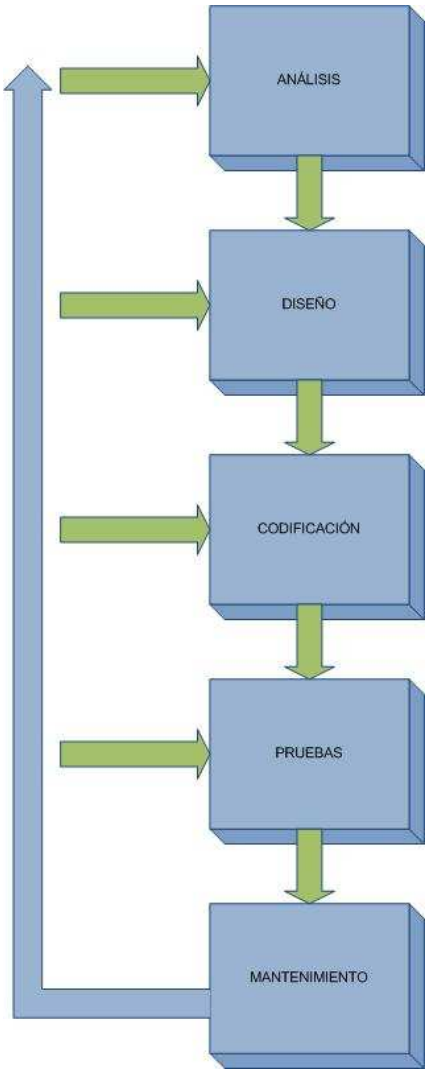

Figura 36 Modelo en cascada

Este modelo admite la posibilidad de realizar iteraciones, durante las modificaciones que se hacen en el mantenimiento se puede ver por ejemplo la necesidad de cambiar algo en el diseño, lo cual significa que se harán los cambios necesarios en la codificación y se tendrán que realizar de nuevo las pruebas, es decir, si se tiene que volver a una de las etapas anteriores al mantenimiento hay que recorrer de nuevo el resto de las etapas.

Después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente.

Este modelo trabaja en base a documentos, es decir, la entrada y la salida de cada fase es un tipo de documento específico. Idealmente, cada fase podría hacerla un equipo diferente gracias a la documentación generada entre las fases.

Los documentos son:

- **Análisis**: Toma como entrada una descripción en lenguaje natural de lo que quiere el cliente. Produce el S.R.D. (Software Requirements Document).
- **Diseño**: Su entrada es el S.R.D. Produce el S.D.D. (Software Design Document)
- **Codificación**: A partir del S.D.D. produce módulos. En esta fase se hacen también pruebas unitarias.
- **Pruebas**: A partir de los módulos probados se realiza la integración y pruebas de todo el sistema. El resultado de las pruebas es el producto final listo para entregar.

La aproximación tradicional tiene las siguientes ventajas:

- La planificación es sencilla.
- La calidad del producto resultante es alta.
- Permite trabajar con personal poco cualificado.

Y los siguientes inconvenientes:

- La necesidad de tener todos los requisitos al principio. Lo normal es que el cliente no tenga perfectamente definidas las especificaciones del sistema, o puede ser que surjan necesidades imprevistas.
- Si se han cometido errores en una fase, es difícil volver atrás.
- No se tiene el producto hasta el final, esto quiere decir que:
  - Si se comete un error en la fase de análisis no lo descubrimos hasta la entrega, con el consiguiente gasto inútil de recursos.
  - El cliente no verá resultados hasta el final, con lo que podría inquietarse.
- No se tienen indicadores fiables del progreso del trabajo.
- Es comparativamente más lento que los demás y el coste es mayor también.

En este modelo existen variantes, como son:

- **Ciclo de vida en cascada incremental:**

En esta tipología se va creando el sistema añadiendo pequeñas funcionalidades.

Cada uno de los pequeños incrementos es parecido a lo que ocurre dentro de la fase de mantenimiento.

La gran ventaja de este método es que no es necesario tener todos los requisitos en un principio, y el inconveniente principal es que los errores en la detección de requisitos se encuentran tarde.

- **Ciclo de vida iterativo:**

Es un modelo derivado del ciclo de vida en cascada. 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 en la fase de análisis.

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.

- **Ciclo de vida en V**

Propuesto por Alan Davis, tiene las mismas fases que el anterior pero se considera el nivel de abstracción de cada una. Una fase además de utilizarse como entrada para la siguiente, sirve para validar o verificar otras fases posteriores.

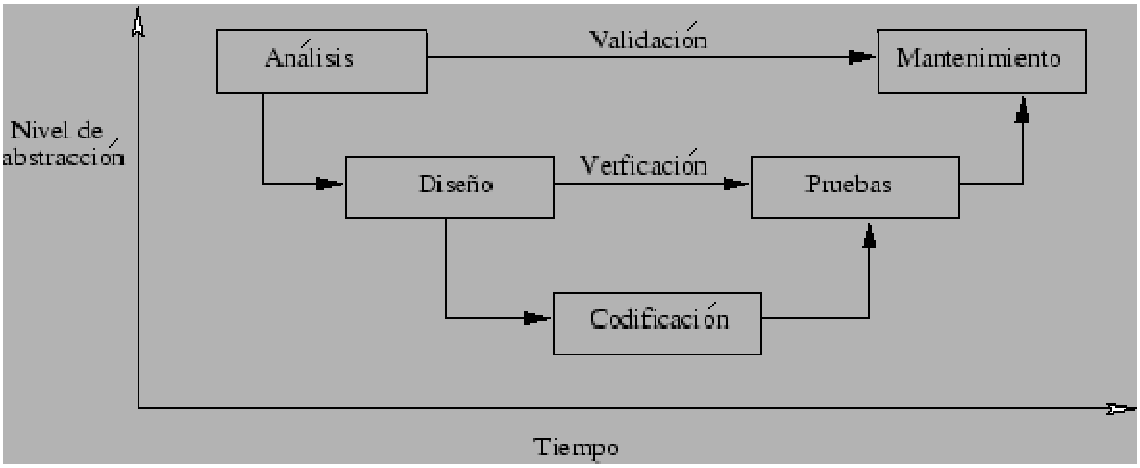

Figura 37 Ciclo de vida en V

- **Ciclo de vida tipo sashimi**

Según el modelo en cascada puro una fase solo puede empezar cuando ha terminado la anterior. En este caso, se permite un solapamiento entre fases. Por ejemplo, sin tener terminado del todo el diseño se comienza a implementar.

El nombre "sashimi" deriva del modo del estilo de presentación de rodajas de pescado crudo en Japón. Una ventaja de este modelo es que no necesita generar tanta documentación como el ciclo de vida en cascada puro debido a la continuidad del mismo personal entre fases.

Los problemas que plantean son:

- Es aún más difícil controlar el progreso del proyecto debido a que los finales de fase ya no son un punto de referencia claro.
- Al hacer cosas en paralelo si hay problemas de comunicación pueden surgir inconsistencias.

5.1.3.2. Aproximación prototipo

Es muy normal que en un proyecto software no se identifiquen los requisitos detallados de entrada, procesamiento o salida. En otros casos no se conoce la eficiencia de un algoritmo, o de la forma en que se ha de implantar un interface hombre-máquina.

En este tipo de situaciones, lo habitual es construir un prototipo, que idealmente pudiera como mecanismo para identificar los requisitos del software.

Esta aproximación consiste en realizar la fase de análisis en base a los siguientes factores:

- Un alto grado de iteración
- Un muy alto grado de participación del usuario
- Un uso extensivo de prototipos

Las ventajas de este modelo son:

- Los prototipos constituyen un medio de comunicación mejor que los modelos en papel (y la documentación escrita en general).
- Las iteraciones conducen el proceso de aprendizaje en la dirección correcta. Esta aproximación se orienta a mejorar la efectividad del proceso de desarrollo.

Y los principales inconvenientes:

- Los usuarios finales, ven el prototipo, sin considerar que no es la versión definitiva y por lo tanto no tiene en cuenta aspectos de calidad o de facilidad de mantenimiento.
- Muchas veces se siguen solicitando cambios sobre el prototipo, de forma que el avance es muy lento.

5.1.3.3. Aproximación en espiral

Este modelo fue propuesto originalmente por Boehm en 1988. Se basa en una serie de ciclos que se repiten, cada uno tiene las mismas fases y cuando termina da un producto ampliado con respecto al ciclo anterior.

En este sentido es similar al modelo incremental, la diferencia importante es que tiene en cuenta el concepto de riesgo. Un riesgo puede ser requisitos no comprendido, un mal diseño, errores en la implementación, etc.

En cada una de las iteraciones, se recomienda recopilar la siguiente lista de informaciones:

- **Objetivos**: Se hacen entrevistas a los clientes, se les hace rellenar cuestionarios, etc.
- **Alternativas**: Diferentes formas posibles para conseguir los objetivos. Se consideran desde dos puntos de vista:
  - Características del producto.
  - Formas de gestionar el proyecto.
- **Restricciones**:
  - Desde el punto de vista del producto: Interfaces de tal o cual manera, rendimiento, etc.
  - Desde el punto de vista organizativo: Coste, tiempo, personal, etc.
- **Riesgos**: Lista de riesgos identificados.
- **Resolución de riesgos**: La técnica más usada es la construcción de prototipos.
- **Resultados**: Son lo que realmente ha ocurrido después de la resolución de riesgos.
- **Planes**: Lo que se va a hacer en la siguiente fase.
- **Compromiso**: Decisiones de gestión sobre como continuar.

Al terminar una iteración se comprueba que lo que se ha hecho efectivamente cumple con los requisitos establecidos, también se verifica que funciona correctamente. El propio cliente evalúa el producto. No existe una diferenciación bien definida entre cuando termina el proyecto y cuando empieza la fase de mantenimiento. Cuando hay que hacer un cambio, este puede consistir en un nuevo ciclo.

Como ventajas de esta aproximación, podemos destacar:

- No requiere una definición completa de los requisitos para empezar a funcionar.
- Al entregar productos desde el final de la primera iteración es más fácil validar los requisitos.
- El riesgo en general es menor, porque si todo se hace mal, solo se ha perdido el tiempo y recursos invertidos en una iteración (las anteriores iteraciones están bien).
- El riesgo de retrasos es menor, ya que al identificar los problemas en etapas tempranas tenemos tiempo para subsanarlos.

Y tendríamos los siguientes inconvenientes:

- Es difícil evaluar los riesgos.
- Necesita de la participación continua por parte del cliente.
- Cuando se realiza una subcontratación es necesario producir previamente una especificación completa de lo que se necesita, y esto conlleva un tiempo.

5.1.3.4. Aproximación orientada a objetos

Los tipos de ciclos de vida descritos hasta ahora son relativos al análisis y diseño estructurados, pero los objetos tienen una particularidad, y es que están basados en componentes que se relacionan entre ellos a través de interfaces, o lo que es lo mismo, son mas modulares y por lo tanto el trabajo se puede dividir en un conjunto de mini-proyectos. Además, hoy en día existe una tendencia a reducir los riesgos, y en este sentido, el ciclo de vida en cascada no proporciona muchas facilidades. Debido a todo esto, el ciclo de vida típico en una metodología de diseño orientado a objetos es iterativo e incremental.

Dentro del ciclo de vida orientado a objetos, el más representativo es el modelo fuente.

Este modelo fue creado por Henderson-Sellers y Edwards en 1990. Es un tipo de ciclo de vida pensado para la orientación a objetos y posiblemente el más seguido.

Un proyecto según este modelo, se divide en las fases:

- Planificación del negocio
- Construcción: Es la fase más importante, dividiéndose a su vez en otras cinco actividades
  - Planificación
  - Investigación
  - Especificación
  - Implementación
  - Revisión
- Entrega

Siendo la primera, y la tercera fase independientes de la metodología.

Además de las tres fases, existen dos periodos:

- Crecimiento: Tiempo durante el cual se construye el sistema.
- Madurez: Es el periodo de mantenimiento del producto. Cada una de las mejoras se planifica igual que en el periodo anterior, es decir, con las fases de Planificación del negocio, Construcción y Entrega.

Cada clase puede tener un ciclo de vida sólo para ella debido a que cada una puede estar en una fase diferente en un momento cualquiera. La gran ventaja es que nos permite un desarrollo solapado e iterativo.

5.1.3.5. Aproximación desarrollo iterativo e incremental

En un ciclo de vida iterativo e incremental el proyecto se planifica en diversos bloques temporales llamados iteraciones.

Cada una de las iteraciones se puede considerar como pequeños proyectos, en todas las iteraciones se repite un proceso de trabajo similar para proporcionar un resultado completo sobre 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.

Uno de los aspectos más fundamentales para guiar el desarrollo iterativo e incremental, es la priorización de los objetivos y los requisitos en función del valor que aportan al cliente.

Este ciclo de vida es utilizado por la gran mayoría de las metodologías ágiles.

Presenta las siguientes ventajas:

- Gestiona las expectativas del cliente (requisitos desarrollados, velocidad de desarrollo, calidad) de manera regular, puede tomar decisiones en cada iteración. Esto es especialmente interesante cuando:
  - El cliente no sabe exactamente qué es lo que necesita, lo va conociendo conforme va viendo cuales son los resultados del proyecto.
  - El cliente necesita hacer cambios a corto plazo (nuevos requisitos o a cambios en los ya realizados).
  - El equipo necesita saber si lo que ha entendido es lo que el cliente espera.
- 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.
- El cliente puede obtener buenos resultados desde las primeras iteraciones.

- Es posible gestionar de una manera natural los cambios que pueden ir apareciendo durante el proyecto. La finalización de cada iteración es el lugar natural donde el cliente puede proporcionar sus comentarios tras examinar el resultado que se ha obtenido. Con esta información ya nos es posible planificar los cambios que harán falta para alinearse con las expectativas del cliente desde las primeras iteraciones, de manera que al finalizar el proyecto el cliente obtenga los objetivos esperados.
- El cliente como máximo puede perder los recursos dedicados a una iteración, no los de todo el proyecto.
- La finalización de cada iteración es el lugar natural donde el equipo puede decidir cómo mejorar su proceso de trabajo, en función de la experiencia obtenida. Con esta información disponible, es posible planificar los cambios que podrían ser necesarios para aumentar la productividad y la calidad desde las primeras iteraciones.
- 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 re-priorizar 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.
- En una iteración sólo se trabaja en los requisitos que aportan más valor en ese momento.
- Permite gestionar la complejidad del proyecto. 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.

Y los siguientes inconvenientes:

- Requiere una alta disponibilidad por parte del cliente durante todo el proyecto, dado que participa de manera continua:
  - El inicio de una iteración, el cliente ha de detallar (o haber detallado previamente) los requisitos que se van a desarrollar.
  - En la finalización de cada iteración, el cliente ha de revisar los requisitos desarrollados.
- 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 debe dar como resultado requisitos terminados, de manera que el resultado sea realmente útil para el cliente y no deje tareas pendientes para futuras iteraciones o para la finalización del proyecto.
- Cada iteración ha de aportar un valor al cliente, entregar unos resultados cerrados que sean susceptibles de ser utilizados por él.

- Se necesita disponer de técnicas y herramientas que permitan realizar cambios fácilmente en el producto, de manera que pueda crecer en cada iteración de manera incremental sin hacer un gran esfuerzo adicional, manteniendo su complejidad minimizada y su calidad.

5.2. Situación actual

Para describir la situación actual de los proyectos software, utilizaremos los resultados del informe Chaos publicados por The Standish Group en 2009 [25].

Este informe presenta un estudio sobre los resultados de proyectos en el sector de las tecnologías de la información (TI). Categoriza los proyectos en:

- **Exitosos** (Suceceeded): entregados en plazo, sin incrementos de presupuesto, y con las características requeridas.
- **Con problemas** (Challenged): entregados tarde, con sobre costes y/o con menos funcionalidades de las esperadas.
- **Fallidos** (Failed): Cancelados antes de terminarse, o entregados pero que no se ha llegado a utilizar su producto resultante.

A continuación, podemos mostrar la gráfica con los resultados:

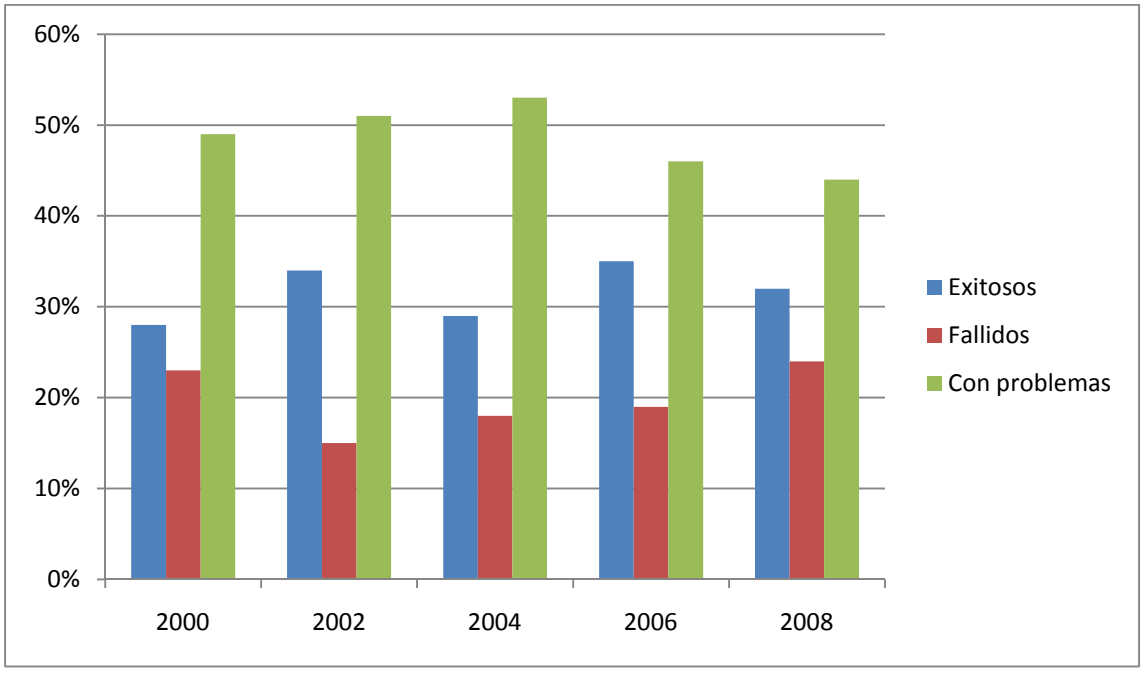

Figura 38 Gráfica Informe Chaos 2009

Cómo se puede apreciar, el porcentaje de proyectos que son fallidos, o tienen problemas es muy elevado (68% en 2009).

¿Cuál es el motivo de estos resultados en los proyectos software?

Una de las explicaciones, es que los proyectos software tienen características únicas que los hacen muy diferentes a los proyectos de otras disciplinas.

Según George Stepanek en su libro, "*Why Software Projects Fail*" [52], podríamos enumerar doce características, que hace diferente el desarrollo de software, respecto a otros tipos de desarrollos o fabricación.

1. **El software es complejo:** Un pequeño programa, fácilmente puede ejecutar decenas de miles de líneas de código, otros más importantes, como las últimas versiones de Microsoft Windows, decenas de millones.

   Además, la complejidad de un programa no depende sólo del número de instrucciones (sentencias) que se ejecutan, sino también de las interacciones entre dichas instrucciones, haciéndolo aún más complejo, ya que cada iteración puede ser diferente a las previas.

2. **El software es abstracto:** El software es el producto más abstracto que se puede crear en un proyecto.

   El software es una codificación de un conjunto amplio de comportamientos: Si esto ocurre, entonces esto otro debe suceder, y así sucesivamente.

   Podemos visualizar los comportamientos individuales, pero se tienen grandes dificultades para visualizar un gran número de comportamientos secuenciales y alternativos.

3. **Los requerimientos son incompletos:** El software es normalmente encargado para cumplir con las necesidades de usuarios y administradores, no de desarrolladores profesionales.

   Estas personas son expertos en sus propios roles, pero rara vez tienen tanta experiencia como desarrolladores profesionales en relación con la abstracción y la complejidad del software.

   Ellos entienden los procesos de negocio mucho mejor que los desarrolladores, pero incluso cuando alguien tiene una buena comprensión de los flujos principales de comportamiento del sistema que se desea implementar, no suelen tener en cuenta todos los flujos alternativos y las condiciones de error, y cómo diferentes grupos de requisitos se relacionan entre sí.

Para tener éxito, los usuarios y los desarrolladores deben trabajar juntos para precisar los requisitos del software.

Además, a través de las pruebas, los usuarios suelen ir añadiendo más funcionalidad al producto (según lo van visualizando).

4. **La tecnología cambia rápidamente:** Las nuevas versiones de los sistemas operativos salen cada pocos años, por ejemplo, hace unos veinte años estábamos trabajando con MS-DOS, y ahora prácticamente nadie lo recuerda.

   Hoy en día, cualquier software nuevo, es casi seguro que será construido con una estructura de aplicaciones empresariales o frameworks como Java de Sun Microsystems 2 Enterprise Edition (J2EE) o Microsoft NET. Estos frameworks en los que se basan gran número de aplicaciones, van evolucionando, proporcionando nuevas características o modificando las existentes.

   Resumiendo, las tecnologías de desarrollo de software cambian más rápido que otras tecnologías, como pueden ser las de la construcción.

5. **Las mejores prácticas no están suficientemente maduras:** La mayoría de las tecnologías de desarrollo de software están lo suficientemente maduras para tener un conjunto de buenas prácticas probadas.

   Muchas tecnologías aparecen y desaparecen en pocos años, en contraposición a otros procesos de desarrollo o fabricación fuera del mundo del software, por lo que no existen guías de buenas prácticas (o no están maduras).

6. **La tecnología es un dominio muy vasto:** En el desarrollo de software se utilizan muchas tecnologías, normalmente estas cambian en cada proyecto, y se utilizan varias a la vez, por lo que la complejidad aumenta y los programadores no llegan a tener nunca una experiencia amplia en un juego de tecnologías concreto.

   Cómo factor agravante, cada año aparecen nuevas tecnologías en el mercado, que aunque pueden tener funcionalidades interesantes, requieren aprendizaje.

7. **La experiencia en tecnología es incompleta:** Las tecnologías de desarrollo de software se quedan obsoletas en muy poco tiempo, apareciendo nuevas tecnologías.

   Esto implica que parte de los conocimientos y la experiencia adquirida con una tecnología particular pierda valor, y que los desarrolladores estén aprendiendo nuevas tecnologías a la vez que está desarrollando con ellas.

8. **El desarrollo de software implica investigación:** El proceso de desarrollo de software es un proceso de descubrimiento progresivo, buscando las características de los programas que satisfagan las necesidades de los clientes.

   Los desarrolladores deben combinar sus habilidades analíticas y creativas para averiguar lo que realmente quiere su cliente (incluso si la descripción del cliente es confusa e incompleta) e inventar maneras de combinar estos requisitos en un sistema que sea lógicamente consistente y fácil de usar.

9. **El trabajo repetitivo es automatizado:** El desarrollo de software se ha automatizado en un mayor grado que otras actividades basadas en proyectos.

   Todas las tareas repetitivas pueden ser automatizadas en el desarrollo de software, y esto es debido a que el software no existe en el mundo real. Reside en un entorno controlado de la computadora. Cada parte puede ser creado y controlado por medio de una amplia gama de herramientas de software.

10. **En realidad, la construcción es diseño:** El desarrollo de software es todavía un proceso de investigación. Según va avanzando el proyecto se van realizando diseños y rediseños para ir refinando el producto, hasta conseguir el objetivo deseado de forma incremental.

    La programación es algo más que escribir código. Cada paso requiere que el desarrollador analice una parte del problema y diseñe algún aspecto de la solución.

11. **Los cambios se consideran fáciles:** Cambios de última hora en los requerimientos son poco frecuentes en la construcción de carreteras, porque las consecuencias son graves. Si se descubre en el transcurso de un proyecto que los cimientos están en el lugar equivocado, entonces se necesita un esfuerzo considerable para desenterrar y reconstruir en otro lugar. Esto es obvio para los clientes y contratistas por igual.

    Esto es diferente en el desarrollo de software. Cualquier parte del software se puede cambiar en cualquier momento, con sólo volver a escribir la parte del código que corresponda. Los clientes piensan que los cambios en el software son fáciles de implementar y se piden con mucha frecuencia y en un gran número.

12. **El cambio es inevitable:** Ningún software es perfecto en su primera versión, sino que siempre se requieren cambios para que se adapte de forma óptima a las necesidades del cliente.

    De esta forma es importante trabajar con códigos consistentes que soporten cambios en la medida de lo posible, y evitar códigos “frágiles”.

Otro de los motivos, es que tradicionalmente se han gestionado los proyectos software con metodologías tradicionales, las cuales se ha podido comprobar que no son las idóneas para este tipo particular de proyectos en la mayoría de los casos.

Para encajar mejor en los proyectos software, han aparecido las metodologías ágiles.

Estas metodologías se basan en los principios del manifiesto ágil [53], que fue firmado en 2001 por Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas.

Los doce principios del manifiesto son:

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
9. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
10. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
11. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
12. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

5.3. Tendencias y líneas de estudio

La gestión de proyectos es un área de investigación que está en continuo desarrollo y crecimiento. Cada año aumenta el número de artículos que trata temas acerca de la gestión de proyectos.

Para mostrar las tendencias y las líneas de estudio recomendadas, nos basaremos en el artículo "*Uncovering the trends in Project management: Journal empjases over the last 10 years*" [18].

En este artículo se examinan las tendencias en la literatura de gestión de proyectos. Para realizar estudio utiliza varias fuentes:

- Siete reconocidos estudios previos sobre tendencias en la gestión de proyectos desde 1984 hasta 1999.
- Tendencias más actuales en la literatura de gestión de proyectos (1994 - 2003), obtenidas utilizando un análisis de palabras claves, haciendo uso del método computacional de análisis de de Corpus Linguistics.

Tras el estudio, se ha comprobado que la Gestión de la Comunicación, la Gestión de los Recursos Humanos, la Gestión de la Planificación, la Gestión de los costes y la Gestión de los riesgos se mantienen constantes en importancia, en cambio, Gestión del Alcance, Cierre de proyectos y/o fases y marketing han perdido interés. El estudio también sugiere, que Gestión de la Calidad y cuestiones interpersonales llegaron a alcanzar su punto máximo y ahora están decreciendo.

Por último, el estudio establece como las áreas que más están incrementando su importancia, la Evaluación y mejora de Proyectos (aprendizaje organizacional, gestión del rendimiento y evaluación y revisión de proyectos) y el alineamiento estratégico (caso de negocio, gestión de la financiación, alineamiento estratégico).

5.4. Casos de estudio

En este apartado se documentarán diferentes casos de estudio que han sido exitosos en la gestión de proyectos software haciendo uso de metodologías ágiles.

5.4.1. Scrum and CMMI – Systematic

En este caso de éxito comentaremos la introducción de Scrum en la empresa danesa Systematic [13].

Systematic es una compañía multinacional dedicada a desarrollar soluciones software de gran complejidad y criticidad, con sedes en Dinamarca, Finlandia, Estados Unidos e Inglaterra y más de 500 empleados.

Las soluciones que desarrolla Systematic son utilizadas en defensa, sanidad, industria y servicios.

Systematic tenía implantado el nivel 5 de CMMi desde el 2005.

Con la colaboración de Jeff Sutherland, en 2006 se realizó un piloto de Scrum en dos proyectos. Los resultados fueron excelentes, por lo que a finales de 2006, la mayoría de los proyectos de Systematic utilizaban la metodología scrum en lugar del desarrollo tradicional en cascada.

Los equipos que trabajaron con Scrum en el piloto doblaron la productividad respecto al resto de equipos, y además redujeron un 40% los errores. Y esto no fue todo, sino que una vez que se estableció Scrum en la compañía y se asentó y mejoró el proceso, llegaron hasta cuadriplicar la productividad que proyectos que utilizaban el desarrollo tradicional en cascada.

El éxito se basó en la gran capacidad de adaptabilidad y predictibilidad al utilizar juntos Scrum y CMMi juntos. La perspectiva de CMMi considera a Scrum como un proceso más que es monitorizado para mejorar su eficiencia y efectividad. A su vez Scrum demostró que agilizaba el desarrollo, acortaba el tiempo de solución de errores, y adelantaba las entregas.

5.4.2. DSDM – OCLC Online Computer Library Center, Inc.

En este caso, se documentará la introducción de DSDM en la organización norteamericana sin ánimo de lucro, OCLC Online Computer Library Center [2].

Creada en 1967, OCLC desarrolla software para su uso en bibliotecas, y sus usuarios son museos, bibliotecas y otras instituciones académicas. Más de 54.000 bibliotecas en 96 países utilizan los servicios de OCLC. Desde 1998 era poseedora de la ISO 9000.

En 1998, OCLC comenzó a tener problemas en la entrega del software debido a su incremento de clientes, y decidió buscar nuevas prácticas para el desarrollo del software.

Con la ayuda de la consultora inglesa TCC, OCLC incorporó DSDM (Dynamic Systems Development Methods) en su cultura de desarrollo de software en lugar de los métodos tradicionales en cascada que estaba utilizando.

La implantación no fue fácil por diversos motivos:

- La ISO 9000 contiene unos requerimientos rígidos que hubo que flexibilizar.
- Había muchos interesados en el proyecto, por lo que las decisiones a veces se hacían lentas y era difícil acceder a los clientes “reales”.
- Había muchas prácticas y métodos establecidos de trabajo en OCLC establecidos desde muchos años atrás que fueron difíciles de cambiar.
- Conseguir que toda la organización confíe en la nueva metodología. La gestión del cambio no fue fácil, ya que muchas personas se resistían al cambio.
- Calidad. Hubo que reconciliar la visión de la calidad de las metodologías tradicionales con la visión de las metodologías ágiles en la organización.
- Métricas. Hubo que convencer a los usuarios que es más importante tener los requisitos necesarios a tiempo, que tener características superfluas que provoquen retrasos.

Como conclusión, con la aplicación del nuevo proceso de desarrollo los tiempos de entrega del software se redujeron y se demostró el éxito de la nueva metodología. A partir de esta experiencia OCLC continuó en el camino de las metodologías ágiles.

5.4.3. XP – Sabre Airline Solutions

A continuación, se comentarán los resultados de la implantación de la metodología XP Programming en Sabre Airline Solutions [6].

Sabre Airlines es una compañía norteamericana fundada en el año 2000 con más de 1.300 empleados alrededor del mundo. Es el proveedor número uno de soluciones software as a service (SaaS) para la industria de la aviación. Más de 300 aerolíneas de todo el mundo utilizan sus servicios.

Sabre Airlines decidió introducir la metodología XP a partir de investigaciones y revisiones de lecciones aprendidas de organizaciones que ya lo estaban utilizando.

Para realizar el estudio se tuvieron en cuenta dos entregas de proyectos de las mismas características, uno antes de la implantación de la metodología XP, y otro después de dos años haciendo uso de la metodología.

Los resultados fueron muy favorables en nombre de la metodología XP:

- 50% de incremento en productividad
- 65% de mejora en la calidad de las pre-entregas
- 35 % de mejora de la calidad en las entregas finales

5.5. Metodologías recomendadas para los proyectos software

5.5.1. Introducción

A la hora de elegir una metodología, es necesario realizar un estudio sobre las características del proyecto a abordar, y en base a este estudio seleccionar la metodología que mejor se adapte a sus necesidades.

Realmente todas las metodologías aportan valor a la gestión de los proyectos, aunque algunas se adaptan mejor que otras a diferentes tipologías de proyectos.

5.5.2. Problemas de las metodologías tradicionales

Al estudiar las metodologías tradicionales aplicándolas a proyectos software, encontramos los siguientes problemas o dificultades:

- Existen unas costosas fases previas de especificación de requisitos, análisis y diseño. La corrección durante el desarrollo de errores introducidos en estas fases será costosa, es decir, se pierde flexibilidad ante los cambios.
- El proceso de desarrollo está encorsetado por documentos firmados.
- El desarrollo es más lento. Es difícil para los desarrolladores entender un sistema complejo en su globalidad.

5.5.3. Mejoras e inconvenientes que aportan las metodologías ágiles

En los proyectos software, las metodologías ágiles aportan una serie de mejoras sobre las metodologías tradicionales para evitar sus dificultades, tal como se describían anteriormente en el manifiesto ágil, pudiendo sintetizarlas en:

- Capacidad de respuesta a cambios de requisitos a lo largo del desarrollo.
- Entrega continua y en plazos breves de software funcional.
- Trabajo conjunto entre el cliente y el equipo de desarrollo.
- Importancia de la simplicidad, eliminado el trabajo innecesario.
- Atención continúa a la excelencia técnica y al buen diseño.
- Mejora continua de los procesos y el equipo de desarrollo.

Pero también aportan una serie de inconvenientes, como son:

- En caso de proyectos software de gran dimensión, es difícil evaluar el esfuerzo requerido desde el inicio para todo el ciclo de vida del desarrollo.
- Puede faltar el énfasis necesario para el diseño y para la generación de documentación.
- El proyecto puede perder su identidad, y el caso de negocio que lo generó si el cliente o el patrocinador no tiene una visión clara del producto final.
- Es necesario que en los proyectos haya una gran proporción de programadores sénior que sean capaces de tomar decisiones de forma autónoma.

5.5.4. Solución propuesta

En el caso particular de los proyectos software se ha comprobado que un enfoque hacia las metodologías ágiles incrementa las posibilidades de éxito. Esto es debido a que estas metodologías se adaptan muy bien a las características de los proyectos software, en los que los requisitos no suelen estar bien definidos en las etapas iniciales, y no existe un proceso bien definido y estandarizado de "fabricación".

Dentro de las metodologías ágiles, se encuentran muchos casos de éxito (como los descritos anteriormente) utilizando Scrum, Lean Software Development, XP Programming, y DSDM.

Dando un paso más, combinando estas metodologías ágiles con directrices de gestión de proyectos, como CMMI [30] o la guía del PMBOK [32], reducimos sus debilidades y las hacemos más robustas, mejoramos la organización, aumentamos la capacidad de medición (y de mejora continua), y en general se incrementa el control del proyecto.