

UNIVERSIDAD DE OVIEDO

DEPARTAMENTO DE EXPLOTACIÓN Y PROSPECCIÓN DE MINAS

MASTER INTERUNIVERSITARIO EN DIRECCIÓN DE PROYECTOS

TRABAJO FIN DE MASTER

Estudio comparativo entre las metodologías ágiles y las metodologías tradicionales para la gestión de proyectos software

Autor: Manuel José García Rodríguez
Director: Ramiro Concepción Suárez

Julio 2015

Resumen

Este TFM tiene como finalidad adentrarse en el estudio de los métodos de gestión de proyectos de tipo software. Estos proyectos tienen una serie de particularidade que lo hacen distintos al resto de proyectos de ingeniería. Las más importantes es que son intensivos en mano de obra (el desarrollo de software es un proceso intelectual que no se puede automatizar) y en muchos proyectos hay una pobre definición del alcance. Consecuentemente, las metodologías para gestionar estos proyectos tendrán que abordar estas problemáticas y conseguir que el proyecto acabe en plazo, coste y calidad.

Se comienza realizando una introducción sobre la Ing. de Software, la gestión de proyectos y las causas de sus fracasos. Después se estudia las principales metodologías tradicionales, aquellas que se pueden englobar en el paradigma waterfall (cascada), y las metodologías ágiles que asumen que no se pueden establecer los requisitos en la fase inicial del proyecto. Por último, se comparan ambas familias de metodologías desde varios puntos de vista: áreas de conocimiento que gestionan, retorno de la inversión y coste de los cambios.

Palabras clave: *ingeniería de software, gestión proyectos software, metodologías tradicionales, cascada, metodologías ágiles, comparativa, PMBOK, PRINCE2, ICB, SWEBOK, ISO 21500, AM, ASD, AUP, Crystal, DSDM, FDD, LSD, SCRUM, XP*

Abstract

The aim of this TFM is study methodologies of software project management. These projects have a lot of issues that make it different from other engineering projects. The most important issue is that projects are labour intensive (software development is an intellectual process that can not be automated) and many projects have a poor scope definition. Consequently, these methodologies have to manage these troubles and get the project finished on time, cost and quality.

It begins with an introduction of software engineering, project management and the causes of their failures. It continues studying the main traditional methodologies, those that can be included in the waterfall paradigm, and then agile methodologies that assume that it can not set the requirements in the initial phase of the project. Finally, the two families of methodologies are compared from several points of view: knowledge areas they manage, return on investment and cost of changes.

Keywords: *software engineering, software project management, traditional methodologies, waterfall, agile methodologies, comparative, PMBOK, PRINCE2, ICB, SWEBOK, ISO 21500, AM, ASD, AUP, Crystal, DSDM, FDD, LSD, SCRUM, XP*

Agradecimientos

Sería injusto no hacer un breve agradecimiento a las personas que, directa o indirectamente, me han ayudado de alguna manera en poder realizar este Trabajo Fin de Máster. Cuanto más creces como persona y profesional más cuenta te das que casi cualquier trabajo intelectual que realiza una persona es gracias a la ayuda que te han brindado a lo largo de la vida tus profesores, tus compañeros, tus familiares y también al buen trabajo realizado previamente por otras personas sobre el tema.

Quisiera dar las gracias a mi tutor *Ramiro Concepción Suárez* y a los profesores del Máster, con especial atención y admiración a su director *Francisco Ortega Fernández*. También quisiera dar las gracias a mi familia y amigos por su cariño y apoyo constante.

Y, por último, quisiera citar el texto alojado en la placa de inauguración de la presa hidráulica de Belesar (Lugo, Galicia) en 1963. Porque vivimos en una sociedad donde importa más el hoy que el mañana y, sin embargo, nos debería importar mucho más el mañana que el hoy.

> CUANDO CONSTRUIMOS, NO SÓLO DEBEMOS SENTIR
> NUESTRA UTILIDAD PRESENTE. LA OBRA HA DE SER TAL QUE
> MEREZCA EL RESPETO DE NUESTROS SUCESORES, Y AL
> COLOCAR CON ALIENTO Y TRABAJO SILLAR SOBRE SILLAR,
> DEBEMOS PENSAR QUE ALGÚN DÍA LOS HOMBRES
> PUEDEN DECIR MIENTRAS CONTEMPLAN NUESTRA OBRA:
> ESTO QUE NOS LEGARON NUESTROS ANTECESORES
> TAMBIÉN ESTÁ EN EL CAMINO QUE SIGUE INEXORABLE
> LA HUMANIDAD HACIA EL PROGRESO Y LA VERDAD.

Índice

Índice de figuras

Índice de tablas

Glosario

ACM Association for Computing Machinery

AEIPRO Asociación Española de Ingeniería de Proyectos

AENOR Asociación Española de Normalización y Certificación

AM Agile Modeling

ANSI American National Standards Institute

APM Association for Project Management

APMBOK Association for Project Management Body Of Knowledge

AAPP Administraciones Públicas

ASD Adaptive Software Development

AUP Agile Unified Process

CCTA Central Computer and Telecommunications Agency

CMMI Capability Maturity Model Integration

CoRR Computing Research Repository

DAD Disciplined Agile Delivery

DOD Department of Defense (USA)

DSDM Dynamic Systems Development Method

FDD Feature Drive Development

ICB IPMA Competence Baseline

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

IPMA International Project Management Association

ISO International Organization for Standardization

IT Information Technology

LSD Lean Software Development

NASA National Aeronautics and Space Administration

NATO North Atlantic Treaty Organisation

OGC UK Office of Government Commerce

P2M Project & Program Management for Enterprise Innovation

PMAJ Project Management Association of Japan

PMBOK Project Management Body Of Knowledge

PMP Project Management Professional

PMI Project Management Institute

PMO Project Management Office

PRINCE2 Projects in a Controlled Environment

PROMPT Project Resource Organisation Management Planning Technique

RAD Rapid Application Development

RUP Rational Unified Process

SEI Software Engineering Institute

SSADM Structured Systems Analysis and Design Method

SWEBOK Software Engineering Body of Knowledge

TDD Test Driven Development

TIC Tecnologías de la Información y Comunicación

TFM Trabajo Fin de Máster

UML Unified Modeling Language

UNE Una Norma Española

WBS Work Breakdown Structure

YAGNI You Aren't Gonna Need It

XP eXtreme Programming

Parte I

MEMORIA DESCRIPTIVA

Capítulo Primero

Introducción

1.1. Estructura del Trabajo Fin de Máster

La estructura de este trabajo se divide en 2 partes. La primera es la **Memoria Descriptiva** que se compone de 6 capítulos.

Cap. 1: Introducción. Sitúa al lector en la temática explicando qué es un proyecto, en particular los de tipo software y cuáles son las causas de su fracaso. Además describe el enfoque que tiene las metodologías tradicionales y ágiles. También se explica las motivaciones, objetivos y fuentes de información que se han utilizado para la realización del estudio.

Cap. 2: Ingeniería de Software. Se hace una introducción, una evolución histórica del campo así como se definen términos importantes que se repiten y los modelos de ciclo de vida de los proyectos software. Además, se explica las particularidades de la dirección de proyectos software.

Cap. 3: Metodologías tradicionales. Se estudiarán 5 metodologías (PMBOK, ISO 21500, ICB, PRINCE2 y SWEBOK) haciendo una introducción sobre la organización que la creó, su evolución histórica y después se explicará la estructura de la metodología en cuestión. Además se reseñará muy brevemente otras 3 metodologías públicas (SSADM, MERISE Y MÉTRICA) y 2 privadas (APMBOK y P2M).

Cap. 4: Metodologías ágiles. Se hará una introducción a la filosofía ágil y se describirán resumidamente 11 metodologías: Agile Modeling (AM), Adaptive Software Development (ASD), Agile Unified Process (AUP), Crystal, Dynamic Systems Development Method (DSDM), Feature Drive Development (FDD), Lean Software Development (LSD), eXtreme Programming (XP), Scrum, Kanban y Scrumban.

Cap. 5: Comparación entre metodologías. Por un lado se hará una comparación entre varias metodologías ágiles y por otro una comparación entre tradicionales y ágiles desde aspectos distintos.

Cap. 6: Conclusiones finales. Conclusiones del trabajo y líneas futuras del campo.

La segunda parte son los **Anexos**.

Anexo A: Estándares del IEEE e ISO/IEC sobre gestión de proyectos software. Describir resumidamente los estándares reconocidos por el IEEE y el ISO/IEC que tratan de manera más o menos directa la gestión de proyectos software.

Anexo B: Manifiesto Ágil. Transcipción del manifiesto que está alojado en su página web.

1.2. Gestión de proyectos

Hay muchas deficiones de lo que es un proyecto y su gestión por ser unos términos generales que tienen bastantes ámbitos de actuación. Dos posibles definiciones serían las siguientes.

- Un **proyecto** es un esfuerzo temporal encaminado a la creación de un producto, servicio o resultado único.
- La **gestión de proyectos** es aplicar tanto la ciencia como el arte para planificar, organizar, poner en marcha, dirigir y controlar el trabajo de un proyecto para cumplir con los objetivos y metas de la organización.

Una gestión eficiente de proyectos es muy importante para las organizaciones por diversos motivos como la alta complejidad de los proyectos, la fuerte competencia entre empresas, la reducción de costes e ineficiencias, la gestión de equipos humanos, asegurar la calidad, la diversidad de involucrados en el proyectos, el control de la desviación según la planificación y un largo etcétera.

Dentro de una organización (figura 1.1) un grupo de proyectos pueden formar un programa (línea de negocio) y estos a su vez un portfolio (área de negocio). El presente trabajo estudiará algunos aspectos de la gestión de un tipo de proyectos (software), quedando fuera la gestión de programas y portfolios. Para más información consultar *1.4 Relationships among Portfolio Management, Program Management, Project Management and organizational Project Management* [1].

1.3. Qué es software

Como se ha dicho en el apartado anterior, se van a considerar solamente los proyectos de tipo software. El término **software** fue usado por primera por *John W. Tukey* en un

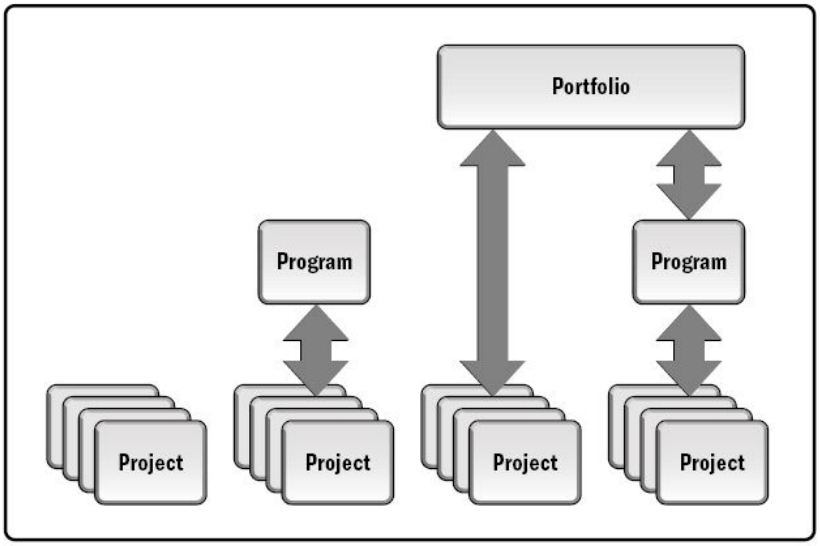

Figura 1.1: Relación entre portfolios, programas y proyectos.

artículo académico en la American Mathematical Monthly en 1958. Se podrían dar varias deficiones de qué es software.

- Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora (Real Academia de la Lengua).
- Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación (IEEE).
- Instrucciones (código de ordenador) que cuando son ejecutadas proveen las deseadas características, funciones y rendimiento; estructuras de datos que permiten a los programas manipular información e información descriptiva para la operación y uso de los programas (*Pressman*, 2010, [12]).

El término **Ingeniería de Software** fue sugerido en las conferencias organizadas por la North Atlantic Treaty Organisation (NATO) en 1968 y 1969 para discutir la llamada "*crisis del software*". Se le llamó así por las dificultades en desarrollar grandes y complejos sistemas en la década de 1960 porque no se obtenían los resultados deseados, había sobrecostes y tenían poca flexibilidad. Se propuso que la adopción de un enfoque ingenieril al desarrollo de software debería reducir los costes de desarrollo y conseguir un software más seguro. Han pasado casi 50 años y todavía los proyectos software tienen altos porcentajes de fracaso (se estudiará en los apartados posteriores) a pesar de la enorme evolución que ha tenido el sector: constante innovación en tecnologías software y hardware, herramientas de desarrollo de software y de gestión, documentación libre y gratuita en Internet, certi-

ficaciones, estándares, metodologías, etc... Esto lleva a pensar a que realizar un proyecto desde el punto de visto técnico y de gestión ha sido y sigue siendo complejo.

¿Cómo se podría definir la Ing. de Software? Se enuncian algunas definiciones en orden cronológico.

- Trata sobre el establecimiento de los principios y métodos de la ingeniería a fin de obtener software de modo rentable, que sea fiable y trabaje en máquinas reales (*Bauer*, 1972).
- La aplicación práctica del conocimiento científico al diseño y construcción de programas de computador y a la documentación asociada requerida para desarrollarlos, operarlos y mantenerlos. Se conoce también como desarrollo de software o producción de software (*Bohem*, 1976).
- El estudio de los principios y metodologías para el desarrollo y mantenimiento de sistemas software (*Zelkovitz*, 1978).
- La aplicación sistemática, disciplinada y cuantificable al desarrollo, operación y mantenimiento del software (ISO/IEC/IEEE, 1993).
- Es la disciplina ingenieril concerniente a todos los aspectos de la producción software: especificaciones, desarrollo, validación y evolución (*Sommerville*, 2011, [13]).
- Es la ingeniería encargada de desarrollar sistemas intensivos en software, esto es, una colección de hardware, software y manuales. El software es el principal componente que integra y coordina la operación del sistema (Project Management Body Of Knowledge (PMBOK), 2013, [1]).

La definición se ha ido ampliando cada vez más a lo largo de los años y evolucionando para adaptarse a la realidad. Hay que desterrar la creencia de que el software es tan sólo el código que se ejecuta en una máquina. Cualquier sistema software necesita una buena documentación para poder comprender su diseño, su arquitectura y los requisitos que satisface para poder operar, mantener y realizar evolutivos en el futuro.

Si nos atenemos solamente al código, Sommerville [13] enumera los siguientes características que definen a un buen software.

Mantenibilidad: el código debe estar escrito de tal manera que pueda evolucionar para satisfacer las necesidades cambiantes del cliente. Esto es un atributo crítico porque los cambios son inevitables en un entorno empresarial cambiante.

Fiabilidad y seguridad: la fiabilidad del software incluye las características de confiabilidad, seguridad y protección. El software confiable no debería causar daños físicos o económicos en caso de fallo. Los usuarios malintencionados no deberían ser capaces de acceder o dañar el sistema.

Eficiencia: el software no debe hacer un uso excesivo de recursos del sistema. Por lo tanto, la eficiencia incluye la capacidad de respuesta, el tiempo de procesamiento, la utilización de la memoria, etc...

Aceptabilidad: el software debe ser aceptado por los usuarios para el que fue diseñado. Esto significa que debe ser comprensible, útil y compatible con otros sistemas que utilizan.

1.4. Motivación, objetivos y fuentes de información

Este Trabajo Fin de Máster (TFM) tiene como objetivo presentar y comparar las distintas metodologías de dirección de proyectos dentro del sector de la Ingeniería de Software. Este sector tiene particularidades propias que lo distinguen del resto de ingenierías y, consecuentemente, la forma de abordar la dirección y gestión de los proyectos será distinta. A juicio del autor, algunas particularidades relevantes son:

- El principal y casi exclusivo recurso para realizar el proyecto son los ingenieros software. Por tanto, toda la actividad del proyecto recae en las personas no pudiendo sistematizar o automatizar gran parte de los trabajos como ocurre en otras ingenierías.
- Generalmente los proyectos empiezan con unas especificaciones vagas porque el cliente no sabe especificar en detalle los requisitos y es a medida que avanza el proyecto cuando va haciendo modificaciones o ampliaciones.
- Aun suponiendo que se tengan unos requisitos totalmente detallados del proyecto (inviable en la práctica), la traducción de éstos a código software supone una actividad creativa en mayor o menor medida porque cada individuo piensa de manera distinta a los demás.
- Hay que definir métricas propias para evaluar y controlar el desarrollo del proyecto por ser una actividad intelectual realizada por personas. No es fácil realizar esta tarea de supervisión como sí ocurre en otras ingenierías.
- La Ing. Software es relativamente nueva (década de los 50 del siglo XX) y no hay tanta experiencia en la gestión y dirección de proyectos como otras ingenierías con más recorrido e historia.
- En grandes macroproyectos donde hay miles de personas diseñando funcionalidades y escribiendo código, resulta fundamental tener una estructura organizativa que tenga una visión global del proyecto y también suficientemente detallada para coordinar a los distintos grupos desarrolladores. La gestión de enormes equipos humanos es una tarea compleja e ineludible para las grandes empresas software.

Por éstas y otras razones han surgido diversas metodologías que tratan de estandarizar la gestión y dirección de proyectos software. Los más importantes y que se van a estudiar en este TFM son:

Project Management Body Of Knowledge (PMBOK) es una guía basada en procesos realizada por el Project Management Institute (PMI) [1]. Es la más utilizada universalmente en el ámbito de la ingeniería y en 2013 sacaron una extensión de su guía para proyectos software [2].

ISO 21500 [6] es la norma de gestión de proyecto de International Organization for Standardization (ISO) siendo prácticamente igual a PMBOK.

IPMA Competence Baseline (ICB) es una guía basada en competencias realizada por el International Project Management Association (IPMA) [3].

Projects in a Controlled Environment (PRINCE2) es una guía realizada por el UK Office of Government Commerce (OGC) [4].

Software Engineering Body of Knowledge (SWEBOK) es una guía específica para proyectos software (no como las anteriores guías que son para proyectos en general) realizada por el IEEE Computer Society [5]. Además de la gestión de proyectos, trata todas las áreas relativas a la Ing. de Software: requisitos, diseño, desarrollo, testeo, mantenimiento, etc...

Metodologías ágiles. Es un relativamente nuevo paradigma de desarrollo software con múltiples técnicas para llevarlo a la práctica. En este TFM se tratarán:

- eXtreme Programming (XP) ([10], [12] y [13]).
- Adaptive Software Development (ASD) [12].
- Dynamic Systems Development Method (DSDM).
- Crystal [12].
- Feature Drive Development (FDD) [12].
- Lean Software Development (LSD) [12].
- Agile Modeling (AM) [12].
- Agile Unified Process (AUP) [12].
- Scrum ([7], [12] y [20]).
- Kanban.
- Scrumban.

Se pretende comparar las metodologías mencionadas y buscar diferencias, similitudes, fortalezas y debilidades. Se van a hacer dos grandes grupos: las tradicionales (PMBOK,

ISO 21500, ICB, PRINCE2 y SWEBOK) y las ágiles. Estas últimas suponen un cambio de paradigma frente a las primeras y parten de la verdad empírica de que no se pueden establecer los requisitos en la fase inicial del proyecto software, sino que lo que hay que fijar son los recursos y los plazos y los requisitos y alcance irán cambiando a lo largo del desarrollo del proyecto. Para entender y elaborar el grupo de metodologías ágiles se ha consultado la bibliografía [8], [9], [11] y [14].

Para realizar la comparación entre ambos grupos se han consultados los artículos [16], [21], [24] y [25]. También se han consultado artículos donde sólo se comparan metodologías ágiles [18] y el impacto que ocasionan en la gestión del proyecto [19]. El autor ha tenido especial interés en buscar bibliografía donde se trate la evolución histórica del desarrollo software [17] y [23] para tener una perspectiva de hacia qué caminos se tiende a ir.

Además de leer y estudiar bibliografía especializada, se han consultado libros de Ing. de Software ([12] y [13]) para comprender mejor todo lo que involucra a esta disciplina y cómo puede afectar a la dirección de proyectos. También se ha estudiado con detenimiento las causas que provocan los fracasos de los proyectos software [15] y [22].

Para tener una idea aproximada de qué es lo que se espera de un TFM se han consultado varias Tesis Doctorales y TFM [26-29] relacionadas con la temática objeto de estudio.

Hoy en día Internet resulta de extraordinario valor para conseguir casi cualquier tipo de información que se desee. Casi toda de la bibliografía previamente comentada se han conseguido de varios repositorios científicos especializados y universitarios [43-47]. Para finalizar la referencia bibliográfica, también se han consultado las páginas web de los organismos más relevantes en el campo objeto de estudio [30-42].

1.5. El fracaso de los proyectos software

El fracaso de proyectos software no es algo reciente sino que, como ya se ha mencionado, en 1968 hubo la llamada "*crisis del software*". Englobó a una serie de sucesos que se venían observando en los proyectos de desarrollo de software:

- No terminaban en plazo.
- No se ajustaban al presupuesto inicial.
- Baja calidad del software generado.
- Software que no cumplía las especificaciones.
- Código inmantenible que dificultaba la gestión y evolución del proyecto.

Todos estos puntos siguen siendo actualmente de suma importancia para el éxito del proyecto porque todavía no han sido resueltos de manera óptima. Esto hace pensar que

los proyectos software tienen una serie de dificultades propias, intrínsecas al campo. La Software Engineering Body of Knowledge (SWEBOK) [5] resume muy acertada y concisamente estas dificultades.

- Los clientes a menudo no saben lo que necesitan o lo que es factible.
- Los clientes suelen carecer del conocimiento sobre la complejidad de los proyecto software, particularmente el impacto de los cambios de requerimientos comenzado el proyecto.
- Es probable que el aumento de la comprensión y las condiciones cambiantes generarán requisitos nuevos o modificaciones de software.
- Como resultado de los cambios de requerimientos, el software es desarrollado usando procesos iterativos en vez de una secuencia de tareas definidas y cerradas.
- La Ing. de Software necesariamente tiene 2 componentes opuestas: creatividad y disciplina. Mantener un apropiado balance entre ambas es a veces difícil.
- El grado de novedad y complejidad es a menudo alto.
- Suele haber una rápida tasa de cambio en la tecnología subyacente. Esto no es tan acusado en otras ingenierías.

Hoy en día se disponen de datos para cuantificar en qué estado finalizan los proyectos y poder extraer causas. El estudio más citado y relevante es el *The CHAOS Report* [15] publicado por el *Standish Group*. Este grupo se creó en 1985 por una serie de profesionales de West Yarmouth (Massachussets) para obtener información de los proyectos software fallidos e intentar encontrar las causas de los fracasos. Su primer informe fue en 1994 y con el tiempo se ha convertido en un referente sobre los factores que inciden en el éxito o fracaso de los proyectos. No está exento de críticas (ver referencia [22]) por su opacidad y posibles sesgos estadísticos aunque hay que decir en su favor que no es trivial conseguir esta información (sensible para las empresas).

Sus datos dicen ser de más de 50.000 proyectos aunque no permiten el acceso a esta información. Según el país de origen, aproximadamente el 60 % son empresas de USA, el 25 % Europeas y las restantes del resto del mundo. Según el tamaño de la empresa, aproximadamente el 50 % son grandes empresas (Fortune Top 1000 Companies), el 30 % son de tamaño medio y el 20 % restante son pequeñas empresas.

El *Standish Group* clasifica los proyectos en tres tipos:

Successful (éxitoso). Se completa en costes, plazos y tiene todas las funcionalidades. El informe deja fuera aspectos discutibles como la calidad, el riesgo y la satisfacción del cliente.

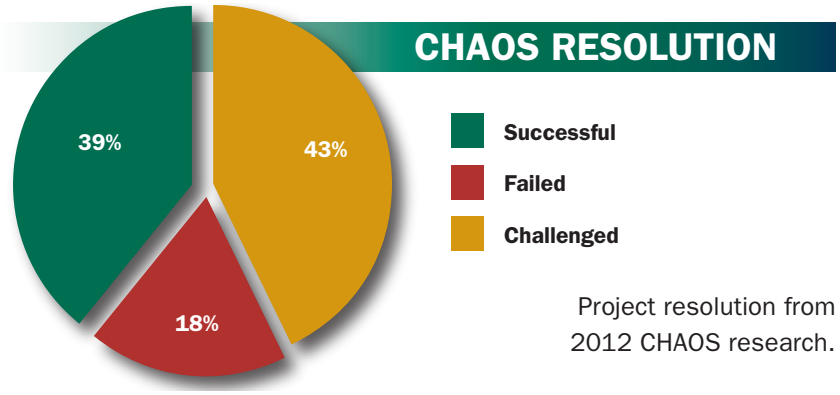

Figura 1.2: Estado de los proyectos a su finalización en el año 2012. Dato de *CHAOS Manifesto 2013*.

	2004	2006	2008	2010	2012
Successful	29%	35%	32%	37%	39%
Failed	18%	19%	24%	21%	18%
Challenged	53%	46%	44%	42%	43%

Tabla 1.1: Evolución histórica (2004-2012) de los estado de los proyectos a su finalización. Dato de *CHAOS Manifesto 2013*.

Challenged (cuestionado). Se completa y es operacional pero no ha cumplido en costes, plazos y/o funcionalidades.

Failed (fracasado). Es cancelado antes de completarse.

En la figura 1.2 se muestra el estado de los proyectos a su finalización en el año 2012. Sorprende el bajo porcentaje de proyectos exitosos debido a que el número de proyectos cuestionados es muy alto. Si no hubiese proyectos cuestionados, habría un 82 % de exitosos. Es decir, si se hubiese hecho una buena labor de gestión del proyecto, no habría tantos proyectos cuestionados.

La tabla 1.1 expone la evolución del porcentaje de éxito de los proyectos del año 2004 al 2012. Se observa que el porcentaje de fracaso es prácticamente constante y el porcentaje de éxito ha aumentado un 10 % por haberse reducido esa misma cantidad los proyectos cuestionados. Por tanto, se aprecia que en esos 8 años ha habido una mejor gestión de los proyectos software pero todavía queda un largo camino por recorrer.

La experiencia ha demostrado que la relativamente baja tasa de éxito está relacionado con plazos de entrega inalcanzables o presupuestos demasiados optimistas sin utilizar métricas para llegar a estimaciones aproximadas. Las empresas maduras en desarrollo de software son atinadas a la hora de estimar porque, a lo largo de su vida como empresa,

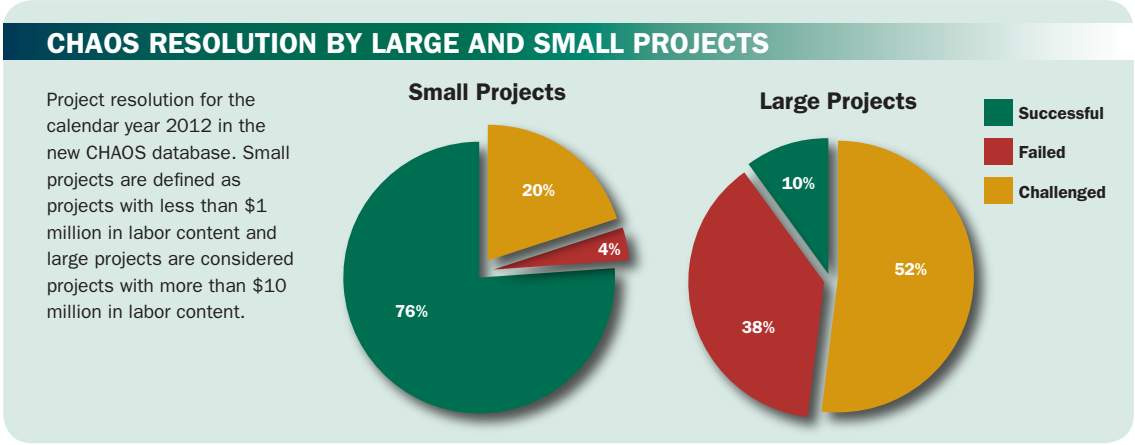

Figura 1.3: Estado de los proyectos a su finalización según su tamaño (pequeños y grandes). Dato de *CHAOS Manifesto 2013*.

han desarrollado métricas y herramientas eficientes para medir los costes y plazos. Esto nos lleva a pensar también que estimar (y gestionar) proyectos pequeños es en la práctica más fácil que estimar grandes proyectos. Esta intuición de sentido común queda patente en la figura 1.3 con unos porcentajes de éxito muy distinto según sea la envergadura. Luego sigue siendo un reto a día de hoy gestionar grandes proyectos y no es por falta de medios porque estos tipos de proyectos los realizan grandes multinacionales de Information Technology (IT).

Otro punto de vista interesante a la hora de analizar el éxito de los proyectos es clasificarlos por las metodologías con las que fueron realizados. De esta forma se pueden extraer conclusiones si unas metodologías alcanzan mayores tasas de éxito que otras. En la figura 1.4 se dividen proyectos pequeños según metodologías ágiles y tradicionales, siendo los porcentajes de éxito, cuestionados y fracasados bastante similares. Por tanto, no podemos decir a priori que un tipo de metodologías sean más exitosas que otras. Sin embargo, también nos podríamos cuestionar si dichos proyectos se han gestionado eficientemente utilizando realmente dichas metodologías o, por contra, no las han sabido llevar a la práctica completamente para explotar todas sus bondades.

Sin ánimo de ser exhaustivo, se puede concluir diciendo que los proyectos software se enfrentan a muchos y diversos factores que dificultan finalizarlos correctamente. La causa principal es que el software es el producto de una actividad intelectual realizado por seres humanos, con todo lo que eso supone.

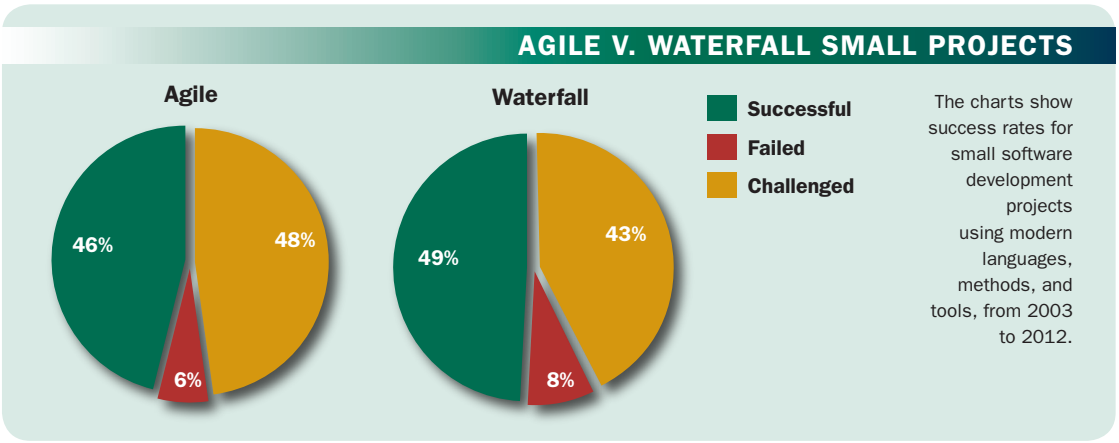

Figura 1.4: Estado de pequeños proyectos a su finalización realizados con metodologías ágiles o tradicionales (waterfall). Dato de *CHAOS Manifesto 2013*.

1.6. Visión tradicional vs ágil

Como se ha comentado en los apartados anteriores, este trabajo pretende comparar distintas metodologías. Se van a agrupar en dos grandes grupos: tradicionales y ágiles. A las tradicionales también se les llaman predictivas o en cascada (waterfall en inglés).

En la década de los 90 surgieron metodologías de desarrollo de software "*ligeras*", después llamadas ágiles, que buscaban reducir la probabilidad de fracaso por no estimar correctamente costes, plazos y alcances de los proyectos. Estas metodologías nacieron como reacción a las metodologías existentes con el propósito de disminuir la burocracia que implica su aplicación. Las metodologías tradicionales buscan imponer disciplina al proceso de desarrollo de software y de esa forma volverlo predecible y eficiente. Para conseguirlo se sustentan en un proceso detallado con énfasis en la planificación propio de otras ingenierías. El principal problema de esta filosofía es que hay muchas actividades que realizar para seguir la metodología y esto retrasa la etapa de desarrollo además de no ser fácilmente adaptables a los cambios. Por tanto, las metodologías ágiles tienen dos diferencias fundamentales con las metodologías tradicionales: las ágiles son adaptativas (no predictivas) y orientadas a las personas (no a procesos).

En resumen, son dos filosofías opuestas de cómo gestionar y desarrollar proyectos software aunque no están claramente delimitadas. Para hacerse una idea de ambas concepciones contrapuestas léase la tabla 1.2.

De una manera gráfica, el triángulo de la gestión de proyectos tiene distinto significado para ambas visiones (figura 1.5). En la visión tradicional los requisitos están fijados siguiendo un plan en los que los plazos y los recursos son estimados para cumplir dicho plan. Por contra en la visión ágil son los plazos y recursos los fijados y siguiendo un plan de

Aspectos	Visión tradicional	Visión ágil
Ciclo de vida	Secuencial: cascada, espiral, ...	Iterativo, modelos evolutivos.
Estilo de desarrollo	Anticipativo.	Adaptativo.
Requisitos	Conocidos, estables, claramente definidos y documentados.	Desconocidos a priori, definidos durante el proyecto.
Arquitectura	Pesada y sobredimensionada para los actuales y futuros requisitos.	Filosofía You Aren't Gonna Need It (YAGNI)
Gestión	Centrada en procesos: mando y control.	Centrada en la gente: liderazgo y colaboración.
Documentación	Detallada, conocimiento explícito.	Ligera, conocimiento tácito.
Metas	Previsibilidad y optimización.	Exploración y adaptación.
Cambios	Aversión al cambio.	Acepta el cambio.
Organización del equipo	Equipos preestructurados.	Equipos autoorganizados.
Involucración del cliente	Pasivo, poca involucración.	Activo, un miembro más del equipo.
Cultura organizativa	Jerárquica con mando y control establecido.	Menos jerárquica con liderazgos y colaboraciones.
Desarrollo del software	Acercamiento universal con solución predecible y altamente segura.	Acercamiento flexible adaptado a las necesidades particulares del proyecto.
Medida del éxito	Conforme a un plan.	Evaluar globalmente el negocio.

Tabla 1.2: Comparación de la perspectiva tradicional y ágil en el desarrollo de software.

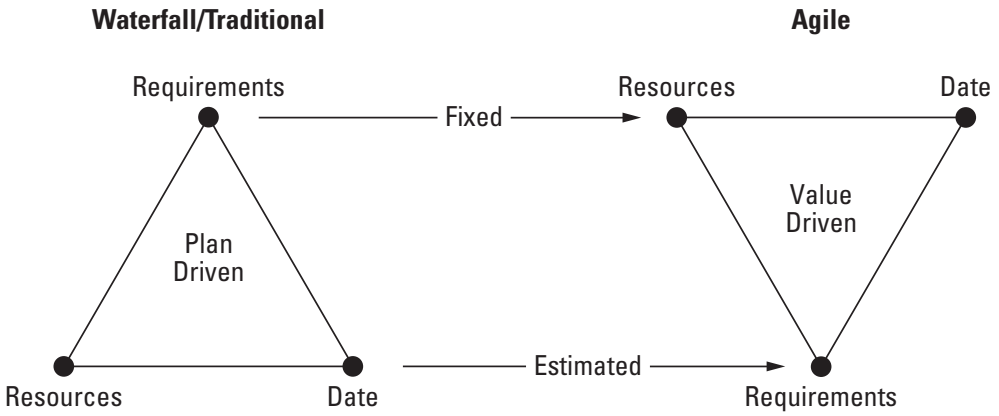

Figura 1.5: Triángulo de la gestión de proyectos.

valor ganado se va desarrollando el proyecto según unos requisitos cambiantes acordados entre todas las partes periódicamente.

Capítulo Segundo

Ingeniería de Software

2.1. Introducción

Gestionar un proyecto implica tener que adoptar una solución de compromiso entre las restricciones de alcance, calidad, plazos, presupuesto, recursos y riesgos. Además en la Ing. de Software hay factores tecnológicos específicos que pueden dar lugar a más restricciones.

- Estado actual de las tecnologías hardware y software: plataformas, protocolos de comunicaciones, sistemas operativos, etc...
- Herramientas de desarrollo software.
- Compatibilidad software con sistemas en uso y adaptación a sistemas a futuros.
- Reuso de componentes software utilizando librerías.
- Uso de software de código libre o propietario.
- Otros: seguridad, disponibilidad y escalabilidad del sistema, estabilidad, testeabilidad, eficiencia computacional, soporte, mantenibilidad del código, etc...

La gestión de proyectos software es una tarea desafiante por muchas cuestiones. A continuación se citan algunas que son significativas.

- El código software es un producto intangible y maleable por ser texto escrito.
- Los cuestiones clave que hacen un proyecto software desafiante son la complejidad del proyecto/producto, escalabilidad no lineal de los recursos humanos, incertidumbre inicial del alcance del proyecto/producto y el conocimiento se va obteniendo según el proyecto evoluciona.

- Los requisitos suelen cambiar durante el desarrollo del proyecto según se va obteniendo conocimiento del proyecto y su alcance se va acotando cada vez más.
- El software es un producto directamente realizado por procesos cognitivos humanos.
- La comunicación y coordinación entre el equipo del proyecto y los involucrados a menudo no es claro ni eficiente (ejemplo gráfico en la figura 2.1).
- Las planificaciones y estimaciones iniciales son poco precisas porque dependen fuertemente de los requisitos (imprecisos) y de la productividad del equipo de trabajo (bastante variable).
- El exhaustivo testeo del software es irrealizable en la práctica y más si cabe cuanto más grande, complejo e interrelacionado sea el sistema software.
- A menudo el proyecto incluye software de terceros o desarrollos de interfaces a otros software. Esto puede dar problemas de integración, rendimiento, estabilidad, etc...
- La mayoría del software está interconectado por lo que la seguridad de la información es crucial.
- La cuantificación objetiva de la calidad del software es muy difícil por su naturaleza intangible.
- Los desarrolladores utilizan procesos, métodos y herramientas que están en constante cambio y actualización.
- Un producto software puede necesitar operar en plataformas hardware e infraestructuras software distintas.

El software es intangible por ser un producto intelectual. Esto implica que no tiene entidad física que pueda ser evaluada por métodos clásicos (masa, volumen, conductividad, ...) pero sin embargo sí tiene otros factores. Por ejemplo, las necesidades de memoria y procesador, ancho de banda de comunicación, etc...

Su naturaleza intangible crea retos a la hora de medir el estado actual de desarrollo del proyecto y, por tanto, es complicado de monitorizar y controlar. Su naturaleza maleable tiene aspectos positivos y negativos. Es beneficioso que muchas veces se pueda responder rápidamente a cambios que necesite el usuario. Sin embargo, interrumpir trabajo en marcha o rehacer el que ya está hecho por cambios en los requisitos puede tener un impacto negativo en el presupuesto y en los plazos de entrega.

La planificación y estimaciones iniciales de costes y plazos son difíciles de hacer en cualquier proyecto pero en este tipo de proyectos lo es todavía más por varias razones.

1. El software es creado por procesos cognitivos de los programadores.

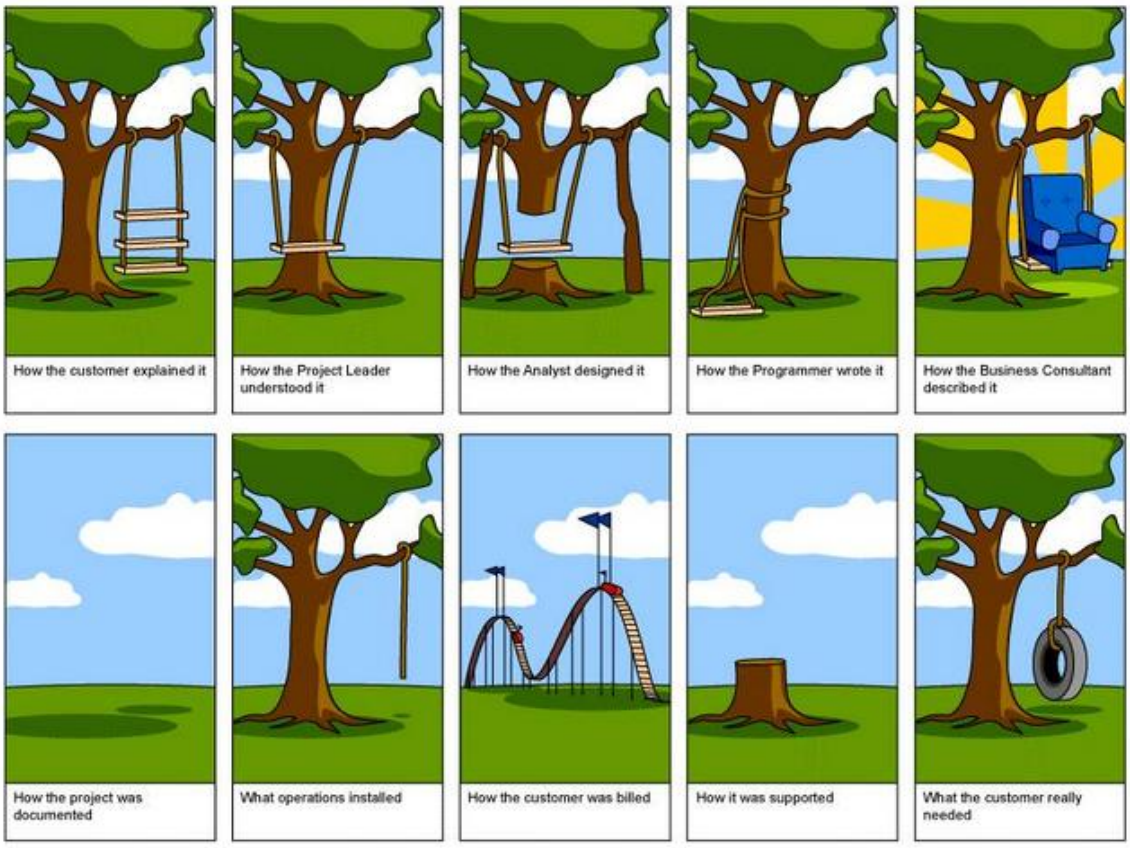

Figura 2.1: La comunicación ineficiente entre los involucrados de proyectos software.

2. La productividad de cada programador varía sensiblemente según su destreza, experiencia, conocimientos de las tecnologías usadas, visión de los requisitos de usuario, etc... Varía tanto en cantidad de trabajo como en calidad.

3. Los requisitos en los que se basan las estimaciones a menudo están pobremente definidos.

4. La continua evolución tecnológica hace que los datos históricos generalmente no sirvan para estimar nuevos proyectos.

La gestión de la calidad también es otra tarea fundamental que se debe hacer en cualquier proyecto para que finalice con éxito. Como esta fuera del alcance del TFM, simplemente se citan los atributos claves de la calidad en proyetos software: seguridad, confiabilidad, resiliencia, dependibilidad, escalabilidad, rendimiento, facilidad de aprendizaje y uso del software creado, interpretación de los mensajes de error, disponibilidad, accesibilidad, eficiencia, flexibilidad, interoperabilidad, robustez, testabilidad, mantenibilidad, portabilidad, extensibilidad y reusabilidad.

2.2. Dirección de proyectos software

Dirigir proyectos consiste en la aplicación de conocimientos, habilidades, herramientas y técnicas a las actividades del proyecto para cumplir con los requisitos del mismo. Además de que cada proyecto es diferente, las empresas de software tienen características muy distintas: volumen de facturación, número de empleados, especialización, organización departamental (por ej., por productos, servicios, funcionalidades o tecnologías), distribución geográfica, etc... Por tanto, la manera de dirigir proyectos no será homogénea. A continuación se tratan los dos roles tradicionales en la dirección y gestión de proyectos particularizándolos para los proyectos software.

2.2.1. Oficina de Gestión de Proyectos Software

Una Oficina de Gestión de Proyectos (Project Management Office (PMO)) es una estructura de gestión que estandariza procesos relativos a proyectos y facilita compartir recursos, metodologías, herramientas y técnicas. Las responsabilidades de una PMO pueden variar desde proveer soporte para la gestión de proyectos a ser el responsable directo de uno o más proyectos. Dependerá de la organización establecer o no una PMO y sus competencias.

La PMO tiene como función principal ayudar a los project managers de las siguientes formas.

- Gestionar recursos compartidos transversales a todos los proyectos administrados por el PMO.
- Identificar y realizar buenas prácticas, estándares y metodologías para la gestión de proyectos.
- Enseñar, entrenar y tutorizar a los project managers.
- Auditar los proyectos que se llevan a cabo para controlar que cumplen la normativa que ha establecido la organización.
- Desarrollar y gestionar dicha normativa: políticas, procedimientos, plantillas y otra documentación compartida.
- Coordinar la comunicación entre proyectos.

Además, una PMO de proyectos software también debería de encargarse de las siguientes tareas por las particularidades y complejidad intrínsecas a este tipo de proyectos.

- Proveer un repositorio común con información histórica relativa a esfuerzos, costes, plazos, fallos, involucrados y riesgos de proyectos ya realizados por la organización.
- Usar dicho repositorio para crear modelos de costes ajustados a la organización.
- Usar dicho repositorio para analizar las fortalezas y debilidades de los proyectos realizados que sirvan de base para promover iniciativas de mejora en la organización.
- Ayudar a los project managers para hacer las estimaciones de costes y plazos y preparar los planes de proyecto.
- Adquirir y homogeneizar el uso de nuevas herramientas y plataformas para desarrollar software.
- Mantener librerías con módulos genéricos de código reusable.

2.2.2. Project manager

El project manager es la persona asignada por la organización para liderar un equipo y ser el responsable de lograr los objetivos. Por tanto, el project manager persigue los objetivos particulares del proyecto asignado mientas las PMOs tienen responsabilidades más genéricas y transversales que ya se han mencionado. Además de las habilidades básicas que debe tener todo project manager (ver figura 2.2), para liderar proyectos software habría que tener muy presente las siguientes facetas.

Figura 2.2: Habilidades del project manager.

- Realizar estimaciones y planes al inicio del proyecto y también según avanza el proyecto para adaptarse a los cambios.
- Monitorizar y controlar los hitos, el presupuesto, los requisitos del software, el rendimiento del equipo, la utilización óptima de los recursos e identificar riesgos potenciales.
- Liderar y dirigir el proyecto teniendo una visión clara de los requerimientos y las restricciones.
- Mantener la conformidad del contrato del proyecto.
- Cohesionar, inspirar y facilitar al equipo de desarrollo sus demandas.
- Comunicación fluida con los involucrados para suplir sus carencias técnicas usando terminología y conceptos que les sean familiares.

En pequeños equipos (por ejemplo menos de 10 personas) el project manager puede tener roles adicionales (líder de equipo, diseñador software, analista, etc...) o tener varios proyectos a su cargo. Las habilidades interpersonales del project manager son fundamentales para lograr una comunicación y coordinación eficaz entre todas las partes. Las más relevantes son: liderazgo, humildad, escuchar atentamente, construir equipos, motivación, comunicación, colaborar y compartir conocimiento, mediar, tomar decisiones y negociación.

Toda la problemática ya comentada en apartados anteriores hace que dirigir y gestionar proyectos software sea una tarea compleja. Las empresa del sector cada vez son más conscientes que tener personal formado y acreditado en estas áreas es fundamental para el desarrollo de sus organizaciones siendo una profesión en ascenso (ver figura 2.2).

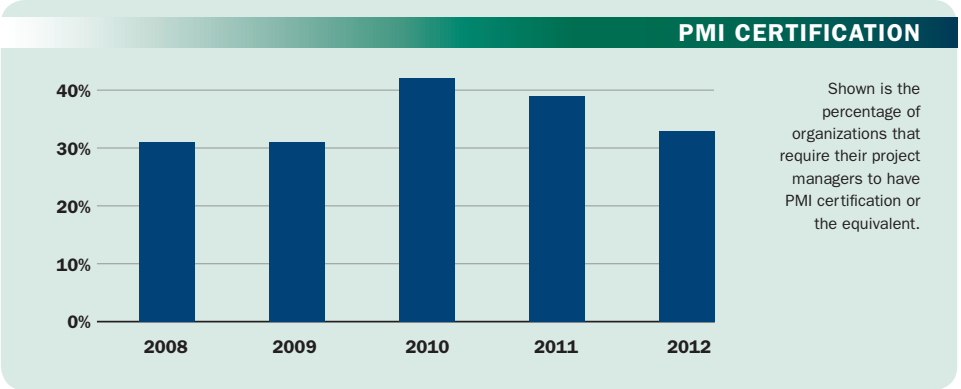

Figura 2.3: Porcentaje de organizaciones que requieren project managers certificados en PMI o equivalente. Dato de *CHAOS Manifesto 2013*.

2.3. Evolución histórica

Este apartado hace un repaso a la historia de la Ing. de Software. Es decir, identifica los fenómenos subyacentes que han influido en la evolución de la disciplina así como las principales fuentes de cambio en el pasado y presente. Como dijo el intelectual Jorge Santayana *"Quien no recuerda el pasado está condenado a repetirlo"*. El resumen que se aborda a continuación se esquematiza en la figura 2.4.

- **1950-1960: Ingeniería Hardware**. La Ing. Software es una nueva disciplina realizada por ingenieros hardware o matemáticos en el seno de grandes organizaciones con fines militares o relacionadas. Uno de los mayores proyectos de procesamiento de información fue el Semi-Automated Ground Environment (SAGE) para la defensa aérea de USA y Canadá. El coste principal de los proyectos es el hardware (horas-máquina) siendo las horas-hombre apenas relevante en el coste final. Es una actividad incipiente sin metodologías ni estándares.

- **1960-1970: Desarrollo del Software**. El software coge otro camino al del hardware por ser actividades cada vez más diferentes. Se programa de la manera *codificar-y-arreglar* en comparación con la exhaustiva manera de programación de los ingenieros hardware. Los programas espaciales de la NASA fomentan este campo. Se crean departamentos de ciencias de la computación e informática en las Universidades, empresas privadas de desarrollo de software y lenguajes de alto nivel como Cobol y Fortran.

- **1970-1980: Formalidad y procesos Waterfall**. La reacción al enfoque *codificar-y-arreglar* fue programar más cuidadosamente precedido de un diseño y una ingeniería de requisitos. Se utiliza el modelo Waterfall (en cascada) para el ciclo de vida del proyecto pero a finales de 1970 se nota que tienen problemas de escalabilidad y

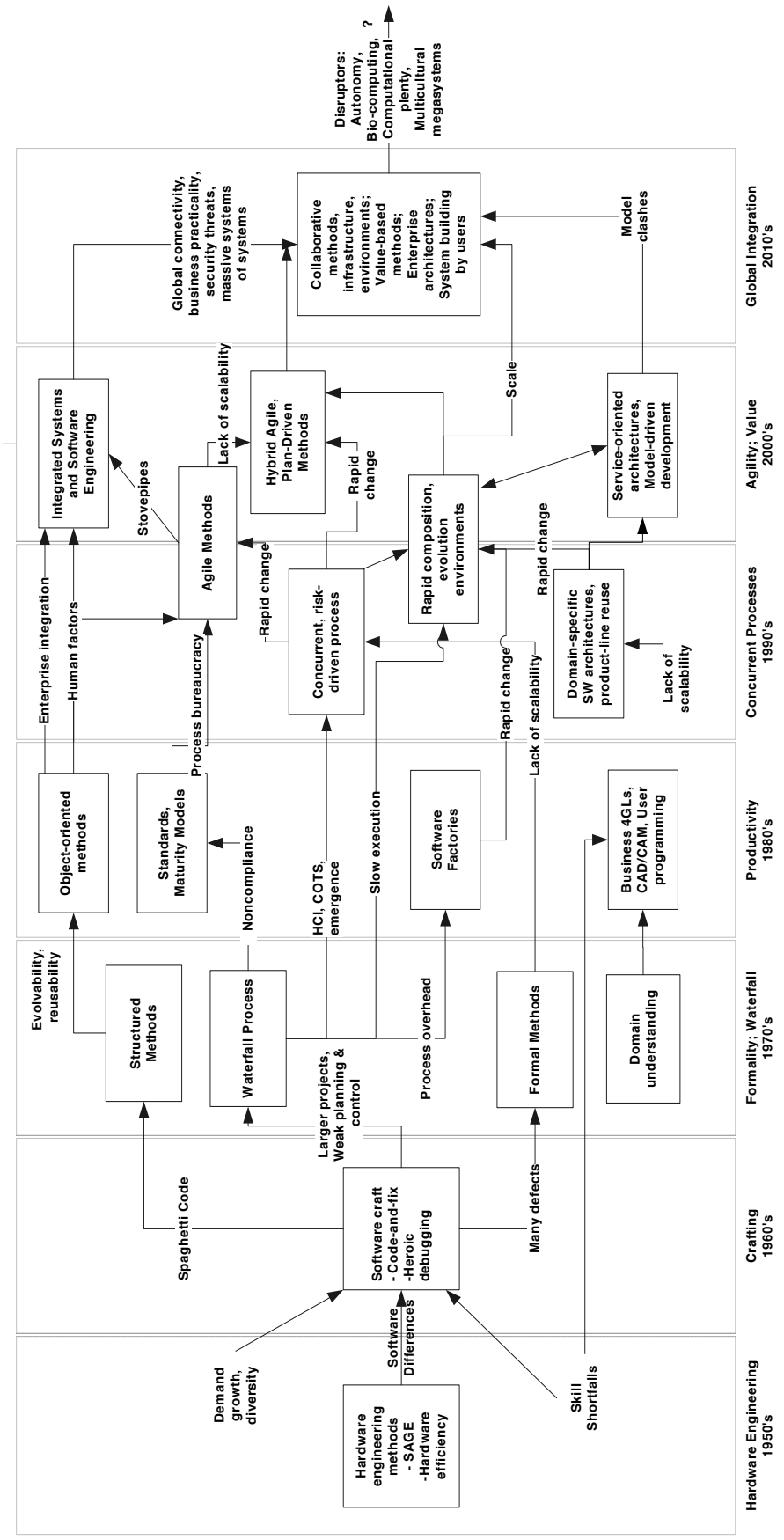

Figura 2.4: Tendencias históricas de la ingeniería de software.

usabilidad. Se avanzó considerablemente en la disciplina: modelos de estimación de fiabilidad del software, métodos cuantitativos para la calidad, modelos de estimaciones de costes y plazos, etc... Muchas organizaciones ya tienen unos costes en personal superiores que los del hardware, es decir, se han invertido los costes.

- **1980-1990: Productividad y escalabilidad**. Junto con las mejores prácticas de la década de 1970, la de 1980 llevó a una serie de iniciativas para hacer frente a los problemas de la década anterior y para mejorar la productividad y la escalabilidad. Se crean estándares internacionales y modelos de madurez. Hay un desarrollo notable de las herramientas software en los campos del test, en la gestión de la configuración y entornos para la ayuda del desarrollo software. Se incrementa la productividad por los sistemas expertos, los lenguajes de muy alto nivel, orientación a objetos (C++, Smalltalk, Eiffel) y ordenadores potentes pero sobre todo por la reutilización del software: sistemas operativos más potentes (ratón y ventanas interactivas posibilitando el *what you see is what you get*), librerías gráficas, sistemas de gestión de bases de datos, etc...

- **1990-2000: Procesos concurrentes vs. secuenciales**. Continuó el fuerte impulso de la programación orientada a objetos fortaleciéndose a través de avances como los patrones de diseño, arquitecturas software y el Unified Modeling Language (UML). Se expandió y popularizó Internet y la World Wide Web. Hubo 4 fenómenos relevantes:

  - Énfasis en lanzar al mercado el producto lo más rápidamente posible para tener una ventaja competitiva.
  - Métodos concurrentes (modelo en espiral) para aumentar la eficacia y reducir los riesgos.
  - El movimiento Open Source toma importancia. En 1985 *Richard Stallman* crea la Free Software Foundation y la licencia GNU. Esto estableció las condiciones de uso gratuito y la evolución de herramientas libres: el compilador GCC, emacs, Linux, Apache, TCL, Python, Perl, Mozilla, etc...
  - Usabilidad e Interacción Hombre-Máquina (HMI). Se aumentó la usabilidad de los productos de software para gente sin basta experiencia en programación.

- **2000-2010: Agilidad y valor**. Continúa la tendencia hacia el desarrollo rápido de aplicaciones, una aceleración en nuevas tecnologías de la información y cambios en las organizaciones (fusiones, adquisiciones y nuevas empresas). La década se puede dividir en 6 hechos importantes.

  - Metodologías ágiles. En 2001 se publica el *Agile Manifesto* como respuesta al cambiante mercado que exige productos rápidamente, baratos y que se ajusten a las necesidades del cliente.
  - Valor. La Ing. de Software se debe basar en aumentar el valor del producto. Esto se hace con incrementos pequeños priorizando los que más valor aporten al cliente. Las metodologías ágiles se basan en este principio.

  - Confiabilidad del software. Cada vez más la gente y las organizaciones dependen de sistemas software. Por tanto, la confiabilidad y seguridad de los sistemas debe aumentar aunque esto no es la máxima prioridad para muchos desarrolladores.
  - Aunar las bondades de los programas Open Source con los propietarios.
  - La gente empieza a utilizar masivamente Internet: nuevas formas de relacionarse entre ellos (redes sociales) y con las AAPP (administración digital), educación a distancia (e-learning), etc...
  - Integración de sistemas. Las organizaciones necesitan integrar sistemas diversos para aumentar su productividad y reducir costes. Los ingenieros de sistemas tienen que aumentar sus conocimientos software por tener cada vez más peso en dichos sistemas.

- **2010-Futuro: Globalización y macro-sistemas (sistemas de sistemas)**. Grandes superestructuras de redes y sistemas en todas las áreas económicas, industriales y sociales.

2.4. Principios y nomenclatura

A continuación se hace un pequeño diccionario con definiciones relevantes para este trabajo así como la palabra equivalente en inglés.

- **Actividad** (activity). Una porción de trabajo temporal que es realizado durante el curso de un proyecto.
- **Alcance** (scope). La suma de productos, servicios y resultados que son realizados en forma de proyecto.
- **Cliente** (customer). Es la persona/s u organización/es que pagarán el producto, servicio o resultado del proyecto. Los clientes pueden ser internos o externos para la organización que realiza el proyecto.
- **Estándar** (standard). Un documento que provee, para uso común y repetitivo, reglas, guías o características para actividades o sus resultados para lograr un óptimo grado de éxito en una circunstancia específica.
- **Herramienta** (tool). Algo tangible, como una plantilla o un programa software, usado para realizar una actividad que genera un producto o resultado.
- **Hito** (milestone). Un punto significativo o evento en el calendario de un proyecto.
- **Involucrado** (stakeholder). Un individuo, grupo u organización que puede afectar o estar afectado por una decisión, actividad o resultado de un proyecto.

- **Metolodogía** (methodology). Un sistema de prácticas, técnicas, procedimientos y reglas usados por quien trabaja en una disciplina.
- **Plantilla** (template). Un documento con formato predefinido que tiene una estructura para recoletar, organizar y presentar información.
- **Política** (policy). Una serie de acciones estructuradas adoptadas por una organización.
- **Práctica** (practice). Un tipo específico de actividad que contribuye a la ejecución de un proceso y que puede emplear una o más técnicas o herramientas.
- **Procedimiento** (procedure). Un método establecido para conseguir un rendimiento o resultado.
- **Proceso** (process). Una serie sistemática de actividades dirigidas a crear un resultado final mediante unas actuaciones sobre unos inputs que generarán unos outputs.
- **Producto** (product). Un bien (material o inmaterial) que es producido y es cuantificable. Puede ser un item final o ser un componente a su vez para otro producto.
- **Recurso** (resource). Habilidades de los trabajadores (individualmente o en equipo), equipamiento, servicios, suministros productos, materiales, presupuestos o fondos.
- **Requisito** (requirement). Una condición que es requerida para ser presentada en un producto, servicio o resultado para satisfacer un contrato o especificación formal.
- **Resultado** (result). Un output realizado por las actividades y procesos de gestión de proyectos. Por ejemplo, integración de sistemas, reestructuración organizacional, revisión de procesos, tests o documentación (políticas, planes, estudios, especificaciones, reportes, etc...). Un resultado no es un producto.
- **Riesgo** (risk). Un potencial evento o condición que, si ocurre, tiene un efecto positivo o negativo en uno o más objetivos del proyecto.
- **Técnica** (technique). Un procedimiento sistematizado y bien definido empleado por un trabajador para realizar una actividad que genera un producto, resultado o servicio y que puede emplear uno o más herramientas.

2.5. Modelos de ciclo de vida

Todos los modelos tienen explícita o implícamente 5 fases: requisitos, diseño, desarrollo, pruebas y mantenimiento. Cómo se organizen, su interrelación y la importancia de cada una de ellas dentro del ciclo de vida dependerá del modelo escogido.

1. **Requisitos**. Es la toma formal de requerimientos que deberá cumplir el proyecto. Los clientes suelen tener una idea difusa y abstracta del resultado final, pero no sobre las funciones que debería cumplir el software.

2. **Diseño**. Es la fase que en que se establece cómo se va a estructurar el proyecto software (a alto nivel) según los requisitos y funcionalidades que se han especificado.

3. **Desarrollo**. Es la fase en el que los ingenieros software programan el código para el proyecto partiendo del análisis y diseño de la solución ya realizados.

4. **Pruebas**. Se testea el software desarrollado para detectar posibles errores lo antes posible y subsanarlos. Los errores humanos dentro de la programación pueden ser muchos y aumentan considerablemente con la complejidad del problema. Por tanto, es una parte esencial del proceso de desarrollo del software.

5. **Mantenimiento**. Una vez que se despliega el software, puede ser necesario realizar un mantenimiento para solucionar errores que han sido detectados en servicio o desarrollar evolutivos del software con nuevas funcionalidades.

En todas las fases anteriores habría que incluir la tarea de documentación (procedimientos, guías, esquemas, etc...). Son la recopilación escrita en sus diferentes formas que se hace durante toda la vida del proyecto. La importancia de la documentación radica en que a menudo un código escrito por una persona es modificado por otra. Por ello la documentación sirve para ayudar a comprender, usar o mantener un programa.

El estándar internacional que regula el método de selección, implementación y monitorización del ciclo de vida del software es el ISO/IEC 12207:2008 *Standard for Systems and Software Engineering - Software Life Cycle Processes*. La estructura ha sido concebida de manera que pueda ser adaptada a las necesidades de cualquiera que lo use.

En los siguientes apartados se resumirán brevemente los distintos modelos de ciclo de vida software que hay.

2.5.1. Cascada

Es el modelo más antiguo, propuesto por *Winston Royce* en 1970. Es un modelo secuencial de fácil entendimiento e implementación (figura 2.5). Refuerza el buen hábito de definir antes de diseñar y diseñar antes que programar. Sin embargo, ésta es su gran debilidad porque esperar tener requerimientos definidos completamente al inicio del proyecto es una situación ideal que casi nunca sucede en la realidad. Además no utiliza la iteración ni posibilita mecanismos para hacer cambios en los requisitos por lo que nunca encajará en las metodologías ágiles.

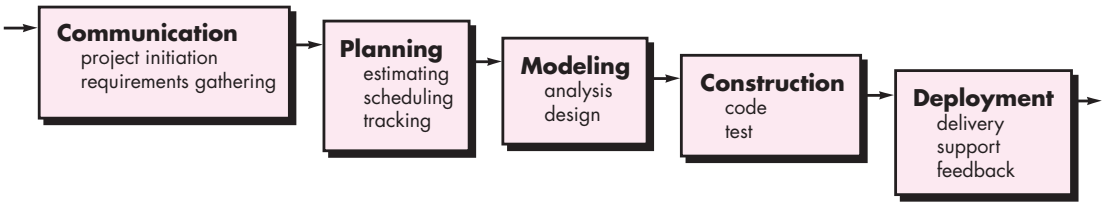

Figura 2.5: Esquema del modelo de ciclo de vida en cascada.

2.5.2. V

El modelo V es una variación del modelo anterior donde se busca hacer la tarea de pruebas más efectiva y productiva (figura 2.6). Para ello los planes de pruebas se van elaborando a medida que avanza el desarrollo del proyecto. Es decir, el equipo de pruebas hace unos planes de tests en paralelo a las activades de desarrollo y genera entregables de pruebas para validar el sistema y así asegurar la calidad. En realidad, no hay diferencias fundamentales entre el modelo en V y en cascada. Sus fortalezas y sus debilidades son prácticamente las mismas.

2.5.3. Incremental

El desarrollo incremental se basa en la idea de desarrollar una implementación inicial, mostrándosela al cliente y evolucionándola a través de versiones hasta que el software tenga todas las funcionalidades pedidas por el cliente (figura 2.7). El modelo incremental combina procesos con flujos lineales y paralelos. Es decir, emplea secuencias lineales de forma escalonada a medida que avanza el tiempo del calendario y cada secuencia produce "*incrementos*" (entregables del software) de una manera similar a los incrementos de los modelos evolutivos. Por tanto, el modelo incremental se adapta a la filosofía de las metodologías ágiles de desarrollo software.

Tiene 3 ventajas importantes: el coste por realizar cambios se reduce, se obtiene más fácilmente realimentación del cliente y se desarrolla rápidamente el software útil para el cliente. Sin embargo, este enfoque incremental (ágil) genera desventajas: la estructura del sistema tiende a degradarse tras una nueva versión (a no ser que se refactorize el software) y los procesos no son visibles quedándose la documentación desfasada tras la siguiente versión.

2.5.4. Prototipo

Es un modelo de tipo evolutivo. El paradigma del modelo (figura 2.8) empieza con la etapa de comunicación para establecer en qué consistirá el prototipo. Idealmente, es-

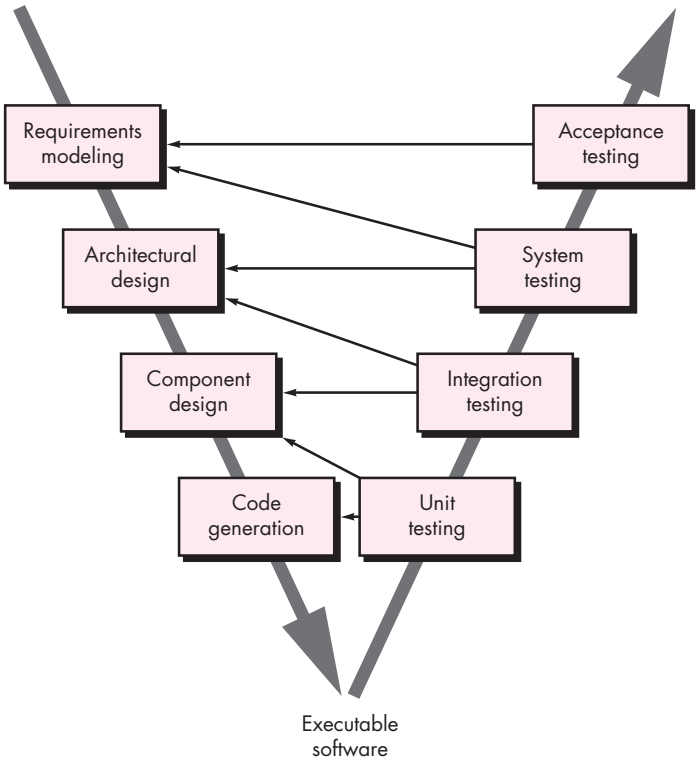

Figura 2.6: Esquema del modelo de ciclo de vida en V.

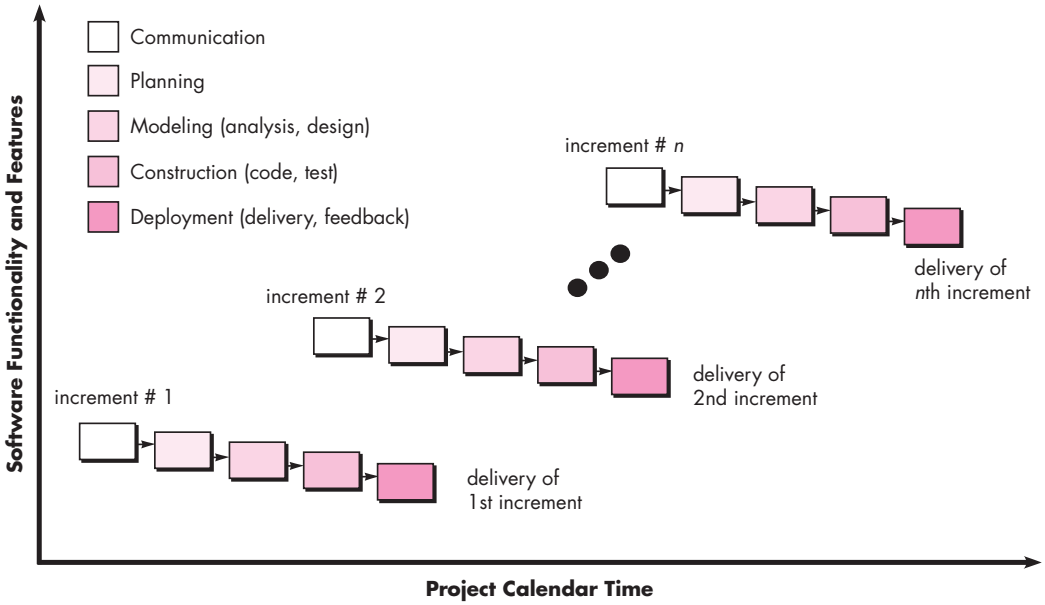

Figura 2.7: Esquema del modelo de ciclo de vida incremental.

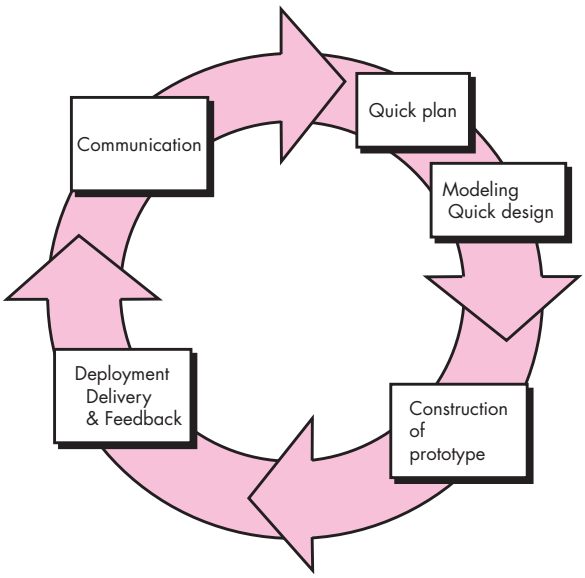

Figura 2.8: Esquema del modelo de ciclo de vida prototipado.

te modelo sirve como herramienta para identificar requisitos del software. Por esto es especialmente recomendable en aquellos proyectos donde el cliente tiene unos requisitos generales pero no se identifican las funcionalidades y características detalladas. Es decir, el prototipo ayuda a todos los involucrados del proyecto a entender mejor lo que se va a construir cuando los requisitos son difusos.

2.5.5. Espiral

Es otro modelo evolutivo creado por *Barry Boehm* en 1988. Es un modelo (figura 2.9) que une la naturaleza iterativa del prototipo con los aspectos controlados y sistemáticos del modelo en cascada. Proporciona los mecanismos para el desarrollo rápido de versiones cada vez más completas y complejas del software. Cada bucle en la espiral representa una fase del proceso de software. Por ejemplo, el bucle más interno podría estar preocupado con la viabilidad del sistema, el siguiente bucle con los requisitos, el siguiente con el diseño del sistema, etc...

Tiene un enfoque realista para el desarrollo de sistemas software de gran escala. Dado que el software evoluciona a medida que el proceso avanza, el desarrollador y cliente comprenden y reaccionan mejor a los riesgos en cada evolución. Por tanto, el punto fuerte de este modelo frente a otros es el reconocimiento explícito de riesgo y su control y minimización.

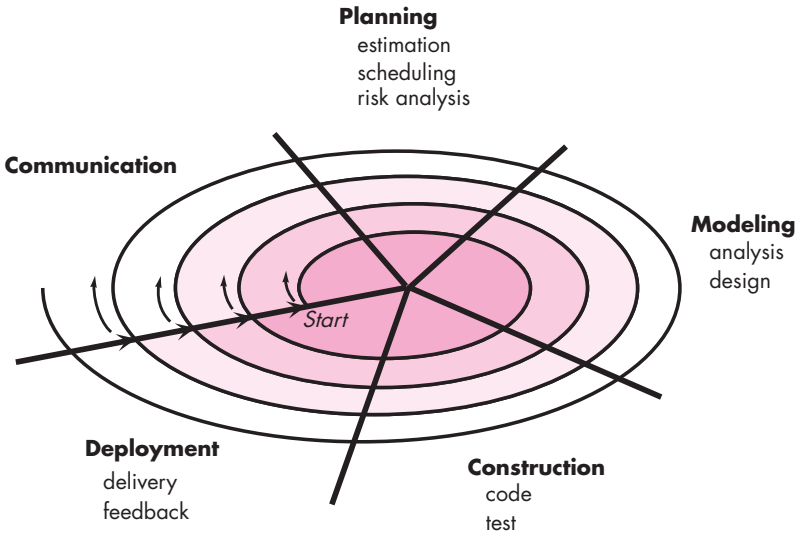

Figura 2.9: Esquema del modelo de ciclo de vida en espiral.

2.5.6. Concurrente

El modelo define una serie de eventos que desencadenarán transiciones de estado a estado para cada una de las actividades. Existen todas las actividades de ingeniería de software al mismo tiempo, pero residen en diferentes estados. Si no una actividad todavía no ha empezado estará en estado inactivo. Es aplicable a todos los tipos de desarrollo de software y proporciona una imagen precisa de la situación actual del proyecto.

2.5.7. Proceso Unificado

El Proceso Unificado (también llamado Proceso Unificado de Desarrollo Software) es un marco de desarrollo que se caracteriza por estar dirigido por casos de uso, centrado en la arquitectura y por ser iterativo e incremental. El refinamiento más conocido y documentado del Proceso Unificado es el Proceso Unificado de la empresa *Rational Software* (RUP), actualmente perteneciente a *IBM*.

El Proceso Unificado no es simplemente un proceso, sino un marco de trabajo extensible que puede ser adaptado a organizaciones o proyectos específicos. El primer libro sobre el tema fue publicado en 1999 por *Ivar Jacobson*, *Grady Booch* y *James Rumbaugh*, conocidos también por ser los desarrolladores del UML. De alguna manera el Proceso Unificado es un intento de aprovechar las mejores características de los modelos tradicionales (previamente descritos) incorporando algunos principios del desarrollo ágil de software. En él reconoce la importancia del cliente y su interacción utilizando métodos simples para describir la perspectiva que tiene el cliente sobre el sistema (casos de uso). Tiene un flujo de proceso

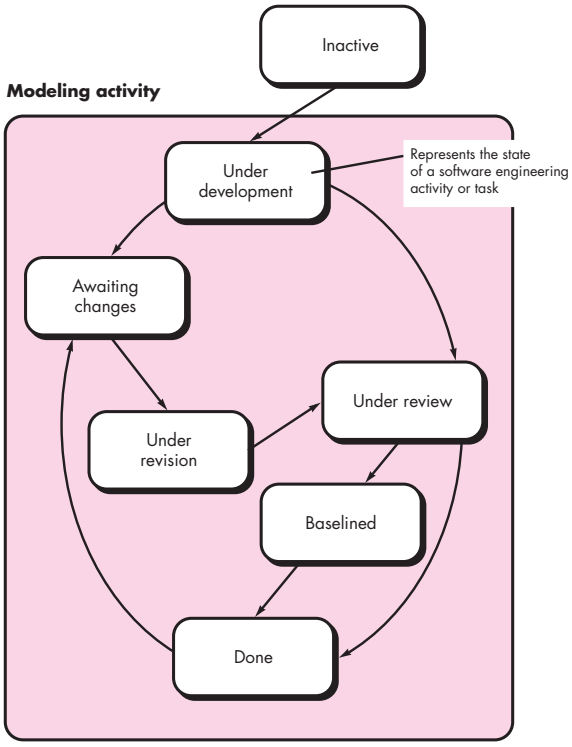

Figura 2.10: Esquema del modelo de ciclo de vida concurrente.

que es iterativo e incremental, proporcionando la sensación de evolución que es esencial en el desarrollo de software moderno. El Proceso Unificado puede ser dividido en 4 fases.

1. **Inicio**. Se analiza el alcance inicial del proyecto, una arquitectura para el sistema, obtener recursos y conseguir la aceptación por parte de lo involucrados.

2. **Elaboración**. Se prueba la arquitectura del sistema. Se seleccionan los casos de uso que permiten definir la arquitectura base del sistema y que se desarrollarán en esta fase. Se realiza la especificación de los casos de uso seleccionados y el primer análisis del problema. Se diseña la solución preliminar.

3. **Construcción**. Se desarrolla software operativo de manera incremental que cumpla con las necesidades más prioritarias de los involucrados en el proyecto. Para ello se deben clarificar los requisitos pendientes y administrar los cambios de acuerdo a las evaluaciones realizados por los usuarios.

4. **Transición**. Se valida y despliega el sistema en el entorno de producción. Se subsanan los errores y defectos encontrados en las pruebas de aceptación, se capacitan a los usuarios y se provee del soporte técnico necesario. Una vez finalizada esta fase, se debe comenzar a pensar en futuras novedades para el sistema.

Ver la figura 2.11 para comprender como se relacionan dichas fases con las cinco etapas que se decía al principio de este apartado que todo modelo de ciclo de vida tenía de una manera más explícita o implícita.

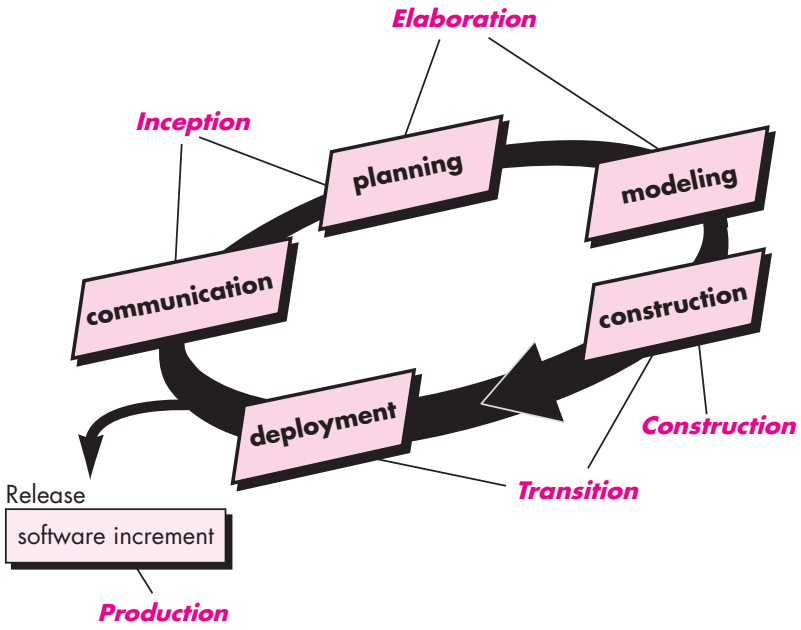

Figura 2.11: Esquema del modelo de ciclo de vida proceso unificado.

Capítulo Tercero

Metodologías tradicionales

3.1. Introducción

Este capítulo trata sobre metodologías de gestión de proyectos que se podrían denominar tradicionales por tener una filosofía alejada de las ágiles. Se estudiarán 5 metodologías: PMBOK (y por similitud ISO 21500), ICB, PRINCE2 y SWEBOK. Para cada una de ellas se hará una introducción hablando de la organización que la creó y su evolución histórica. Después se resumirá la estructura de cada metodología. Finalmente se reseñan muy brevemente otras 5 metodologías: 3 públicas (SSADM, MERISE Y MÉTRICA) y 2 privadas (APMBOK y P2M).

3.2. PMBOK / ISO 21500

3.2.1. Introducción

La guía Project Management Body Of Knowledge (PMBOK) es el estándar de gestión de proyectos del Project Management Institute (PMI) y es el único estándar acreditado por la American National Standards Institute (ANSI) (organismo para la coordinación y el uso de estándares de EEUU). Proporciona un marco común para los gestores de proyectos, suministrando un léxico y unos procedimientos estructurados aplicables a cualquier tipo de proyecto. PMI se fundó en 1960 y es una organización internacional sin ánimo de lucro que asocia a profesionales relacionados con la gestión de proyectos. A finales de 1970 casi 2000 miembros formaban parte de la organización y en los 80 se realizó la primera evaluación para la certificación como profesional en gestión de proyectos, llamado Project Management Professional (PMP). Desde 2011 es la más grande del mundo por estar integrada por más de 700.000 miembros de cerca de 170 países.

En 1996 se publicó la primera edición de PMBOK y actualmente está en su quinta

edición publicada en 2013 [1]. Además tiene tres extensiones: *Construction Extension*, *Government Extension* y *Software Extension to the PMBOK Guide* [2]. Ésta última es la más reciente y que ha permitido a PMBOK adaptarse a los proyectos software y que se ha tenido presente para realizar este TFM aunque no aporta mejoras significativas respecto a PMBOK.

En la actualidad PMI tiene ocho certificaciones: Project Management Professional, Certified Associate in Project Management, Program Management Professional, Portfolio Management Professional, PMI Agile Certified Practitioner, PMI Professional in Business Analysis, PMI Risk Management Professional y PMI Scheduling Professional.

En este apartado se ha agrupado el PMBOK con la norma internacional UNE-ISO 21500 *Directrices para la dirección y gestión de proyectos* [6] por su gran similitud de éste último con el primero. Por ello, la estructura del PMBOK será extensible a la del ISO 21500. Sus pequeñas diferencias se mencionarán en el último subapartado.

La ISO 21500 es un estándar internacional desarrollado por la International Organization for Standardization (ISO) a partir de 2007 y publicado en 2012. Proporciona una orientación genérica, explica los principios básicos y lo que constituye unas buenas prácticas en la gestión de proyectos. Es un documento de orientación que no está destinado a ser utilizado con fines de certificación. ISO tiene previsto que esta norma sea la primero de una familia de normas de gestión de proyectos y la diseño también para alinearse con otros estándares relacionados, tales como la ISO 10006:2003, ISO 10007:2003 e ISO 31000:2009.

3.2.2. Estructura

PMBOK e ISO 21500 es una metodología basada en procesos. Esto es, describir el trabajo por paquetes que se llevan a cabo en procesos. Este enfoque es similar a otras normas de gestión, tales como ISO 9000 o Capability Maturity Model Integration (CMMI) del Software Engineering Institute (SEI). Los procesos se superponen e interactúan a lo largo de un proyecto en sus diversas fases. Los procesos se describen en términos de:

- Entradas: documentos, planos, diseños, etc...
- Herramientas y técnicas: son las transformaciones aplicadas a las entradas que producirán las salidas.
- Salidas (idem a las entradas): documentos, planos, diseños, etc...

La guía establece 47 procesos que se dividen en 5 grupos de procesos básicos y 10 áreas de conocimiento (típicas en la mayoría de los proyectos). Las definiciones de los 5 grupos de procesos se explican a continuación y sus interacciones se muestran en la figura 3.1

1. **Inicio**. Aquellos procesos realizados para definir un nuevo proyecto o una nueva fase de un proyecto existente.
2. **Planificación**. Aquellos procesos requeridos para establecer el alcance del proyecto, refinar los objetivos y definir la estrategia para alcanzar los objetivos que se marcaron al comienzo del proyecto.
3. **Ejecución**. Aquellos procesos realizados para completar el trabajo definido en el plan del proyecto y así satisfacer sus especificaciones.
4. **Monitorización y Control**. Aquellos procesos requeridos para realizar el seguimiento, revisar y controlar el progreso y la eficiencia del proyecto. Se identifican las áreas en las que los cambios en el plan son necesarios y se inician los cambios correspondientes.
5. **Cierre**. Aquellos procesos realizados para finalizar formalmente todas las actividades de todos los procesos.

Las 10 áreas de conocimiento son las siguientes (ver la tabla 3.1 para más detalles).

- **Gestión de la Integración**. Se incluyen todas las actividades y procesos que hay que realizar para identificar, combinar y coordinar los diversos procesos y actividades de gestión dentro de los grupos de gestión de procesos.
- **Gestión del Alcance**. Se incluyen los procesos para asegurar que el proyecto tiene todo el trabajo necesario y sólo el necesario, para completar el proyecto de forma satisfactoria.
- **Gestión de Plazos**. Se incluyen los procesos requeridos para finalizar el proyecto de forma satisfactoria en el plazo previsto.
- **Gestión de Costes**. Se incluyen los procesos necesarios para poder planificar, estimar, presupuestar y controlar los costes de forma que se pueda finalizar dentro de los costes planificados.
- **Gestión de la Calidad**. Se determinan las políticas de calidad, objetivos y responsabilidades de forma que el proyecto satisfaga las necesidades previstas.
- **Gestión de los RRHH**. Se encarga de organizar y gestionar al equipo de proyecto, asignando los roles y responsabilidades correspondientes.
- **Gestión de la Comunicación**. Se asegura la generación temporal apropiada y la distribución, colección y almacenamiento de la información del proyecto.
- **Gestión del Riesgo**. Son los procesos que realizan la planificación, identificación, análisis cualitativo y cuantitativo de los riesgos, así como la planificación de las medidas a adoptar y su control.

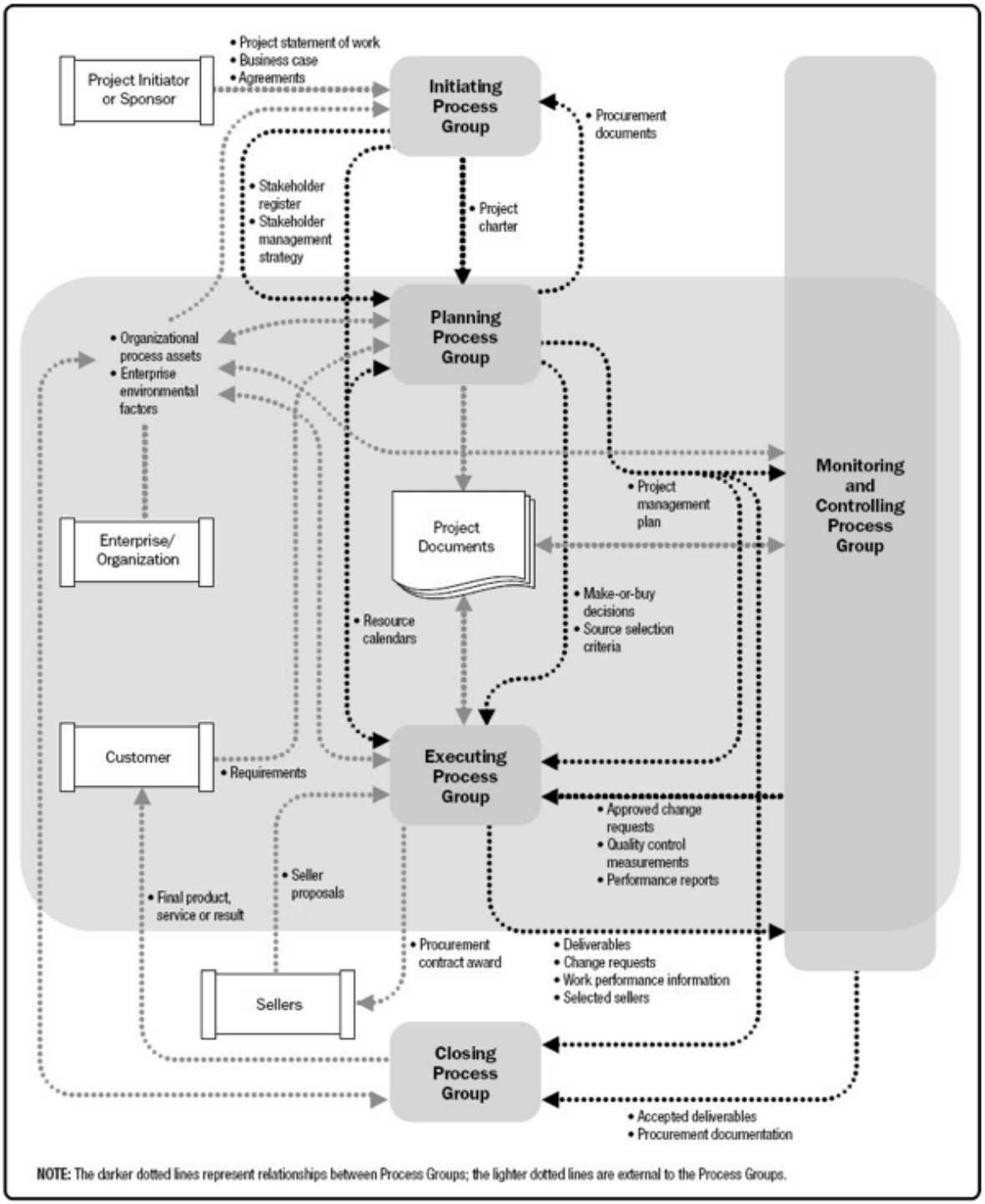

Figura 3.1: Interacciones entre los grupos de procesos de PMBOK.

- **Gestión de Aprovisionamiento**. Son los procesos que incluyen la adquisición de productos, servicios o resultados necesarios y que siendo ajenos al equipo del proyecto son necesarios para el trabajo a realizar.
- **Gestión de Involucrados**. Se incluye los procesos requeridos para identificar a todas las personas u organizaciones afectadas por el proyecto, analizando las expectativas y el impacto de las partes interesadas en el proyecto y el desarrollo de estrategias de gestión adecuadas para la participación efectiva de los interesados en las decisiones y la ejecución de proyectos.

3.2.3. Diferencias entre PMBOK e ISO 21500

El conjunto de 10 áreas de conocimiento de PMBOK tiene su correspondencia exacta con la norma ISO 21500. Hay 39 procesos en la ISO 21500 y 47 procesos PMBOK. 33 procesos de ISO 21500 tienen sus equivalentes directos en PMBOK. Uno de ellos ha cambiado su lugar dentro de los grupos de procesos. 4 procesos se han movido entre áreas de conocimiento. 3 pares de procesos de PMBOK se han fusionado en 4 procesos individuales de la ISO 21500.

8 procesos del PMBOK están ausentes en la ISO 21500: 5.1 Plan de gestión del alcance, 5.5 Validar alcance, 6.1 Plan de gestión del calendario, 7.1 Plan de gestión de costes, 9.1 Plan de gestión de RRHH, 12.1 Plan de gestión de riesgos, 13.2 Plan de gestión de los involucrados y 13.4 Control del compromiso de los involucrados.

3 nuevos procesos se han introducido en la ISO 21500: 4.3.8 Recoger las lecciones aprendidas, 4.3.17 Definir la organización del proyecto y 4.3.19 Control de recursos.

3.3. ICB

3.3.1. Introducción

IPMA Competence Baseline (ICB) es el estándar de International Project Management Association (IPMA) para la competencia en la dirección de proyectos. IPMA se creó en 1965 en Suiza siendo la organización más antigua de gestión de proyectos. Está formada por una red de asociaciones nacionales. Es decir, se constituye como la organización representativa de todas las asociaciones nacionales instaladas en cada país, que orientan sus servicios a las necesidades nacionales de desarrollo en el área de gestión de proyectos y en su misma lengua. La asociación española se llama Asociación Española de Ingeniería de Proyectos (AEIPRO) que es otra organización sin ánimo de lucro creada en 1992. También tiene suscrito un convenio de colaboración con el PMI.

Knowledge Areas	Project Management Process Groups				
	Initiating Process Group	**Planning Process Group**	**Executing Process Group**	**Monitoring and Controlling Process Group**	**Closing Process Group**
4. Project Integration Management	4.1 Develop Project Charter	4.2 Develop Project Management Plan	4.3 Direct and Manage Project Work	4.4 Monitor and Control Project Work 4.5 Perform Integrated Change Control	4.6 Close Project or Phase
5. Project Scope Management		5.1 Plan Scope Management 5.2 Collect Requirements 5.3 Define Scope 5.4 Create WBS		5.5 Validate Scope 5.6 Control Scope	
6. Project Time Management		6.1 Plan Schedule Management 6.2 Define Activities 6.3 Sequence Activities 6.4 Estimate Activity Resources 6.5 Estimate Activity Durations 6.6 Develop Schedule		6.7 Control Schedule	
7. Project Cost Management		7.1 Plan Cost Management 7.2 Estimate Costs 7.3 Determine Budget		7.4 Control Costs	
8. Project Quality Management		8.1 Plan Quality Management	8.2 Perform Quality Assurance	8.3 Control Quality	
9. Project Human Resource Management		9.1 Plan Human Resource Management	9.2 Acquire Project Team 9.3 Develop Project Team 9.4 Manage Project Team		
10. Project Communications Management		10.1 Plan Communications Management	10.2 Manage Communications	10.3 Control Communications	
11. Project Risk Management		11.1 Plan Risk Management 11.2 Identify Risks 11.3 Perform Qualitative Risk Analysis 11.4 Perform Quantitative Risk Analysis 11.5 Plan Risk Responses		11.6 Control Risks	
12. Project Procurement Management		12.1 Plan Procurement Management	12.2 Conduct Procurements	12.3 Control Procurements	12.4 Close Procurements
13. Project Stakeholder Management	13.1 Identify Stakeholders	13.2 Plan Stakeholder Management	13.3 Manage Stakeholder Engagement	13.4 Control Stakeholder Engagement	

Tabla 3.1: Procesos de PMBOK.

IPMA comenzó con una versión inicial de ICB en 1995, definiendo y validando la competencia de los directores de proyectos. En 1999 se publicó la versión ICB 2.0 y en 2006 se renovó con ICB 3.0 [3] que sigue vigente en la actualidad.

IPMA tiene para individuos 4 niveles de certificación, de menos a más avanzado: técnico en dirección de proyectos (IPMA nivel D), profesional en dirección de proyectos (IPMA nivel C), director de proyecto (IPMA nivel B) y director de cartera de proyectos (IPMA nivel A).

3.3.2. Estructura

ICB contiene los términos básicos, tareas, habilidades, funciones, procesos, métodos, técnicas y herramientas que se deben usar, tanto teórica como prácticamente, para una buena gestión de proyectos. El objetivo principal de ICB es estandarizar y reducir las tareas fundamentales necesarias para completar un proyecto de la forma más efectiva y eficiente.

La estructura de ICB es bastante distinta a las demás metodologías porque se organiza por competencias, no por procesos. Los 46 elementos de competencia se agrupan en 3 grandes grupos (ver figura 3.2).

- **Competencias técnicas**. Abarcan el cumplimiento de los requisitos del proyecto (o programa o portfolio) según los involucrados, la integración de los trabajos de la organización (sea en un proyecto temporal, programa o portfolio) y la producción de entregables simples en la organización del proyecto. Las competencias son 20:

  Éxito en la dirección de proyectos; Partes involucradas; Requisitos y objetivos de proyectos; Riesgos y oportunidades; Calidad; Organizaciones de proyectos; Trabajo en equipo; Resolución de problemas; Estructuras de proyectos; Alcance y entregables; Tiempo y fases de los proyectos; Recursos; Coste y financiación; Aprovisionamiento y contratos; Cambios; Controles e informes; Información y documentación; Comunicación; Arranque; Cierre.

- **Competencias de comportamiento**. Abarcan las competencias relacionadas directamente con el propio director del proyecto, las derivados de la gestión del director, las comunes y contextuales utilizados en la gestión global del proyecto y sus involucrados y las competencias que tienen sus orígenes en la economía, la sociedad, la cultura y la historia. Las competencias son 15:

  Implicación; Autocontrol; Asertividad; Relajación; Accesibilidad; Creatividad; Liderazgo; Orientación al resultado; Eficiencia; Consultable; Negociación; Crisis y conflictos; Credibilidad; Apreciación de valores; Ética.

- **Competencias contextuales**. Se agrupan en términos del rol de la gestión de proyectos en las organizaciones permanentes y de las interrelaciones de gestión de proyectos con la administración de negocios. Las competencias son 11:

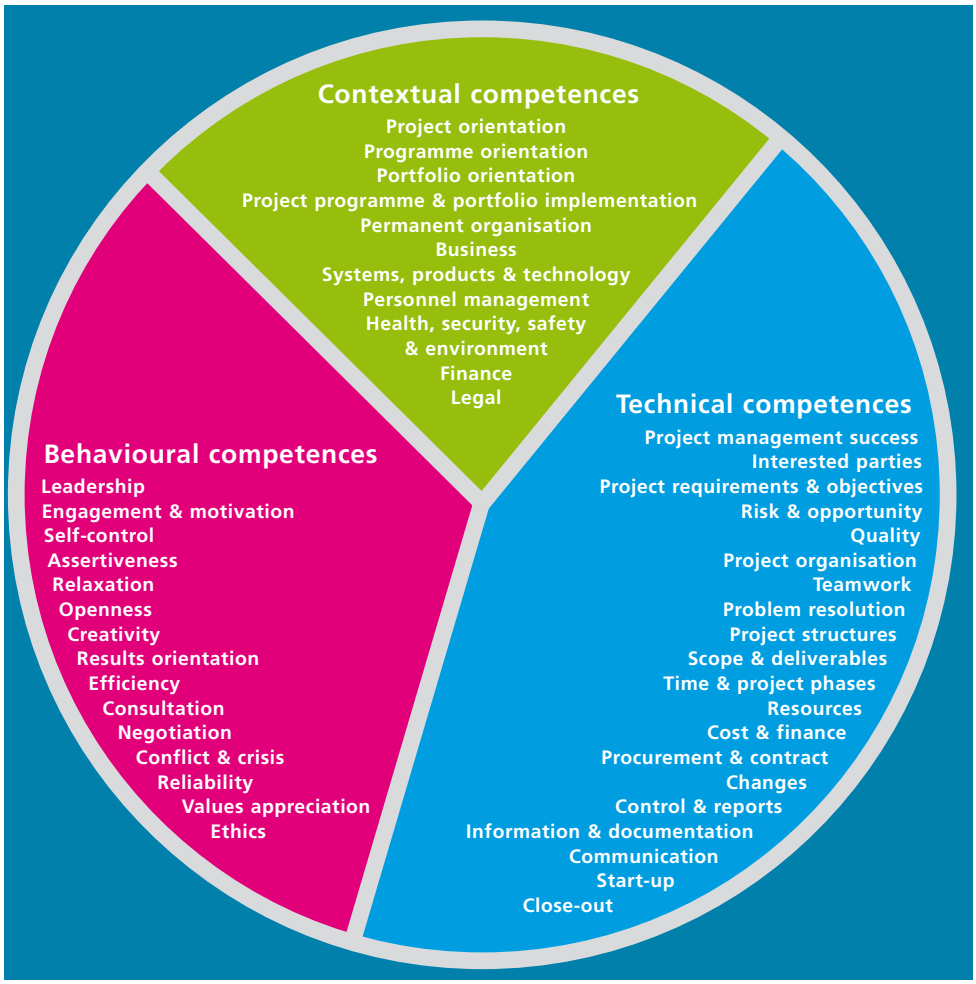

Figura 3.2: Competencias para la gestión de proyectos de ICB.

Orientación al proyecto; Orientación al programa; Orientación al portfolio; Implementación de proyecto, programa y porftolio; Legalidad; Negocio; Organización permanente; Sistemas, productos y tecnología; Gestión de personal; Salud, seguridad, prevención y entorno; Financiación.

3.4. PRINCE2

3.4.1. Introducción

La metodología Projects in a Controlled Environment (PRINCE2) deriva del método de gestión de proyectos PRINCE, que fue desarrollado inicialmente en 1989 por la Central Computer and Telecommunications Agency (CCTA) como un estándar del gobierno

del Reino Unido para los sistemas de IT de gestión de proyectos. A su vez, PRINCE proviene de la metodología Project Resource Organisation Management Planning Technique (PROMPT) creada a mediados de los 70 por la empresa Simpact Systems Limited que proporciona un marco adecuado para gestionar la estrategia, viabilidad, desarrollo y apoyo de los sistemas de IT a través de un enfoque estructurado de gestión de proyectos. Por tanto, esta metodología tiene una especial significación en este TFM por sus orígenes ligados al sector Tecnologías de la Información y Comunicación (TIC).

PRINCE es un acrónimo de "proyectos en ambientes controlados" pero pronto llegó a ser aplicado regularmente fuera del entorno TIC, tanto en el gobierno del Reino Unido como en el sector privado. PRINCE2 fue lanzado en 1996 como un método genérico de gestión de proyecto alcanzando cotas de bastante popularidad por ser un estándar de facto para la gestión de proyectos en varios países (sobre todo de la Commonwealth), empresas multinacionales y organizaciones internacionales.

En 2009 se publicó una revisión "PRINCE2: 2009 Refresh"manteniendo el nombre (en lugar de PRINCE3 o similar) para indicar que el método sigue siendo fiel a sus principios. Sin embargo, se trata de una revisión importante de la versión de 1996 para adaptarla al entorno empresarial cambiante transformándolo en una metodología más simple y ligera. PRINCE2 esta perfectamente alineada con la norma ISO 21500.

En 2013 la propiedad de los derechos de PRINCE2 se transfirió del organismo público Británico HM Cabinet Office a Axelos Ltd, una empresa conjunta de HM Cabinet Office y la empresa privada Capita plc. Hay 3 niveles de certificación, de menos a más avanzada: Foundation Examination, Practitioner Examination y Professional Examination.

3.4.2. Estructura

La metodología PRINCE2 se apoya en 7 principios, enriqueciendo no sólo al proyecto en concreto, sino a toda la organización en la que se desarrolla.

- **Justificación comercial continua**. Se asegura que hay un motivo justificable para iniciar el proyecto y que perdura durante toda su vida.
- **Aprender de la experiencia**. Se recogen experiencias anteriores, las que se van obteniendo durante el proyecto y las lecciones aprendidas a su cierre.
- **Roles y Responsabilidades definidos**. Asegurando que los involucrados del proyecto están representados en la toma de decisiones.
- **Gestión por Fases**. El proyecto se planifica, supervisa y controla fase a fase.
- **Gestión por excepción**. Delegar la autoridad de gestión al subordinado, dándole autonomía según unas tolerancias pautadas (de plazos, coste, calidad, alcance, be-

neficio y/o riesgo) de manera que si se sobrepasa la tolerancia, se consulte al superior como actuar.

- **Orientación a productos**. Concentrarse en la definición y entrega de productos. Es decir, un proyecto no son un conjunto de tareas, sino que se entregan productos (que se elaboran tras la ejecución de las tareas necesarias).
- **Adaptación**. Se asegura que la metodología y los controles a aplicar se basan en el tamaño, complejidad, importancia, capacidad y nivel de riesgo del proyecto.

Hay 8 grupos de procesos que se resumen a continuación y sus interacciones se esquematizan en la figura 3.3.

1. **Emprender un proyecto (SU)**. Se realiza al comienzo del proyecto siendo una preparación inicial para la gestión del proyecto, su control y viabilidad. Este proceso lo crea la Junta del proyecto garantizando los recursos necesarios.
2. **Dirigir Proyecto (DP)**. Dirige y define las responsabilidades de la Junta en la supervisión del proyecto. Está por encima de todos los procesos, interactuando con el resto. Proporciona los protocolos de aprobación al final de cada etapa y al cierre del proyecto. Este proceso es el marco de suministro de entradas, de recepción de requisitos y toma de decisiones. Es el único proceso en el que actúa la Junta porque el resto de procesos son llevados por el Director del proyecto y su equipo.
3. **Iniciar proyecto (IP)**. Sólo se realiza una vez durante el ciclo de vida del proyecto. Sirve para plantear cómo se puede gestionar la totalidad del proyecto y se plasma en el documento de inicio del proyecto (Project Initiation Document, PID). El objetivo del documento es el establecimiento de un entendimiento común de los elementos críticos del proyecto, así como el acuerdo de la Junta para la primera etapa de desarrollo del proyecto.
4. **Planificación (PL)**. Proceso común para el resto de procesos. Los planes se producen identificando los entregables del proyecto, las actividades y recursos necesarios para crearlos. Todo ello tiene en una relación consistente con los requerimientos identificados en el PID.
5. **Control de etapa (CS)**. Provee una guía para la gestión diaria del proyecto. Incluye la autorización y recepción de trabajos, gestión del cambio y de versiones, análisis e informes, consideraciones de viabilidad, acciones correctivas y escalado de incidencias a la Junta. Este proceso de control se realiza de forma iterativa por cada etapa de desarrollo.
6. **Gestión de entrega del producto (MP)**. Forma parte del sistema de autorización siendo el mecanismo que sirve para que los ejecutores del trabajo técnico acuerden los trabajos a realizar. Se repite por cada paquete de trabajo autorizado.

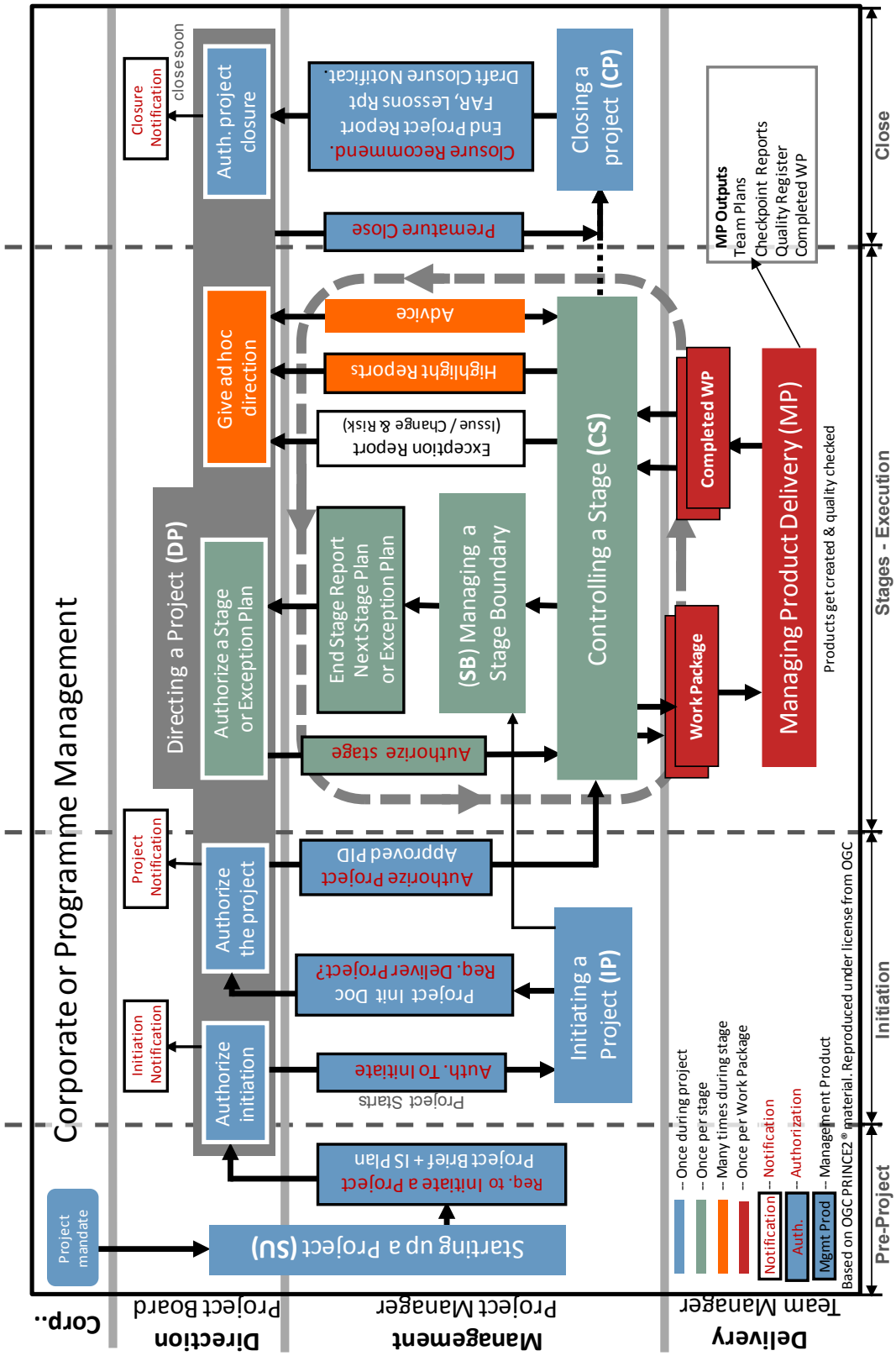

Figura 3.3: Interacciones entre los grupos de procesos de PRINCE2.

7. **Gestión de los límites de etapa (SB)**. Realizar la transición de un estado finalizado al inicio del siguiente estado, garantizando que el trabajo finalizado se ha realizado de acuerdo con los requisitos establecidos.

8. **Cerrar un proyecto (CP)**. Proceso de transición de entrega del proyecto a la organización. Puede finalizar por haber realizado el trabajo satisfactoriamente o por terminación prematura, guardando las lecciones aprendidas. El proceso permite garantizar que si el cierre es por finalización del trabajo, ésta satisface las necesidades del cliente.

Por último, hay 8 áreas de gestión de proyectos que deben abordarse de forma continua durante todo el proyecto.

- **Caso de negocio**. Desarrollar una idea en una propuesta viable para la organización. La gestión del proyecto mantiene la atención en los objetivos de la organización y en los beneficios durante todo el proyecto. Es decir, responde a ¿por qué?

- **Organización**. Describe las funciones y responsabilidades del equipo (temporal) necesario para gestionar el proyecto con eficacia. Es decir, responde a ¿quién?

- **Calidad**. Los requerimientos de calidad de los entregables se basan en las descripciones del producto que a su vez son preparados por el Director del proyecto y aprobados por la Junta. Es decir, responde a ¿qué?

- **Planes**. Describen los pasos necesarios para entregar los productos, las técnicas que se deben aplicar y la comunicación a lo largo del proyecto. Es decir, responde a ¿cómo, cuánto y cuándo?

- **Controles**. El objetivo es garantizar que el proyecto genera los productos necesarios definidos en los criterios de aceptación y que cumple la planificación con los recursos y costes estimados. Además, debe garantizar la viabilidad del proyecto. Es decir, responde a ¿se ajusta a lo planificado?

- **Riesgos**. Aborda la forma de gestionar las incertidumbres en los planes y, en general, en el entorno del proyecto. Es decir, responde a ¿qué sucede si...?

- **Cambio**. El control de los cambios del alcance calcula el impacto de los potenciales cambios, su importancia, costes, impacto en el negocio y decidir si se incluyen o no. Es decir, responde a ¿cuál es el impacto?

- **Progreso**. La principal condición de control de un proyecto PRINCE2 es la existencia de un caso de negocio viable. Explica el proceso de toma de decisiones para la aprobación de planes y supervisar el rendimiento determinando si el proyecto debe continuar y cómo. Es decir, responde a ¿dónde estamos ahora?, ¿a dónde vamos? y ¿debemos continuar?

3.5. SWEBOK

3.5.1. Introducción

El Software Engineering Body of Knowledge (SWEBOK) ha sido promovido y realizado por el Institute of Electrical and Electronics Engineers (IEEE) Computer Society siendo publicado su última versión (V3) en 2014 [5]. Es un estándar internacional, ISO/IEC TR 19759:2005. Los editores de SWEBOK recibieron y contestaron a comentarios de aproximadamente 150 colaboradores de 33 países. La versión 2 data de 2004 y la versión 1 de 2001. Los hitos más relevantes del IEEE Computer Society relacionados para lograr realizar el SWEBOK se resumen en este listado cronológico.

1976. Se funda el comité por la IEEE Computer Society para el desarrollo de normas de Ing. de Software.

1979. Se publica el estándar IEEE 730 para el aseguramiento de la calidad. Es la primera visión integral de la disciplina en emerger del IEEE Computer Society siendo influyente para posteriores normas de gestión de la configuración, requisitos software, diseño software, testeo software, etc...

1986. Se publica el IEEE Std. 1002, *Taxonomy of Software Engineering Standards* que proporcionó una nueva visión holística de la disciplina describiendo la forma y el contenido de la taxonomía de estándares de Ing. de Software.

1993. El IEEE Computer Society en conjunto con la Association for Computing Machinery (ACM) crearon un comité para "Establecer los conjuntos necesarios de criterios y normas para la práctica profesional de la Ing. de Software en los que las decisiones industriales, la certificación profesional y los programas de estudio puedan estar basado." Este fue el comité encargado de trabajar hasta que se liberó SWEBOK V1 en 2001.

1996. Se publica el ISO/IEC 12207, *Standard for Software Life Cycle Processes* proporcionado un importante punto de partida para el conjunto de conocimientos capturados en el SWEBOK.

3.5.2. Estructura

El SWEBOK, como su propio nombre indica, es un cuerpo del conocimiento[1] de la Ing. de Software. Por tanto, no es sólo una metodología de gestión de proyectos en general

[1]Es el conjunto completo de conceptos, términos y actividades que conforman un dominio profesional definido por la sociedad científica pertinente o asociación profesional.

como las de los apartados anteriores sino que abarca 15 áreas de conocimiento que influyen en esta disciplina.

- **Requisitos del Software**. Se ocupa de la obtención, análisis, especificación y validación de requisitos del software, así como la gestión de los requisitos durante todo el ciclo de vida del producto de software. Los requisitos expresan las necesidades y limitaciones impuestas a un producto software que contribuye a la solución de algún problema en el mundo real.

- **Diseño del Software**. Cubre el proceso de diseño y el producto resultante. El proceso de diseño es la actividad del ciclo de vida de Ing. de Software en el que se analizan los requisitos de software con el fin de producir una descripción de la estructura interna del software y sus funcionalidades, que servirán como base para su construcción. Un diseño describe la arquitectura de software, es decir, cómo el software se descompone y es organizado en componentes así como las interfaces entre esos componentes con un nivel de detalle que permita su construcción.

- **Construcción (desarrollo) del Software**. Se refiere a la creación del software a través de una combinación de diseño detallado, codificación, pruebas unitarias, pruebas de integración, depuración y verificación. Se estudian los fundamentos del desarrollo de software, la gestión del desarrollo, tecnologías de construcción, consideraciones prácticas y herramientas de desarrollo.

- **Pruebas del Software**. Es una actividad realizada para evaluar la calidad del producto y mejorarlo mediante la identificación de defectos. Las pruebas implican la verificación dinámica del comportamiento de un programa contra el comportamiento que se espera de un conjunto finito de casos de prueba. Se estudian los fundamentos de las pruebas de software, técnicas, pruebas de interfaz de usuario, medidas relacionadas con las pruebas y consideraciones prácticas.

- **Mantenimiento del Software**. Implica mejorar las capacidades existentes, adaptando el software para funcionar en entornos operativos nuevos y modificados, así como la corrección de defectos. Se estudian los fundamentos de mantenimiento del software, cuestiones clave (técnicas, gestión, estimación de costes, métricas de mantenimiento), el proceso de mantenimiento, técnicas (reingeniería, ingeniería inversa, refactorización, retirada de software en producción), técnicas de recuperación ante desastres y las herramientas de mantenimiento.

- **Gestión de la configuración del Software**. Se encarga de la identificación de la configuración de un sistema en distintos puntos temporales para controlar sistemáticamente cambios en la configuración, así como mantener la integridad y la trazabilidad de la configuración en todo el ciclo de vida. Se estudia la gestión del proceso de la configuración, identificación del software de configuración, control, contabilidad de estados, auditoría, gestión de versiones y entregas de software y herramientas de gestión de configuración.

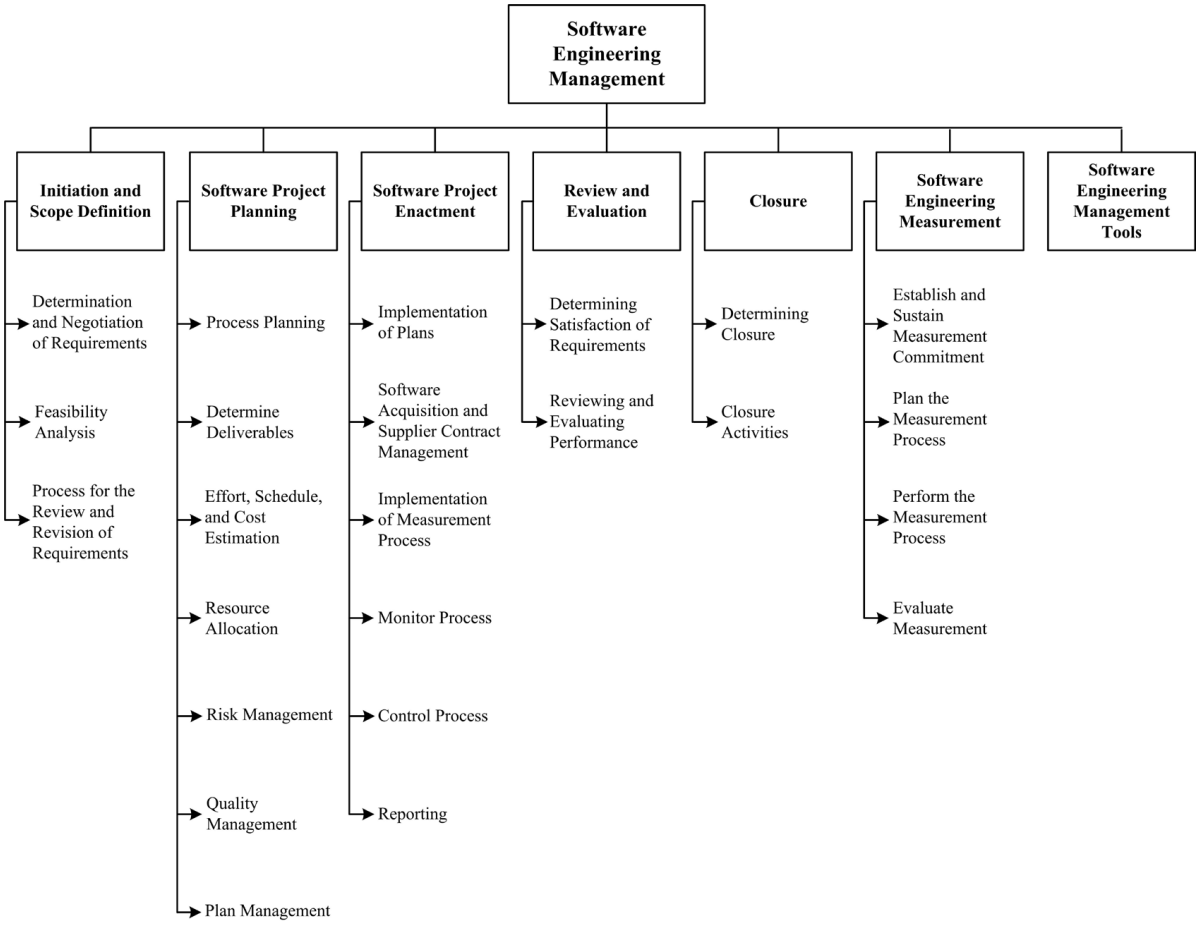

Figura 3.4: Procesos de la gestión de proyectos software de SWEBOK.

- **Gestión de la Ing. de Software**. Consiste en la planificación, coordinación, medición, presentación de entregables y el control de un proyecto para asegurar que el desarrollo y mantenimiento del software es sistemático, disciplinado y cuantificado. Se estudian la iniciación y el alcance del proyecto, su planificación, su ejecución, la aceptación del producto, la revisión y análisis de los resultados del proyecto, su cierre y las herramientas de gestión. En la figura 3.4 se muestra con más detalle el contenido de este área por ser la que interesa en este TFM.

- **Procesos de la Ing. de Software**. Se ocupa de la definición, implementación, evaluación, medición, gestión y mejora de los procesos del ciclo de vida. Los temas cubiertos son la implementación de procesos y el cambio (infraestructura y gestión de procesos, modelos de implementación de procesos y el cambio), definición de procesos (modelos y procesos del ciclo de vida, notaciones para la definición de procesos y la adaptación y automatización de procesos), proceso de modelos y métodos de evaluación, medición (de procesos, de productos, técnicas de medición y la calidad de los resultados) y herramientas de proceso de software.

- **Modelos y métodos de la Ing. de Software**. Imponen una estructura en la ingeniería con el objetivo de hacer que la actividad sea sistemática y repetible y, en última instancia, más orientada al éxito. Se estudia el modelado (principios y propiedades, sintaxis vs. semántica vs. invariantes, precondiciones, postcondiciones e invariantes), tipos de modelos (información y modelos de comportamiento estructural), análisis (de exactitud, integridad, consistencia, calidad, interacciones y trazabilidad) y los métodos de desarrollo (heurísticos, formales, prototipos y ágiles).

- **Calidad del Software**. Es una preocupación omnipresente abordándose en muchas áreas de esta guía. Se estudia los fundamentos (características y cultura de la calidad, el valor, el coste y la mejora), procesos de gestión de la calidad (software de garantía de calidad, verificación y validación, revisiones y auditorías) y consideraciones prácticas (caracterización de defectos, medición y herramientas de calidad).

- **Práctica Profesional de la Ing. de Software**. Es el conocimiento, habilidades y actitudes que los ing. de software deben poseer para ejercer su trabajo de una manera profesional, responsable y ética. Se estudia la profesionalidad (conducta profesional, asociaciones profesionales, normas, contratos de trabajo y asuntos legales), códigos de ética, dinámicas de grupo (trabajo en equipo, la complejidad del problema cognitivo, interactuar con los involucrados, gestionar la incertidumbre y la ambigüedad y también gestionar los entornos multiculturales) y habilidades de comunicación.

- **Economía de la Ing. de Software**. Tiene que ver con la toma de decisiones dentro del negocio para alinear las decisiones técnicas con los objetivos empresariales. Se estudia los fundamentos (propuestas, flujos de caja, valor temporal del dinero, horizontes de planificación, inflación y depreciación, decisiones de reemplazo y retiro), toma de decisiones sin ánimo de lucro (análisis de coste-beneficio y de optimización), estimación, el riesgo económico y la incertidumbre (técnicas de estimación, decisiones relativas al riesgo y la incertidumbre) y la toma de decisiones de atributos múltiples (escalas de medidas, técnicas compensatorias y no compensatorias).

- **Fundamentos de Computación**. Es la base de las ciencias de computación necesaria para la práctica de la disciplina. Se estudia las técnicas de resolución de problemas, abstracción, algoritmos y complejidad, fundamentos de programación, computación paralela y distribuida, fundamentos de computadores, sistemas operativos y comunicaciones de red.

- **Fundamentos Matemáticos**. Es la base matemática necesaria para la práctica de la disciplina. Se estudia los conjuntos, relaciones y funciones, la lógica proposicional y de predicados; técnicas de demostraciones, gráficos y árboles, probabilidad discreta, gramáticas y máquinas de estados finitos y teoría de números.

- **Fundamentos de la Ingeniería**. Es la base en ingeniería necesaria para la práctica de la disciplina. Se estudia los métodos empíricos y técnicas experimentales, el análisis

estadístico, mediciones y métricas, diseño de ingeniería, simulación y modelado y análisis de la causalidad.

En conclusión, es una excelente guía por tratar la disciplina de manera exhaustiva aunque en las áreas de conocimiento no entra muy en detalle explicando las técnicas y cómo habría que hacer las cosas (la Gestión de Ing. de Software no es una excepción). El detalle se deja en manos de la bibliografía que se anota para cada área.

3.6. Otras metodologías

Hay otras metodologías o cuerpos de conocimiento que no se van a tratar por no no ser tan relevantes a juicio del autor. Las razones para no incluirlas en el estudio son variadas: no son tan populares, son viejas sin marcar tendencias actuales, no tienen apenas seguimiento en el sector TIC, etc... No obstante, se hace una breve reseña de algunas de ellas principalmente porque no se ha tratado ninguna metodología que pertenezca a las Administraciones Públicas (AAPP), metodologías públicas se podrían denominar. Si el cliente son las AAPP es fundamental conocer su normativa de gestión de proyectos para realizar el proyecto ajustándose a ellas.

3.6.1. SSADM

Structured Systems Analysis and Design Method (SSADM) es una metodología pública de aproximación en cascada para el desarrollo de sistemas IT y que puede ser considerada como la más completa de las metodologías estructuradas, por su garantía contrastada a lo largo de los años por los desarrolladores. Se considera que representa el pináculo del enfoque riguroso en la documentación hacia el diseño del sistema, en contraposición a las metodologías ágiles.

Se lanzó la primera versión en 1981 y en 1983 se convirtió en uso obligatorio para el desarrollo de todos los proyectos nuevos del gobierno del Reino Unido. En 1988 fue promocionada como un estándar abierto. En el 2000, la CCTA renombró SSADM como "Business System Development" siendo reorganizada en 15 módulos y añadiendo otros 6.

3.6.2. MERISE

MERISE es una metodología pública de la administración francesa, creada por iniciativa de su Ministerio de Industria. Es un método integrado de análisis, concepción y gestión de proyectos. Provee un marco metodológico y un lenguaje común riguroso para los desarrollos informáticos. Comienza a desarrollarse en 1972, concluyendo la primera versión a finales de

1976. Se utilizó masivamente a finales de la década de 1970 y 1980 para la informatización masiva de las organizaciones.

3.6.3. MÉTRICA

MÉTRICA es una metodología pública Española, propuesta por el Ministerio de Administraciones Públicas, para la sistematización de las actividades que dan soporte al ciclo de vida del software. Supone un conjunto de normas, técnicas y documentos para el desarrollo del software de diversa complejidad, tamaño y ámbito. Se ha ido adaptando a la evolución de las tecnologías que han ido surgiendo publicándose varias versiones: MÉTRICA V1 (1989), MÉTRICA V2 (1993) y MÉTRICA V3 (2001).

MÉTRICA está basada en el modelo de procesos del ciclo de vida de desarrollo ISO/IEC 12207 *Information Technology - Software Life Cycle Processes* y en la norma ISO/IEC 15504 *Software Process Improvement And Assurance Standards Capability Determination*.

3.6.4. APMBOK

Association for Project Management Body Of Knowledge (APMBOK) ha sido desarrollado por Association for Project Management (APM), asociación británica sin ánimo de lucro. APM es el organismo de certificación en el Reino Unido de IPMA. El cuerpo de conocimientos APM tratan en toda su extensión la profesión de gestor de proyecto, programa y portfolio. La sexta revisión se ha publicado en 2012 y cuenta con un total de 69 temas divididos entre cuatro bloques: entorno, RRHH, entregas e interfaces.

3.6.5. P2M

La Project Management Association of Japan (PMAJ) es una organización japonesa sin ánimo de lucro que cuenta con su propia metodología en Dirección de Proyectos, llamada Project & Program Management for Enterprise Innovation (P2M) y publicada en 2001. Esta asociación cuenta con 3 niveles de certificación, de menos a más avanzada: Project Management Specialist, Project Manager Registered y Program Management Architect.

P2M proporciona directrices para la innovación empresarial a través del programa y gestión de proyectos. Es decir, se enfoca en la organización y en el programa por lo que está alineado con la gestión de múltiples proyectos y reconoce que un proyecto puede afectar al programa de la organización. Las áreas de conocimiento de la metodología se pueden ver en la figura 3.5.

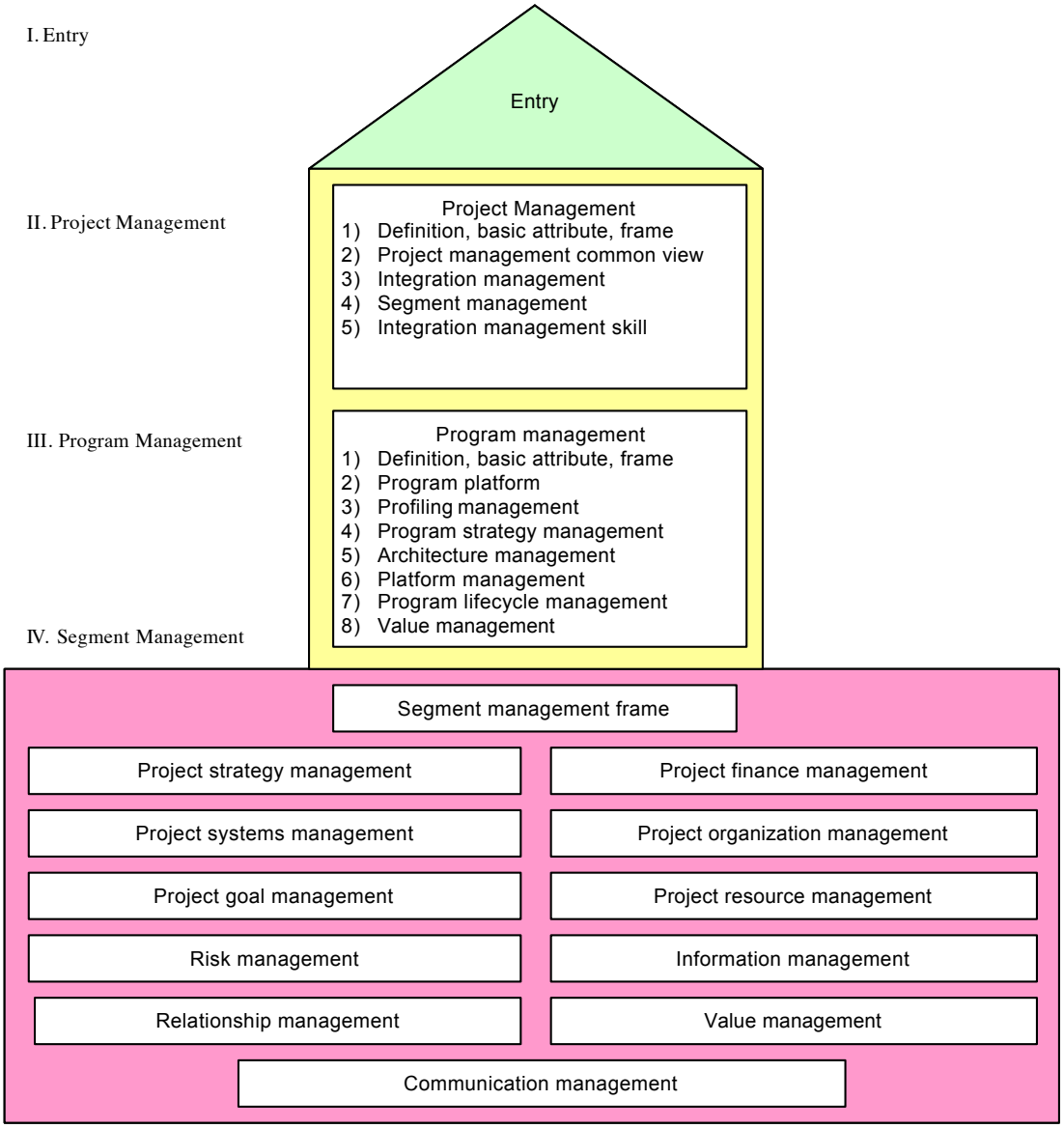

Figura 3.5: Áreas de conocimiento de la metodología P2M.

Capítulo Cuarto

Metodologías Ágiles

4.1. Introducción

El desarrollo ágil de software comprende a los métodos de Ing. de Software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan mediante la colaboración de todos los involucrados en el proyecto. Aunque en muchas ocasiones se considere algo novedoso o revolucionario, es conveniente recordar que el veterano ciclo de vida iterativo e incremental es incluso más antiguo que el ciclo de vida en cascada, empezando a aplicarse al software en la década de 1960. Existen muchos métodos de desarrollo ágil, la mayoría minimizando los riesgos desarrollando software en lapsos cortos (iteraciones).

Este tipo de ciclo de vida es muy recomendable en los proyectos software porque se considera que los cambios de requisitos a medida que avanza un proyecto es un aspecto natural, inevitable e incluso deseable del desarrollo software. Ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

Lo que aquí se llama metodologías ágiles es, en realidad, un conjunto heterogéneo de métodos con más o menos reglas, principios, recomendaciones, buenas prácticas, etc... A esta agrupación se le suele llamar el "paraguas ágil" porque englobla la filosofía que tienen como bases estos métodos. Si hubiese que caracterizar a esta filosofía común, se podría resumir en cuatro términos.

- **Incremental**. Las versiones de software son pequeñas con ciclos de desarrollo rápido.
- **Cooperación**. Existe una estrecha interacción entre cliente y equipo desarrollador.
- **Sencillo**. El método en sí es fácil de aprender, modificar y está suficientemente documentado.

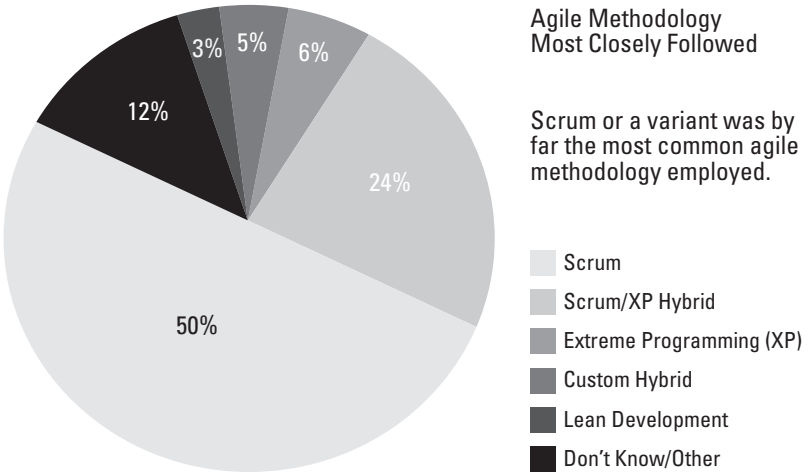

Figura 4.1: Metodologías ágiles más utilizadas.

- **Adaptación**. Gran capacidad para reaccionar ante cambios en todo momento.

En la figura 4.1 se muestran las metodologías más utilizadas en los últimos años. Una metodología ágil no tienen por qué cubrir todas las fases del proyecto: inicio del proyecto, especificación de requisitos, diseño, codificación, pruebas unitarias, tests de integración, tests de aceptación y sistema en servicio. Luego estas metodologías pueden no ser adecuadas o aplicables a cualquier tipo de proyecto (empresa, producto, línea de negocio, etc...) y tendrán que combinarse con otras metodologías si se desea seguir unas directrices en todas las fases de desarrollo software y en las propias de gestión de proyectos, las cuales apenas hacen énfasis la mayoría de metodologías ágiles.

La definición moderna de desarrollo ágil de software evolucionó a mediados de 1990 como parte de una reacción contra los métodos de "peso pesado" que se caracterizaban por ser muy estructurados, burocáticos, lentos y estrictos, extraídos del modelo de desarrollo en cascada. Inicialmente, los métodos ágiles fueron llamados métodos de "peso liviano".

En la figura 4.2 se expone un diagrama de los métodos de desarrollo de software ágiles más relevantes, sus interrelaciones y sus caminos evolutivos. También se han incluido otros métodos que tienen una relación directa o indirecta con las metodologías ágiles por haberlas influido de alguna manera. Las líneas discontinuas en el diagrama tiene el significado que dichos métodos (o sus autores) contribuyeron a la publicación del *Manifiesto Ágil*. Se observa claramente como en 2001 se firmó dicho manifiesto pero ya antes se desarrollaron importantes metodologías ágiles.

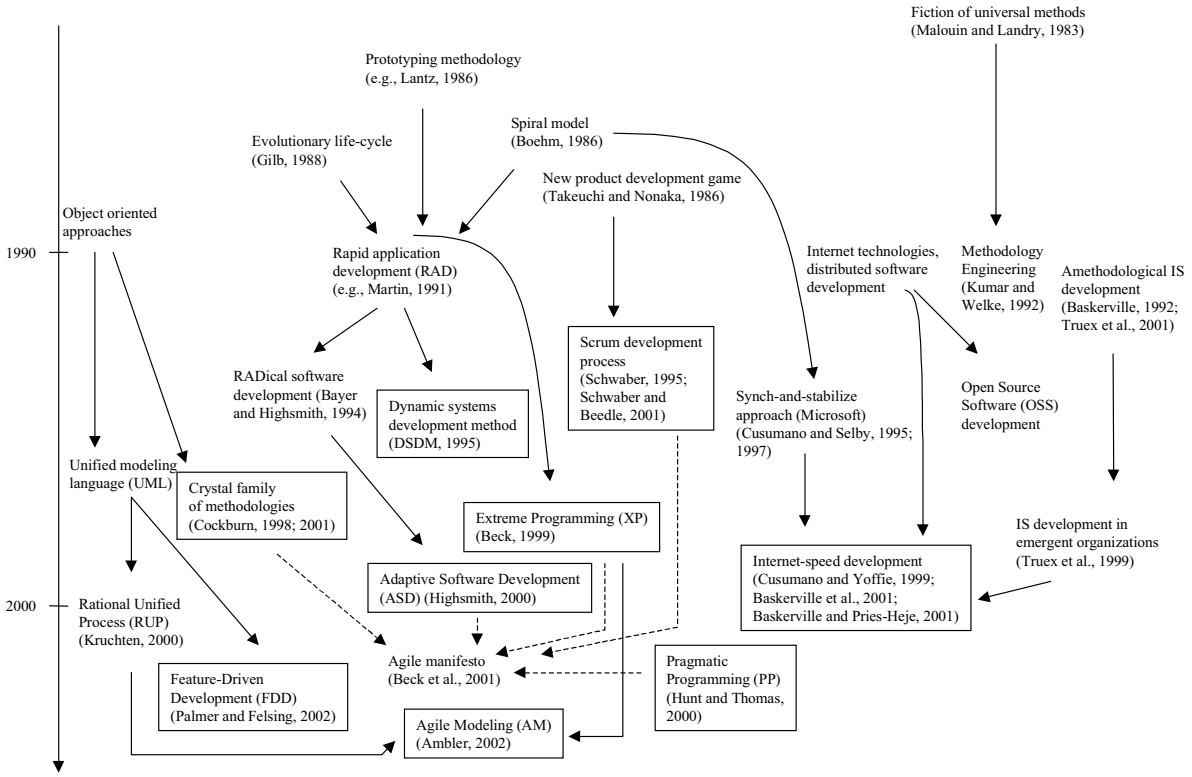

Figura 4.2: Evolución histórica de las metodologías ágiles.

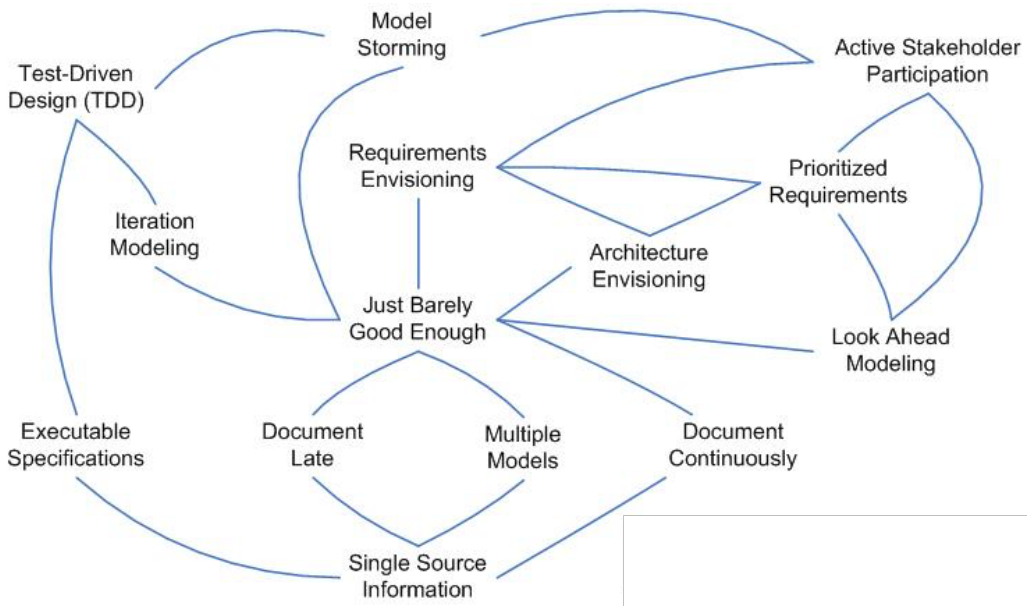

Figura 4.3: Buenas prácticas de la metodología Agile Modeling (AM).

4.2. Metodologías

A continuación se describen resumidamente algunas de las metodologías ágiles más destacadas: Agile Modeling (AM), Adaptive Software Development (ASD), Agile Unified Process (AUP), Crystal, Dynamic Systems Development Method (DSDM), Feature Drive Development (FDD), Lean Software Development (LSD), eXtreme Programming (XP), Scrum, Kanban y Scrumban.

4.2.1. Agile Modeling (AM)

El desarrollo de Agile Modeling (AM) fue iniciado por *Scott Ambler* en el 2000 publicando su libro "Agile Modeling" en 2002. AM es una metodología para el modelado de sistemas y documentación de software utilizando las mejores prácticas. Es una colección de valores, principios y buenas prácticas que se puede aplicar en un proyecto de desarrollo de software. Es decir, AM no es un proceso prescriptivo, no define los procedimientos detallados para la forma de crear un determinado tipo de modelo, sino que ofrece un conjunto de recomendaciones. No es una metodología que abarque todas las fases de un proyecto sino que, como ya se ha dicho, se centra en el modelado y la documentación. No incluye cómo gestionar el proyecto, la programación de actividades, testeo u otras fases.

Por lo tanto, AM es un complemento a otras metodologías ágiles como Scrum, XP o Rational Unified Process (RUP). Sin embargo, según estadísticas del 2011, AM se usa

muy minoritariamente representando tan sólo el 1 % de todo el desarrollo ágil de software.

AM adoptó los mismos **valores** que tiene XP: comunicación, simplicidad, realimentación, coraje y humildad (en XP se nombra como respeto). Los **principios** de AM son: modelar con un propósito, maximizar el retorno de la inversión (ROI) de los involucrados, viajar ligero (cada artefacto creado deberá de ser mantenido), múltiples modelos, realimentación rápida, asumir la simplicidad, abrazar el cambio, pequeños incrementos, trabajo con calidad, el primer objetivo es hacer software funcional y el segundo objetivo es tener siempre en mente el siguiente paso a realizar. Por último, las **buenas prácticas** se muestran en la figura 4.3.

Para ampliar la información visitar http://www.agilemodeling.com donde *Scott Ambler* desarrolla su metodología.

4.2.2. Adaptive Software Development (ASD)

ASD fue propuesta por *Jim Highsmith* y *Sam Bayer* a comienzos de 1990 y publicado en el año 2000. La filosofía de ASD se basa en la colaboración humana y la auto-organización del equipo. Se adapta al cambio en lugar de luchar contra él, es decir, se basa en la adaptación continua a circunstancias cambiantes.

El ciclo de desarrollo de ASD se divide en 3 fases (ver figura 4.4).

1. **Especulación**. Al comienzo del proyecto se establecen sus principales objetivos, limitaciones y riesgos. Se hace una estimación del marco temporal del proyecto: determinar el número de iteraciones y su duración. Para cada iteración se definen sus objetivos y funcionalidades. En cada nueva iteración se volverá a analizar y recalcular en base a lo ya ejecutado.

2. **Colaboración**. Es la etapa donde se realiza concurrentemente el trabajo de desarrollo y gestión del producto. En esta fase son revisados en profundidad los requisitos y se define cómo se va a trabajar de acuerdo a las habilidades de cada miembro del equipo.

3. **Aprendizaje**. Como la filosofía es el aprendizaje continuo, esta fase es un elemento crítico para la eficacia de los equipos. En cada iteración se revisa: calidad del producto desde el punto de vista del cliente y por otro lado desde los desarrolladores, la gestión del rendimiento (evaluar lo que se ha aprendido) y la situación del proyecto (como paso previo a la planificación de la siguiente iteración del proyecto).

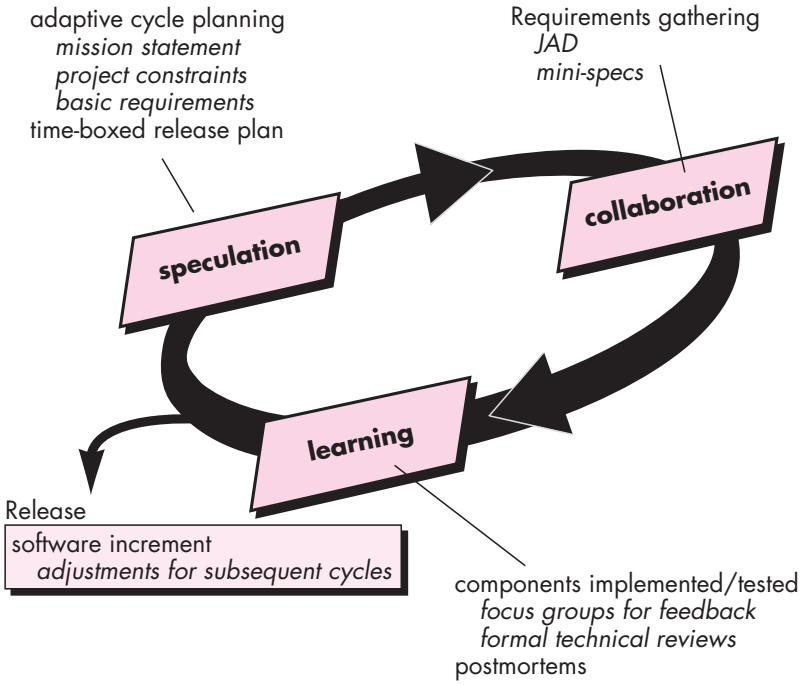

Figura 4.4: Ciclo de desarrollo de la metodología Adaptive Software Development (ASD).

4.2.3. Agile Unified Process (AUP)

AUP fue creada por *Scott Ambler* en 2002. Es una versión simplificada de Rational Unified Process (RUP) desarrollado por *IBM*. Describe una manera simple de entender el desarrollo de aplicaciones de negocio usando técnicas ágiles y conceptos heredados de RUP. Usa técnicas ágiles tales como Test Driven Development (TDD), modelado ágil, gestión de cambios ágil y la refactorización del código. En 2012, AUP fue reemplazado por Disciplined Agile Delivery (DAD)[1] dejándose de evolucionarse.

Se enumeran los 6 **principios** en los que se fundamenta AUP: los empleados saben lo que están haciendo, simplicidad, agilidad, centrarse en las actividades de alto valor, independencia de herramientas (cualquier conjunto que se adapte y optimize el trabajo) y adaptar AUP para cumplir con las necesidades propias del proyecto.

En la figura 4.5 se expone el ciclo de desarrollo de AUP. Se compone de 7 disciplinas y de 4 fases que se explican en los siguientes párrafos. Distingue 2 tipos de iteraciones: iteraciones de desarrollo (son incrementos menores que van al área de pruebas) e iteracciones de versiones de producción (incrementos mayores y probados que van al área de producción).

Las **disciplinas** son ejecutadas de una forma iterativa, definiendo las actividades, las

[1]Más información en http://www.disciplinedagiledelivery.com

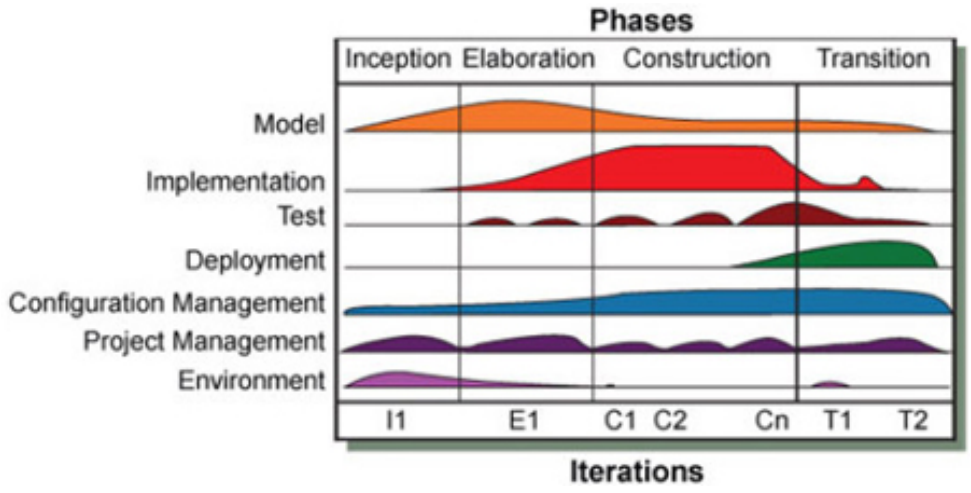

Figura 4.5: Ciclo de desarrollo de la metodología Agile Unified Process (AUP).

cuales, el equipo de desarrollo ejecuta para construir, validar y liberar software funcional, el cual cumple con las necesidades de los involucrados. Las disciplinas son:

- **Modelado**. Su finalidad es comprender el negocio de la organización, el problema del proyecto e identificar una solución factible.
- **Implementación**. Su finalidad es transformar el modelo en un código ejecutable y realizar prueba básicas.
- **Test**. Su finalidad es ejecutar una evaluación de los objetivos para asegurar la calidad. Esto incluye encontrar defectos, validar que el sistema funciona como fue diseñado y verificar que cumple los requerimientos.
- **Despliegue**. Se planifica la entrega del sistema y ejecuta el plan de despliegue para que esté operativo para los usuarios.
- **Gestión de la configuración**. Se gestiona el acceso a los entregables del proyecto: controles de versiones, administras los cambios, etc...
- **Gestión del proyecto**. Se dirigen las actividades y recursos del proyecto: gestión del riesgo, gestión de los RRHH (asignar tareas, seguimiento de los procesos, etc...), coordinarse con todos los involucrados del proyecto, etc...
- **Entorno**. Se asegura que los procesos óptimos, guías, estándares y herramientas estén disponibles para el equipo.

Las **fases** se realizan en serie a lo largo del proyecto.

1. **Inicio**. Identificar el alcance inicial del proyecto, una arquitectura recomendable para el sistema, obtener fondos y la aceptación por parte de las personas involucradas.
2. **Elaboración**. Probar la arquitectura del sistema.
3. **Construcción**. Construir software operativo de forma incremental que cumpla con las necesidades prioritarias de los involucrados.
4. **Transición**. Validar y desplegar el sistema en el entorno de producción.

4.2.4. Crystal

Alistair Cockburn (uno de los autores del *Manifiesto Ágil*) describió a mediados de los 90 un conjunto (familia) de metodologías ligeras o ágiles a las que llamó Crystal. Su nombre proviene de los minerales, que se caracterizan por 2 dimensiones: color y dureza. Análogamente los proyectos software también pueden caracterizarse según 2 dimensiones: tamaño o dimensión (número de personas en el proyecto) y criticidad (consecuencia de los errores). Ver el esquema de Crystal en la figura 4.6. La criticidad se puede clasificar de menos a más grave: pérdida de comfort o usabilidad, pérdidas económicas moderadas, pérdidas económicas graves y pérdida de vidas humanas. Según el número aproximado de personas se le llama con un color, algunos son: crystal clear, crystal yellow, crystal orange, crystal orange web, crystal red, crystal maroon, crystal diamond o crystal sapphire.

Cockburn en las conclusiones de uno de sus artículos, llamado la "Cockburn Scale", rompe con la antigua idea, que aún persiste en algunas empresas, de que existen metodologías omnipontentes que se implantan directamente y por completo. Así expone que no existe una metodología de desarrollo software única, mejor y universal, sino que por cada tipo de proyecto hay una metodología óptima, que permita la maniobrabilidad en el proyecto. Los parámetros fundamentales para seleccionar una metodología son el tamaño del equipo, su distribución, la criticidad del proyecto y las prioridades.

Para lograr la maniobrabilidad, el autor definió un conjunto de metodologías, cada una con elementos básicos comunes a todas. Además definieron los roles, modelos de procesos, productos de trabajo y prácticas que son únicas para cada una. Crystal es en realidad un conjunto de ejemplos de procesos ágiles que se han demostrado eficaces para diferentes tipos de proyectos. El objetivo es permitir a los equipos ágiles seleccionar qué miembro de la familia Crystal es el más apropiado para su proyecto y entorno.

Las metodologías Crystal se fundamentan en 7 **principios**.

- **Entregas frecuentes** en base a un ciclo de vida iterativo e incremental. En función del proyecto puede haber desde entregas semanales hasta trimestrales. No están fijadas como en otras metodologías como Scrum.

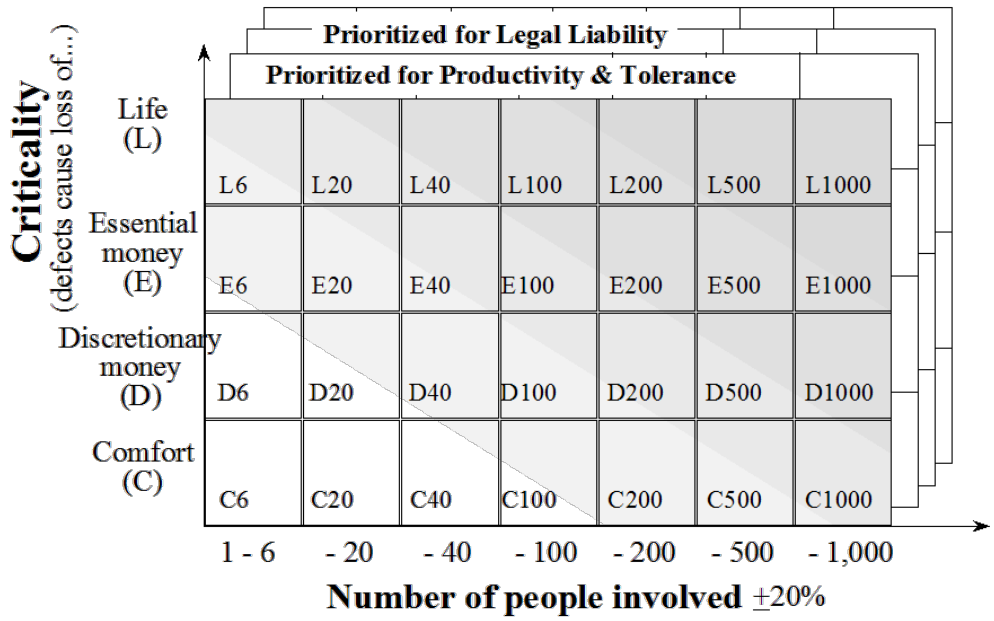

Figura 4.6: Esquema de la metodología Crystal.

- **Mejora reflexiva**. Las iteraciones ayudan a ir ajustando el proyecto, es decir, una mejora continua.
- **Comunicación osmótica**. El equipo está en una misma ubicación física para lograr la comunicación directa y fluida.
- **Seguridad personal**. Todo el mundo puede expresar su opinión sin cortapisas, considerándose su opinión.
- **Enfoque**. Períodos de no interrupción al equipo (2 horas), objetivos y prioridades claros, definiendo así tareas concretas. El entorno físico afecta al rendimiento del desarrollador software.
- **Fácil acceso al cliente**. No es obligatorio que los clientes estén continuamente junto al equipo de proyecto (no todas las organizaciones pueden hacerlo). Como mínimo, reuniones semanales y los clientes deben estar accesibles.
- **Entorno técnico**. Pruebas automatizadas, gestión de la configuración e integración continua.

4.2.5. Dynamic Systems Development Method (DSDM)

El Método de Desarrollo de Sistemas Dinámicos, en inglés Dynamic Systems Development Method (DSDM), es un método que provee un framework para el desarrollo ágil

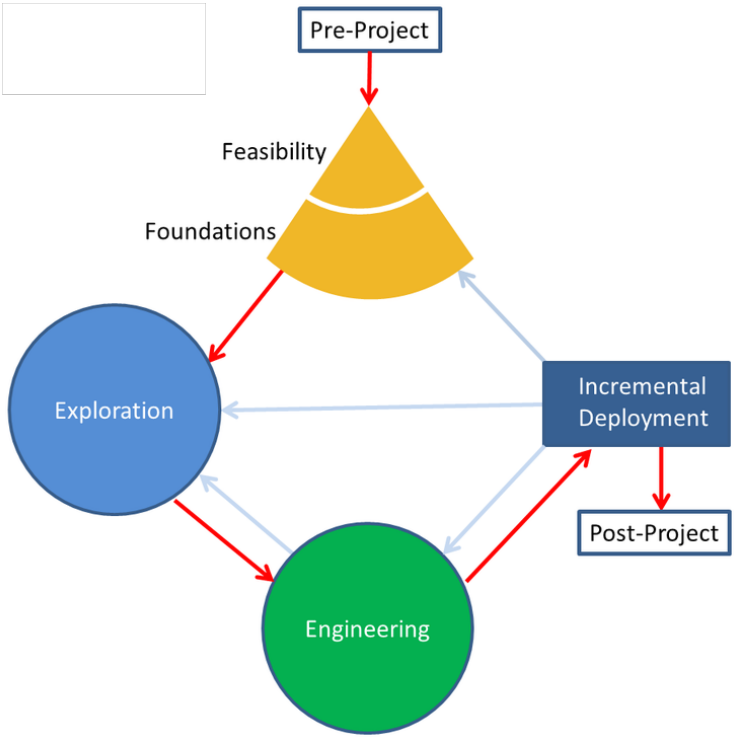

Figura 4.7: Ciclo de desarrollo de la metodología Dynamic Systems Development Method (DSDM).

de software, apoyado por su continua implicación del usuario en un desarrollo iterativo y creciente que sea sensible a los requerimientos cambiantes, para desarrollar un sistema que reuna las necesidades de la empresa en plazos y costes. Se podría decir que es la metodología ágil que más se aproxima a las tradicionales permitiendo alcanzar un nivel 2 de madurez según CMMI.

Es la única de las metodologías que se mencionan en este TFM que ha surgido de un consorcio, el *DSDM Consortium*[2]. El consorcio es una organización no lucrativa y proveedor independiente, que posee, administra y publica gratuitamente el framework. Su objetivo era producir una metodología de dominio público que fuera independiente de las herramientas y que pudiera ser utilizado en proyectos de tipo Rapid Application Development (RAD). La primera versión fue publicada en 1995 habiendo posteriormente otras. La versión actual está disponible en su web titulándose "DSDM Agile Project Framework" (2014).

En algunas circunstancias, hay posibilidades para integrar contenido de otros métodos. Por ejemplo, RUP, XP, PRINCE2 o PMBOK pueden complementar a DSDM en la realización de un proyecto.

DSDM está compuesto por 3 **fases** (ver figura 4.7).

[2]Su web oficial es `http://www.dsdm.org`

1. **Pre-proyecto**. En el anteproyecto se identifican los proyectos candidatos, se consigue la financiación y el compromiso del proyecto está asegurada. El manejo de estas cuestiones al comienzo evita problemas en etapas posteriores.
2. **Ciclo de vida del proyecto**. Se compone de 5 etapas. Las 2 primeras se hacen de manera secuencial y se complementan entre sí. Después se realiza de manera iterativa e incremental las 3 siguientes etapas.
   2.1 **Estudio de viabilidad**. Se estudia si esta metodología se ajusta al proyecto.
   2.2 **Estudio del negocio**. Se involucra al cliente de forma temprana para comprender el problema a abordar. Sienta las bases para iniciar el desarrollo, definiendo las requisitos a alto nivel.
   2.3 **Iteración del modelo funcional**. Produce una serie de prototipos incrementales que muestran la funcionalidad para el cliente. Su propósito es recopilar requisitos adicionales y generar documentación de análisis.
   2.4 **Diseño e iteración de la estructura**. Revisa la construcción de prototipos durante la anterior etapa. En esta se diseña el sistema para su uso operacional. En algunos casos, ambas etapas se realizan de manera concurrente.
   2.5 **Implantación**. Cuando el cliente ha validado el sistema, se pone en servicio.
3. **Post-proyecto**. En esta fase se asegura que el sistema funcione de manera eficiente. Esto se realiza mediante mantenimiento, mejoras o correcciones. El mantenimiento puede ser visto como un desarrollo continuo basado en el carácter iterativo e incremental de DSDM. En lugar de terminar el proyecto en un ciclo, el proyecto puede retornar a etapas anteriores de tal manera que los entregables sean refinados.

Hay 9 **principios** en los que se fundamenta DSDM.

- Involucrar al cliente activamente.
- El equipo del proyecto debe tener el poder para la toma decisiones.
- Dar frecuentemente entregables.
- El principal criterio de aceptación de entregables es que satisfaga las actuales necesidades de negocio (funcionalides críticas, no accesorias).
- Desarrollo iterativo e incremental.
- Los cambios durante el desarrollo pueden ser reversibles.
- El alcance a alto nivel y los requerimientos deberían ser estar fijados como una línea base antes del desarrollo del proyecto.
- Las pruebas son realizadas durante todo el ciclo de vida del proyecto.

Figura 4.8: Roles la metodología Dynamic Systems Development Method (DSDM).

- Comunicación y cooperación entre todos los involucrados.

DSDM propone varios **roles** (ver figura 4.8), cada uno con su propia responsabilidad. Las roles son: esponsor ejecutivo, visionario, usuario embajador, usuario asesor, project manager, coordinador técnico, líder de equipo, desarrollador, tester, recopilador, facilitador y especialistas (arquitecto, gestor de calidad, integrador de sistemas, etc...).

4.2.6. Feature Drive Development (FDD)

Feature Drive Development (FDD) fue concebido originalmente por *Peter Coad* y *Jeff De Luca* mediante su libro "Java Modeling in Color with UML" en 1999 como un modelo de procesos práctico para la Ing. de Software orientado a objetos. *Stephen Palmer* y *John Felsing* en 2002 publicaron "A Practical Guide to Feature Driven Development" donde amplian y mejoran el anterior trabajo, que describen unos procesos adaptativos, ágiles, que se puede aplicar a proyectos de tamaño medios y grandes.

Al igual que otros metodologías ágiles, FDD adopta una filosofía que:

- Hace hincapié en la colaboración entre las personas del equipo.
- Gestiona los problemas y la complejidad del proyecto utilizando una descomposición basada en características (o funciones) mediante su integración en sucesivos incrementos del software.
- Comunica los detalles técnicos utilizando medios verbales, gráficos y escritos.

El ciclo de desarrollo se muestra en la figura 4.9 y más detalladamente en la figura 4.10. El ciclo de desarrollo se divide en 5 fases y es de tipo incremental, constando cada incremento (iteración) en 2 fases: diseño y construcción de una característica. Se explica a continuación cada etapa.

1. **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.
2. **Construir lista de características**. 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.
3. **Planificar**. Se procede a ordenar los conjuntos de funcionalidades conforme a su prioridad y dependencia, y se asigna a los programadores jefes.
4. **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.
5. **Construir**. Se procede a la construcción total del proyecto.

Una característica se define como "una función que aporta valor al cliente y que puede ser implementada en dos semanas o menos". Esta definición ofrece los siguientes beneficios.

- Como las características son pequeños bloques de funcionalidades entregables, los usuarios pueden comprenderlas y describirlas con mayor facilidad así como entender cómo se relacionan entre sí evitando ambigüedades, errores u omisiones.
- Las características pueden estar organizadas en grupos relacionados jerárquicamente con el negocio.
- Dado que una característica es un incremento software entregable, el equipo desarrolla características operativas cada dos semanas.

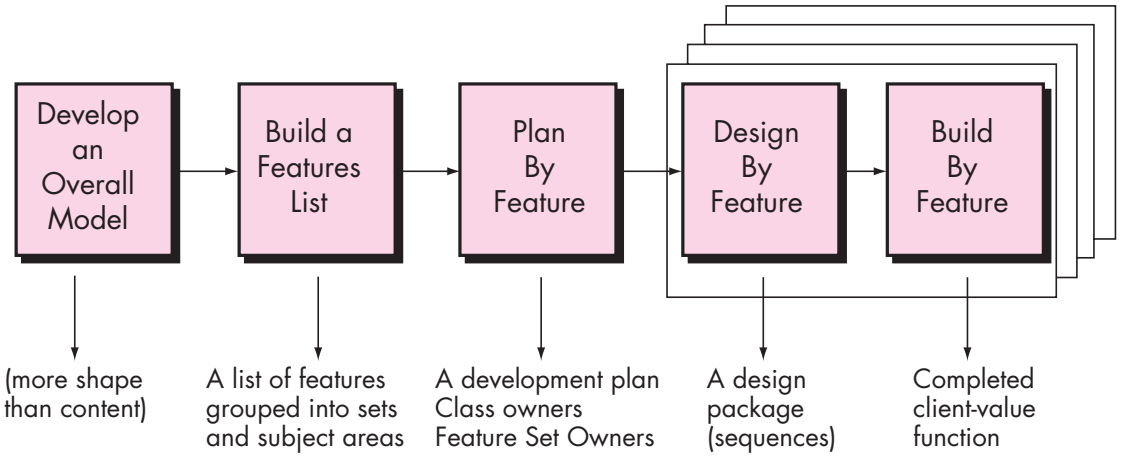

Figura 4.9: Ciclo de desarrollo de la metodología Feature Drive Development (FDD).

- Debido a que las características son pequeñas, su diseño y código son inspeccionables eficientemente.
- La principal desventaja es que, como todo ciclo de desarrollo incremental, hay costes derivados al integrar cada nuevo incremento software: rediseño, refactorización del código, volver a testear el código, etc...

FDD enfatiza las actividades de aseguramiento de la calidad del software con una estrategia incremental de desarrollo, el uso del diseño e inspecciones de código, la aplicación de auditorías de aseguramiento de la calidad, métricas y el uso de patrones para el análisis, diseño y construcción.

4.2.7. Lean Software Development (LSD)

Mary y *Tom Poppendieck* publicaron el libro "Lean Software Development" en 2003 que da origen a esta metodología. Surge como una corriente de pensamiento que aplica los principios de fabricación Lean al desarrollo de software, ideada por *Taiichi Ohno* en 1956 y que es la esencia del sistema de producción de Toyota (llamado *Toyota Production System*, TPS). El libro presenta los tradicionales principios Lean de forma modificada, así como un conjunto de 22 instrumentos y herramientas y las comparaciones con otras prácticas ágiles. La participación de los autores en la comunidad del desarrollo ágil de software, incluyendo charlas en varias conferencias, ha dado lugar a dichos conceptos, que son más ampliamente aceptados en la comunidad de desarrollo ágil.

Lean y LSD no son una metodología de Ing. de Software en el sentido convencional. Es más una síntesis de principios y una filosofía para construir sistemas software. Si Lean se

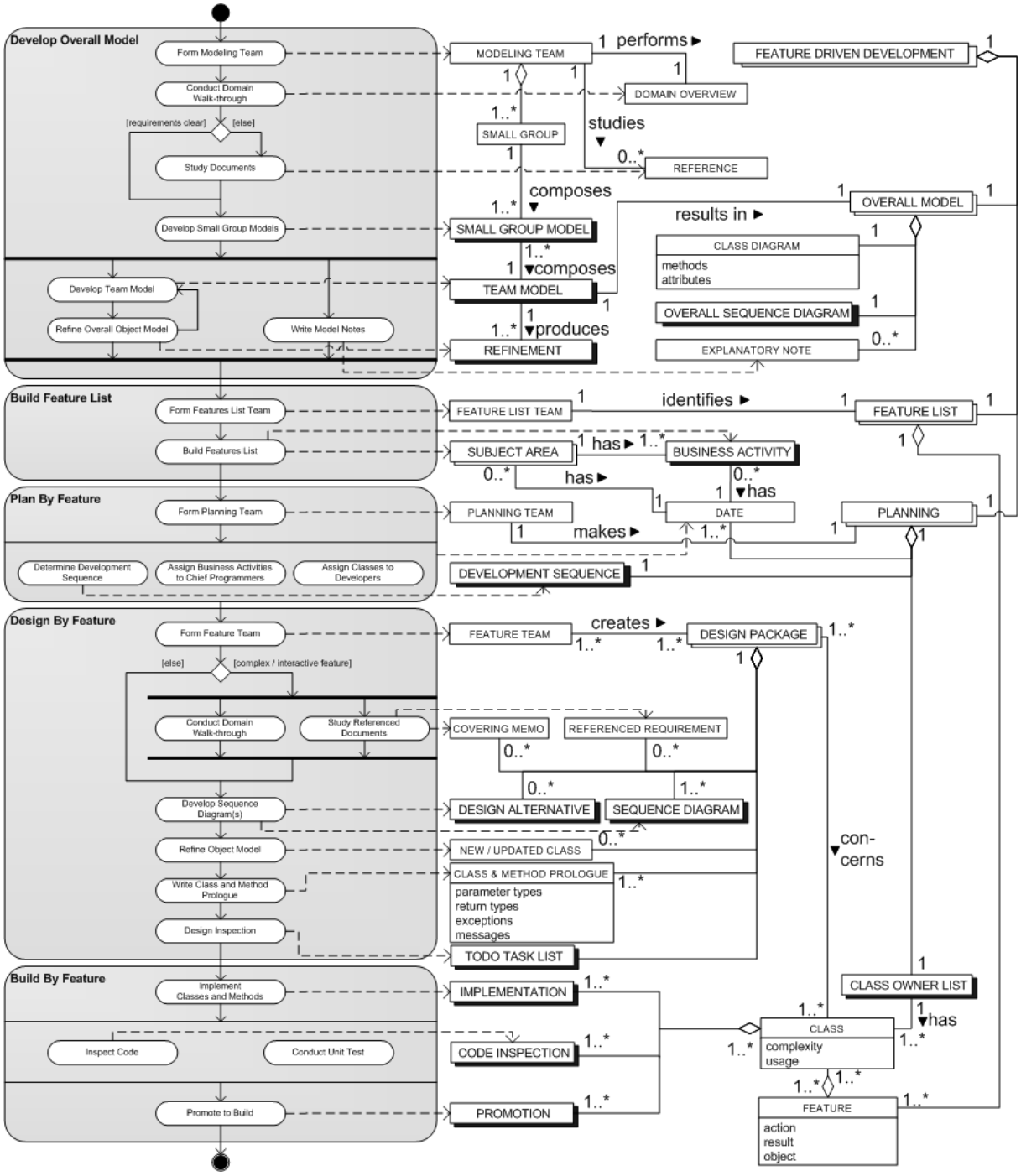

Figura 4.10: Diagrama de los procesos detallados de la metodología Feature Drive Development (FDD).

considera un conjunto de principios más que prácticas, la aplicación de conceptos Lean a la Ing. de Software tiene más sentido y puede ayudar a mejorar la calidad y desarrollo del software. Tiene las siguientes características.

- Lean es una filosofía y una forma de pensar.
- Analiza los procesos de producción y elimina todo aquello (desperdicios) que no produzca valor para el cliente.
- Es un conjunto de conceptos y técnicas pensadas para el aumento de la productividad y la producción con calidad.
- Las metodologías ágiles se basaron en Lean para establecer sus principios.

Hay 7 **principios** en los que se fundamenta LSD.

- **Eliminar los desperdicios**. Todo lo que no añade valor al cliente se considera un desperdicio y se debe eliminar. Por ejemplo, código y funcionalidades innecesarias, retrasos en los plazos, requisitos poco claros, burocracia o comunicación interna lenta. La dificultad radica en ser capaces de reconocer y encontrar los desperdicios en el proyecto. Lean describe 8 tipos de desperdicios: movimiento, sobreproducción, espera, transporte, procesado extra, corrección, inventario y conocimiento desconectado.
- **Crear conocimiento**. Llegar a conocer lo que realmente necesita el cliente requiere esfuerzo y dedicación. Debe convertirse en el aspecto principal a tener en cuenta porque el desarrollo de un producto inútil termina siendo un desperdicio. El proceso de desarrollo de software es un proceso de aprendizaje: comprender qué es lo que el cliente quiere y cómo hacerlo lo mejor posible. El desarrollo iterativo incremental permite repetir muchas veces el proceso de aprendizaje para poder crear el conocimiento necesario.
- **Diferir el compromiso**. El compromiso, esencialmente 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 una posible causa para que fracase el proyecto. Es decir, el desarrollo de software está siempre asociado a un cierto grado de incertidumbre y los mejores resultados se alcanzan con un enfoque basado en opciones por lo que se pueden retrasar las decisiones tanto como sea posible hasta que éstas se basen en hechos y no en suposiciones.
- **Entregar rápido**. El desarrollo iterativo permite realizar entregas rápidas a los clientes encontrándose con código funcional desde etapas tempranas. El código debe ser desarrollado con calidad para mantener una velocidad importante de entrega si no se cuenta con calidad y un equipo disciplinado, comprometido y confiable.

- **Potenciar el equipo**. Ha habido una creencia tradicional en la mayoría de las empresas acerca de la toma de decisiones en la organización: los jefes dicen a los trabajadores cómo hacer su propio trabajo. Sin embargo los roles deben cambiar: los directivos escuchan a los desarrolladores, de manera que éstos puedan explicar mejor qué acciones podrían tomarse dando sugerencias y mejoras. Otra creencia errónea ha sido considerar a las personas como recursos. Las personas necesitan algo más que una lista de tareas y la seguridad de que no será alterada durante su realización. Las personas necesitan motivación y objetivos alcanzables garantizando que el equipo puede elegir sus propios compromisos.

- **Construir con calidad**. La calidad debe ser global, tanto para el proceso como para el producto. Un proceso que respeta la calidad es aquel que es conocido, entendido y mejorado por sus propios participantes. Para desarrollar productos con calidad se pueden tener en cuenta elementos como: técnicas como Test Driven Development (TDD), el programador es responsable de su propio desarrollo (no se debe esperar a futuros test de pruebas), fomentar el desarrollo de pruebas automatizadas y refactorizar el código buscando que no existan duplicaciones (eliminar desperdicios).

- **Optimizar el todo**. Lean se centra en el proceso completo, es decir, en todo el flujo de valor en lugar de hacerlo en cada etapa. El problema con optimizar cada paso es que genera stocks intermedios. En software esto se prodría traducir a trabajo parcialmente terminado. Por ejemplo, requerimientos completos pero sin diseñar, codificar o probar. Lean demostró que un flujo de una sola etapa es un proceso más eficiente que si se dividiese en varias etapas.

4.2.8. eXtreme Programming (XP)

La programación extrema o en inglés eXtreme Programming (XP) es una metodología liviana de desarrollo de software formulada por *Kent Beck*, autor en 1999 del primer libro sobre la materia *Extreme Programming Explained: Embrace Change*. Este libro no cubría los detalles técnicos y de implantación de las prácticas. Posteriormente, otras publicaciones se han encargado de dicha tarea. A comienzos de la década del 2000 fue el más destacado de los procesos ágiles de desarrollo de software pero actualmente ya no es tan popular.

El origen de la metodología es a raíz del proyecto C3 para la empresa *Chrysler*. Se centra en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software promoviendo el trabajo en equipo. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos, muy cambiantes y donde existe un alto riesgo técnico.

Para lograr simplicidad, los desarrolladores sólo deben diseñar y codificar las necesidades

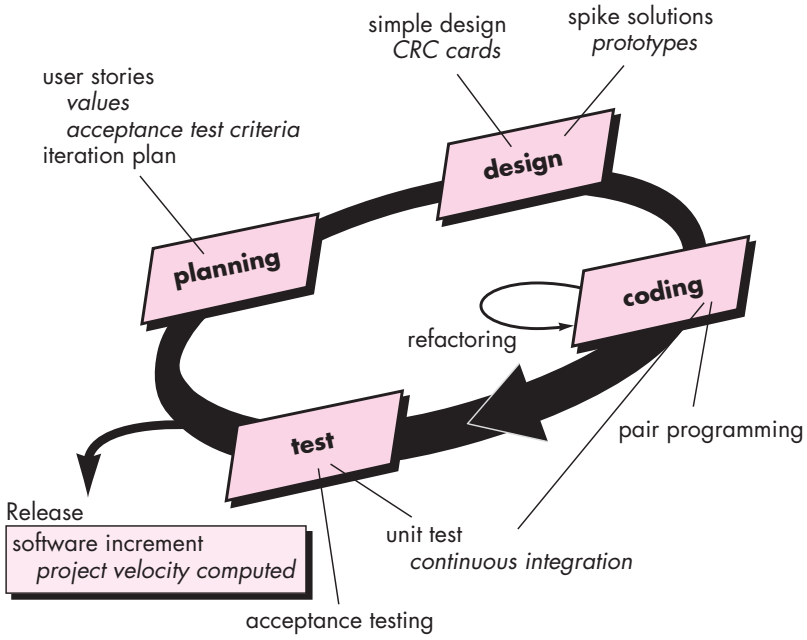

Figura 4.11: Ciclo de desarrollo de la metodología eXtreme Programming (XP).

inmediatas, en lugar de considerar necesidades futuras. La intención es crear un diseño simple que puede ser implementado fácilmente en código. Si el diseño debe ser mejorado, se refactoriza en un momento posterior.

Las **historias de usuario** son la técnica utilizada en XP para especificar los requisitos y priorizar el desarrollo. Se trata de tarjetas en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no. Es decir, cada historia describe las salidas requeridas, sus características y sus funcionalidades. El cliente asigna un *valor* (prioridad) a esa historia y el equipo desarrollador un *coste* (medido en semanas de trabajo). El tratamiento de las historias es muy flexible, en cualquier momento pueden eliminarse, añadirse nuevas, modificarse o reemplazarse por otras más específicas o generales. La *velocidad* es el número de historias que se hacen por versión ayudando a dar estimaciones de plazos.

El ciclo de desarrollo de XP se puede dividir en 4 actividades fundamentales (ver figura 4.11).

1. **Planificación**. Se comienza por entender el negocio para el cual va a ser el proyecto. Se escriben las historias de usuario y entre el cliente y los desarrolladores se agrupan (por su valor y coste) para priorizar las que se van a hacer en la próxima versión (incremento de software).

2. **Diseño**. Se sigue rigurosamente la práctica de diseño simple. XP usa las llamadas tarjetas CRC (Class - Responsibility - Collaborator) como un mecanismo eficaz para

diseñar el software. Dichas tarjetas identifican y organizan las clases orientada a objetos que son relevante para la versión. Si se encuentran problemas en el diseño a raíz de una historia, se recomienda realizar un prototipo (llamado spike solution) para reducir los riesgos. El diseño se produce tanto antes como después de la codificación, es decir, refactorizar significa que el diseño se produce continuamente a medida que el código se construye.

3. **Codificación**. Los desarrolladores no empiezan a escribir código inmediatamente sino que desarrollan una serie de pruebas unitarias para cada una de las historias incluidas en la versión actual. Creadas las pruebas, el desarrollador está en mejores condiciones de centrarse en lo que debe ser implementado para pasar la prueba. Un concepto clave en la actividad de codificación es utilizar la buena práctica de programación en parejas y se hará una integración continua del código con el resto del equipo.

4. **Test**. Además de las pruebas unitarias ya mencionadas también existen las pruebas de aceptación (o pruebas de clientes) que son especificadas por el cliente centrándose en las características del sistema en general y en las funcionalidades visibles. Éstas se derivan de las historias de usuario que se han implementado en la versión. Es importante que se tenga una estrategia de pruebas de regresión para cada vez que se modifica el código (que es frecuente dada la filosofía de refactorización).

Los **valores** en los que se fundamenta XP son 5. Estos valores son la guía para el desarrollo en sí mismo y la inspiración de toda la metodología.

- **Simplicidad**. XP propone el principio de hacer la cosa más simple que pueda funcionar, en relación al proceso y la codificación. Es mejor hacer algo simple ahora que hacerlo complejo y que probablemente nunca se use.

- **Comunicación**. Algunos problemas en los proyectos tienen origen en que alguien no dijo algo importante en algún momento. XP hace casi imposible la falta de comunicación.

- **Realimentación**. Retroalimentación concreta y frecuente del cliente, del equipo y de los usuarios finales da una mayor oportunidad de dirigir el esfuerzo eficientemente.

- **Coraje**. Se necesita valentía para realizar un proyecto donde se exige comunicación entre todas las partes y los requisitos van cambiando a medida que avanza el proyecto.

- **Respeto**. El respeto es imprescindible porque el trabajo es común con fuertes interacciones entre todas las partes del proyecto.

Para que la metodología XP tenga éxito deben seguirse las siguientes 13 **buenas prácticas** (ver figura 4.12). Se basa en combinar las que han demostrado ser las mejores prácticas para desarrollar software y llevarlas al extremo.

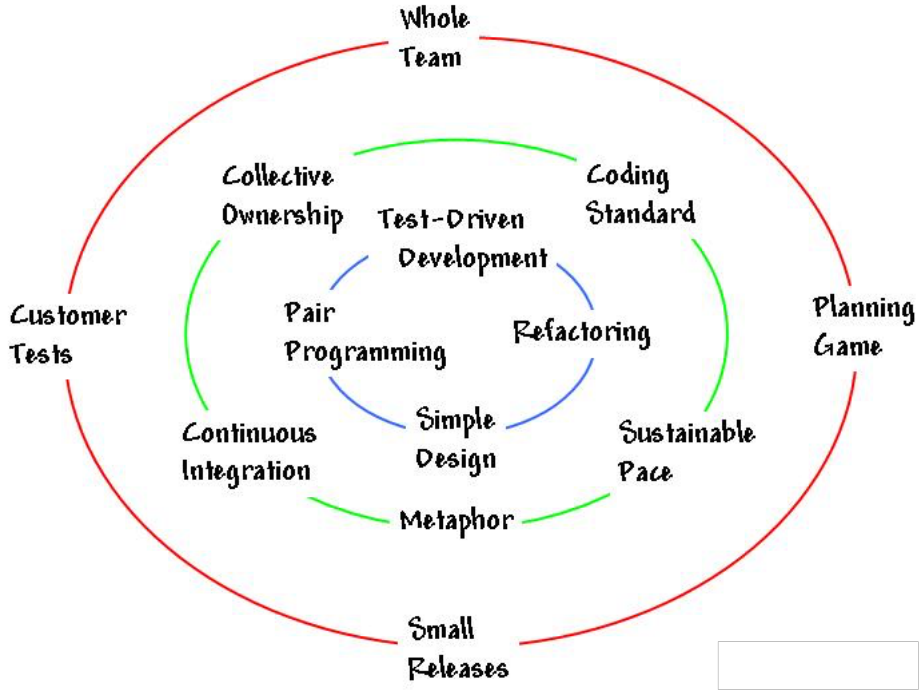

Figura 4.12: Las 13 prácticas básicas de la metodología eXtreme Programming (XP).

- **Equipo completo**. Forman parte del equipo todas las personas que tienen algo que ver con el proyecto, incluido el cliente y el responsable del proyecto.
- **Planificación**. Se hacen las historias de usuario y se planifica en qué orden se van a hacer y las mini-versiones. La planificación se revisa continuamente.
- **Test del cliente**. El cliente, ayudado por los desarrolladores, propone sus propias pruebas para validar las mini-versiones.
- **Versiones pequeñas**. Las mini-versiones deben ser lo suficientemente pequeñas como para poder hacer una cada pocas semanas. Deben ofrecer algo útil al usuario final y no código que no se pueda ver funcionando.
- **Integración continua**. Deben tenerse siempre un ejecutable del proyecto que funcione y en cuanto se tenga una nueva pequeña funcionalidad, debe recompilarse y probarse.
- **Propiedad colectiva**. Cualquiera puede tocar y conocer cualquier parte del código. Para eso se hacen las pruebas automáticas.
- **Codificación estandarizada**. Debe haber un estilo común (guía de estilo) en la codificación para tener un código homogéneo.

- **Metáforas**. Hay que buscar frases o nombres que definan las funcionalidades de las módulos del programa, de forma que sólo con los nombres se sepa a qué se está haciendo referencia. Esto ayuda a que todos los programadores y el cliente sepan de qué se está hablando.

- **Ritmo sostenible**. Se debe trabajar a un ritmo que se pueda mantener indefinidamente. Esto es, no debe haber días muertos sin trabajo ni otros con un exceso de horas trabajadas.

- **Diseño simple**. Hacer siempre lo mínimo imprescindible en el código de la forma más sencilla posible.

- **Programación en parejas**. Los programadores trabajan por parejas delante del mismo ordenador y se intercambian las parejas con frecuencia (un cambio diario).

- **Desarrollo guiado por las pruebas automáticas**. Se deben realizar programas de prueba automática y deben ejecutarse con mucha frecuencia.

- **Refactorización**. Consiste en mejorar la estructura interna de un diseño (o código fuente) sin cambiar su comportamiento o funcionalidad externa. Es muy importante refactorizar el código en cada versión para mantener su sencillez y coherencia.

En la nueva versión de XP, los **principios** son el puente entre los valores (sintéticos y abstractos) y las prácticas (indican cómo desarrollar software). Se enumeran los 14 principios: humanidad, economía, beneficio mutuo, auto-similitud, mejora, diversidad, reflexión, flujo, oportunidad, redundancia, fallo, calidad, pasos de bebé y aceptación de la responsabilidad.

Los **roles** del proyecto propuestos originalmente por *Kent Beck* son los siguientes.

- **Programador**. Produce el código del sistema y realiza pruebas unitarias.

- **Cliente**. El 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 aportar mayor valor al negocio.

- **Tester**. Es el encargado de realizar las pruebas y también ayuda al cliente a escribir las pruebas funcionales. Difunde los resultados al equipo y es el responsable de las herramientas de soporte para pruebas.

- **Tracker**. Es el encargado de seguimiento del proyecto. Su responsabilidad es controlar las estimaciones realizadas y el tiempo real dedicado. También realiza el seguimiento del progreso de cada iteración y evalúa si los objetivos son alcanzables con las restricciones de tiempo y recursos presentes proponiendo los cambios necesarios.

- **Entrenador**. Experto en la metodología XP para proveer guías al equipo de forma que se apliquen las prácticas y se siga el proceso correctamente.
- **Consultor**. Miembro externo al equipo con conocimientos específicos en alguna materia o tecnología necesaria para el proyecto ayudando al equipo en dicho tema.
- **Gestor**. Es el vínculo entre clientes y programadores realizando labores de coordinación. Ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas.

4.2.9. Scrum

Los inicios de Scrum se remontan al artículo "The New Product Development Game" (Harvard Business Review, 1986) de *Hirotaka Takeuchi* e *Ikujiro Nonaka* que introducía las mejores prácticas más utilizadas en 10 compañías tecnológicas japonesas. Describen un enfoque integral que incrementaba la velocidad y flexibilidad del desarrollo de nuevos productos. 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 (de ahí que se llame Scrum). En 1995 *Jeff Sutherland* y *Ken Schwaber* presentaron de forma conjunta la conferencia "Scrum Development Process" en la OOPSLA (Object-Oriented Programming Systems & Applications conference) en Austin, su primera aparición pública. Ambos 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. Actualmente es la metodología ágil más utilizada con diferencia.

Scrum es un marco de trabajo en el cual las personas pueden acometer problemas complejos adaptativos, a la vez que entregar productos del máximo valor posible productiva y creativamente. Según los autores, Scrum es ligero, fácil de entender y extremadamente difícil de llegar a dominar. Scrum no es un proceso o una técnica para construir productos sino que es un marco de trabajo dentro del cual se pueden emplear varias técnicas y procesos. El marco de trabajo Scrum tiene los siguientes componentes: los equipos scrum, roles, eventos, artefactos y reglas asociadas. Las reglas relacionan los eventos, roles y artefactos, gobernando las relaciones e interacciones entre ellos. Las estrategias específicas para usar el marco de trabajo son diversas y no están descritas en la metodología. En la figura 4.13 se muestra la operativa de Scrum con sus principales componentes.

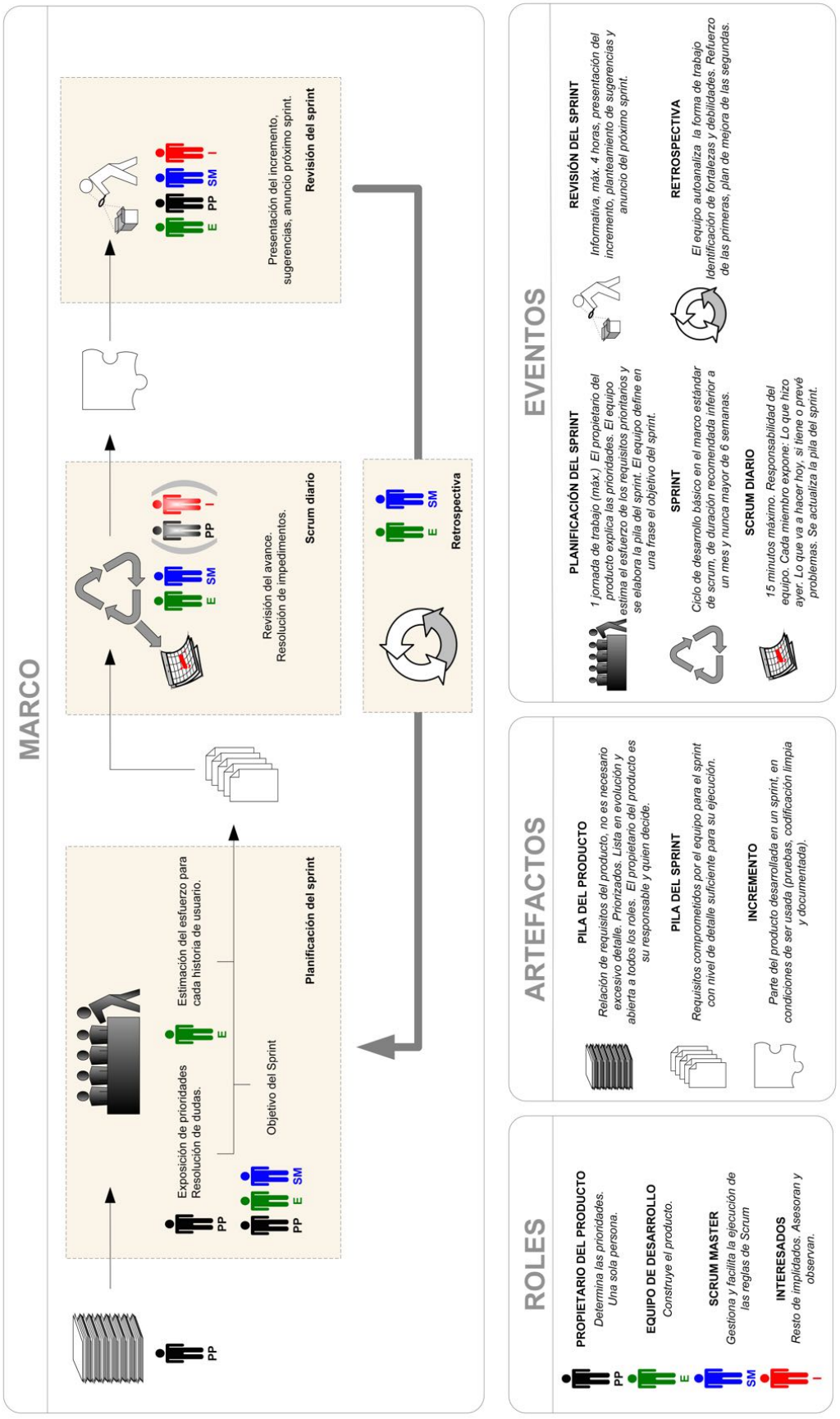

Figura 4.13: Esquema de la metodología Scrum.

Las características más relevantes que se logran con Scrum son: gestión regular de las expectativas del cliente, resultados anticipados, flexibilidad y adaptación, retorno de inversión, mitigación de riesgos, productividad y calidad, alineamiento entre cliente y equipo y un equipo motivado.

Los roles principales en Scrum son el *Scrum Master* que mantiene los procesos y trabaja junto con el jefe de proyecto, el *Product Owner* que representa al cliente y el *Team* que incluye a los desarrolladores.

La operativa de Scrum se resume a continuación. En cada iteración (llamado sprint), típicamente un periodo de 2 a 4 semanas (decidido y fijado para siempre por el equipo), el equipo crea un incremento de software funcional. El conjunto de características que entra en una iteración viene del **product backlog** (pila del producto), que es un conjunto priorizado de requisitos de trabajo de alto nivel (historias de usuario) con su estimación temporal correspondiente. 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 **product backlog** que quiere que se completen. El equipo determina entonces a cuanto de eso puede comprometerse a completar durante la siguiente iteración formándose el **sprint backlog** (pila del sprint). Durante una iteración, nadie puede cambiar el backlog, lo que significa que los requisitos están congelados para esa iteración. Se empieza a desarrollar el software haciendo reuniones diarias (scrum diario) breves, típicamente de 15 minutos, para que cada persona del equipo diga sus avances y se actualice el **sprint backlog**. Cuando se completa una iteración, el equipo muestra el uso del software para que lo validen todos los involucrados del proyecto. Finalmente, se comenzaría otra nueva iteración.

4.2.10. Kanban

El método de desarrollo Kanban está basado en la metodología de fabricación industrial del mismo nombre inspirada en los sistemas *Toyota Production System* y *Lean manufacturing*. Su objetivo es gestionar de manera general como se van completando las tareas de un proyecto de una manera visual. El método Kanban, propuesto por *David J. Anderson* en 2008, es una aproximación al proceso gradual, evolutivo y al cambio de sistemas en las organizaciones.

El trabajo se divide en partes, normalmente cada una de esas partes se escribe en tarjetas que se ponen sobre un tablero. Las tarjetas suelen tener información variada: descripción, estimación temporal, etc... El objetivo de esta visualización es que quede claro el trabajo a realizar: las prioridades, tiempos y estados de las tareas y en qué está trabajando cada persona. La pizarra tiene tantas columnas como estados por los que puede pasar la tarea, típicamente: en standby, en análisis, en desarrollo, en pruebas y entregada. Por tanto, las fases del ciclo de producción o flujo de trabajo se deben decidir según el caso, no hay nada acotado. Se muestra un tablero Kanban a modo de ejemplo en la figura 4.14.

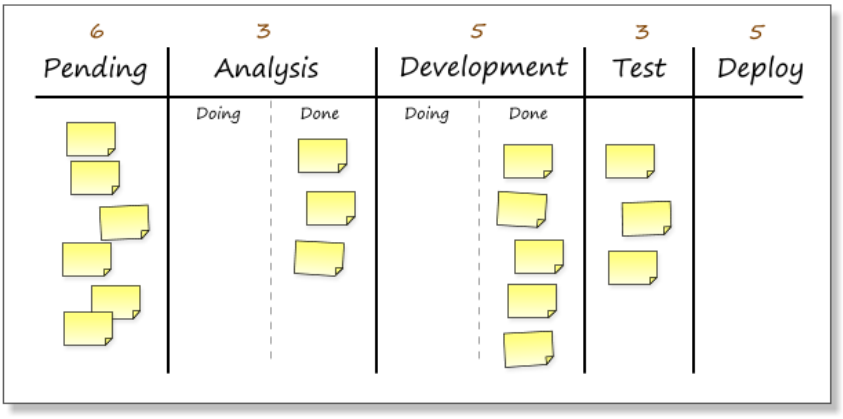

Figura 4.14: El tablero Kanban.

Kanban no prescribe roles porque entiende que su ausencia es una ventaja para el equipo para así evitar la resistencia al cambio para desempeñar un nuevo trabajo. Tampoco fija reuniones diarias ni establece unas fases definidas como ya se ha mencionado, sino que se habla de un flujo que se puede dividir según las necesidades particulares del proyecto liberando versiones o entregables.

Este método es bastante radical en cuanto a lo adaptativo que es y a lo poco que define los procesos, roles e incluso la forma de hacer el tablero Kanban. Muchos expertos consideran a Kanban como una técnica de señalización más que una metodología. Es decir, Kanban puede complementar a otras como Scrum o XP a la hora de representar visualmente las tareas y el avance del proyecto. Por tanto, no se tendrá en cuenta en este TFM como una metodología propiamente dicha como el resto ni se entrará más en detalle.

4.2.11. Scrumban

Scrumban es una metodología combinada y derivada de Scrum y Kanban. Algunos equipos sintieron que estas metodologías no encajaban del todo en su proceso de trabajo. Scrum es demasiado estricta para muchos entornos que necesitan más agilidad, mientras Kanban no se estructura suficientemente. Scrumban intenta proporcionar un punto medio, mezclando la estructura del Scrum y la planificación flexible de Kanban para crear una metodología ajustada a entornos en rápida evolución.

La tabla 4.1 resume las diferencias entre Scrum y Scrumban. Se podría decir que de Scrum toma los roles, las reuniones (diarias) y la pizarra como herramienta. De Kanban coge el flujo visual de las tareas, hacer lo que sea necesario (cuando sea necesario) y limitar la cantidad de trabajo (Work In Progress, WIK).

Es un modelo de desarrollo especialmente adecuado para proyectos de mantenimiento, proyectos en los que las historias de usuarios varíen frecuentemente o en los que la com-

Aspectos	Scrum	Scrumban
Herramientas	Pizarra, backlogs, gráfica burn-down	Pizarra
Tamaño tarea	Debe ser completado en un sprint	Cualquier tamaño
Prioridad tarea	A través del backlog	Recomendado en cada planificación
Reglas	Procesos limitados	Procesos ligeramente limitados
Reuniones	Reunión diaria, planificación, retrospectiva	Reunión diaria
Iteraciones	Sí, sprints	No, flujo continuo
Estimaciones	Sí	Opcional
Equipo	Multidisciplinar	Pueder ser especializado
Roles	Product Owner, Scrum Master y Equipo	Equipo y otros sin especificar
Work in progress (WIP)	Controlado por el contenido del sprint	Controlado por el estado de la tarea
Cambios	Se pasan al siguiente sprint	Se añaden en el tablero en la columna *To Do*
Recomendado	Empresas con equipos maduros en proyectos de más de 1 año	Startups y proyectos de rápido desarrollo

Tabla 4.1: Diferencias entre Scrum y Scrumban.

plejidad o el riesgo sea alto por poder surgir errores de programación inesperados. Para estos casos, los sprints de Scrum no son factibles, dado que los errores/impedimentos que surgirán a lo largo de las tareas son difíciles de determinar y consecuentemente no es posible estimar el tiempo que conlleva cada historia. Por ello, resulta más beneficioso adoptar flujo de trabajo continuo propio de Kanban.

Capítulo Quinto

Comparación entre las metodologías

5.1. Introducción

Este capítulo trata sobre la comparación de las metodologías descritas en los dos capítulos anteriores. Como ya se ha dicho, se llama a todo metodologías cuando algunas de ellas (sobre todo las ágiles) son más bien un conjunto disperso de principios, valores y buenas prácticas. Sin embargo, la comparación tiene interés para explicar cómo abordan las distintas áreas, si lo hacen, de las que se componen la gestión de proyectos.

Hay una primera sección que se compara solamente las metodologías ágiles por ser muy heterogéneas entre sí y para que el lector fije cuales son sus principales diferencias. En la siguiente sección se hace ya una comparación entre las metodologías tradicionales y ágiles.

5.2. Comparación entre las metodologías ágiles

En este apartado se van a comparar las metodologías ágiles más conocidas y usadas: Adaptive Software Development (ASD), Agile Modeling (AM), Crystal, Dynamic Systems Development Method (DSDM), eXtreme Programming (XP), Feature Drive Development (FDD) y Scrum. Para ello se van a evaluar si cubren las distintas fases del ciclo de vida de desarrollo software: concepción del proyecto, especificación de requisitos, diseño, codificación, pruebas unitarias, pruebas de integración, pruebas de sistema, pruebas de aceptación y sistema en servicio. En la comparativa se van a evaluar, para cada fase, 3 aspectos diferentes.

1. **Gestión de proyecto**. ¿Ofrece la metodología directrices a las actividades de gestión del proyecto? Los métodos deberían ser eficientes y para ello se requiere la existencia de actividades de gestión de proyectos que permitan la correcta ejecución de las tareas de desarrollo software.

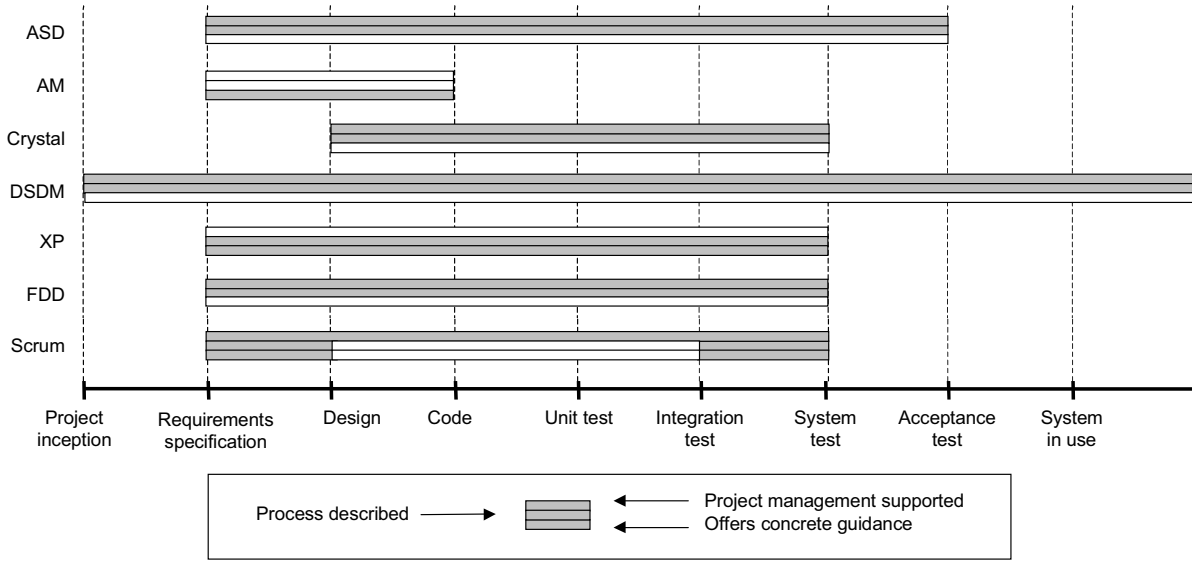

Figura 5.1: Comparativa entre las metodologías ágiles.

2. **Ciclo de vida de desarrollo software**. ¿Qué etapas del ciclo de vida cubre la metodología? ¿Está el proceso descrito?

3. **Abstracción vs concreto**. ¿La metodología se basa en principios abstractos o proporciona orientación concreta? Orientación concreta se refiere a las prácticas, actividades y productos de trabajo que caracterizan y proporcionan orientación sobre cómo se puede ejecutar una tarea específica.

En la figura 5.1 se muestra la comparativa. La primera barra es el aspecto número 1 antes mencionado, la segunda barra para el aspecto 2 y la tercera barra el aspecto 3. Si está en gris es que cubre dicho aspecto y en caso contrario se deja en blanco.

A primera vista se observa que sólo DSDM cubre todas las fases del ciclo de vida y ofrece directrices para la gestión del proyecto pero no proporciona orientaciones concretas para ejecutar las tareas específicas de desarrollo. Si no tuviese esta carencia, sería una metodología bastante completa.

En el polo opuesto se encuentra AM que sólo ofrece guías concretas para las fases de requisitos y diseño sin cubrir el aspecto 1 ni 2. Por lo tanto, es la metodología menos completa de todas y si se usa tendría que ser en combinación con otras si se quiere abarcar todo el ciclo de vida.

En general, casi todas las metodologías (ASD, Crystal, DSDM, FDD y Scrum) tienen carencia en que no dan orientaciones concretas (aspecto 3) para realizar las actividades de desarrollo. Este es uno de los puntos desfavorables más importantes que se le achacan a las metodologías ágiles: no concretar cómo hacer las cosas. Sin embargo, XP sí cubre

este aspecto pero tiene en su contra que descuida totalmente la gestión del proyecto.

Por último, citar Scrum por su notoriedad actual aun a pesar que desde el punto metodológico no cubre todo el ciclo de desarrollo y hay otras mucho más completas. Scrum se centra en las fases más importantes del proyecto dejando en blanco las demás (concepción del proyecto, pruebas de sistema y aceptación y sistema en servicio). Tal vez su éxito se debe a que sí tiene en cuenta la gestión del proyecto y da directrices concretas en dos fases cruciales del proyecto (especificación de requisitos y pruebas de integración). Todo ello hace que sea una metodología útil que combinándola con otra/s (típicamente con XP) se forme una metodología híbrida con gran aceptación por parte del público. Volver a ver la figura 4.1 del capítulo anterior donde se observa que la metodología híbrida Scrum/XP es utilizada un 24 %.

5.3. Comparación entre las metodologías tradicionales y ágiles

El enfoque para la comparación entre metodologías puede ser diverso según qué parámetros se estudien. Aquí se van a comparar atendiendo a los siguientes 3 parámetros.

1. **Costes de los cambios**. ¿Cómo se afrontan los cambios a lo largo del proyecto para las tradicionales vs ágiles?
2. **Retorno de la inversión** (ROI). ¿Cómo es el ROI a lo largo del proyecto para las tradicionales vs ágiles?
3. **Áreas de conocimiento**. ¿Cómo gestiona las distintas áreas de conocimiento del proyecto las tradicionales vs ágiles?

Si representamos los costes de los cambios a lo largo del calendario de un proyecto para las tradicionales vs ágiles se tendría algo parecido a lo dibujado en la figura 5.2. En las fases iniciales del proyecto hay un coste mayor en las ágiles porque se están haciendo periódicamente pequeños incrementos de software funcional mientras que las tradicionales están todavía en las fases de toma de requisitos y diseño que apenas generan costes al hacer cambios. Sin embargo, se observa que el coste en las tradicionales se empieza a disparar en la mitad del proyecto porque ya hay mucho código realizado y cuanto más se acerce el final, el coste generado por los cambios será mayor. Sin embargo, el coste en las ágiles tiene una curva con una pendiente mucho más suave y, si fuera un proyecto realizado idealmente, sería casi una constante.

Si representamos el retorno de la inversión (ROI) a lo largo del proyecto para las tradicionales vs ágiles se tendría algo parecido a lo dibujado en la figura 5.3. En la gráfica de la izquierda se observa el comportamiento de las tradicionales: no hay ningún retorno

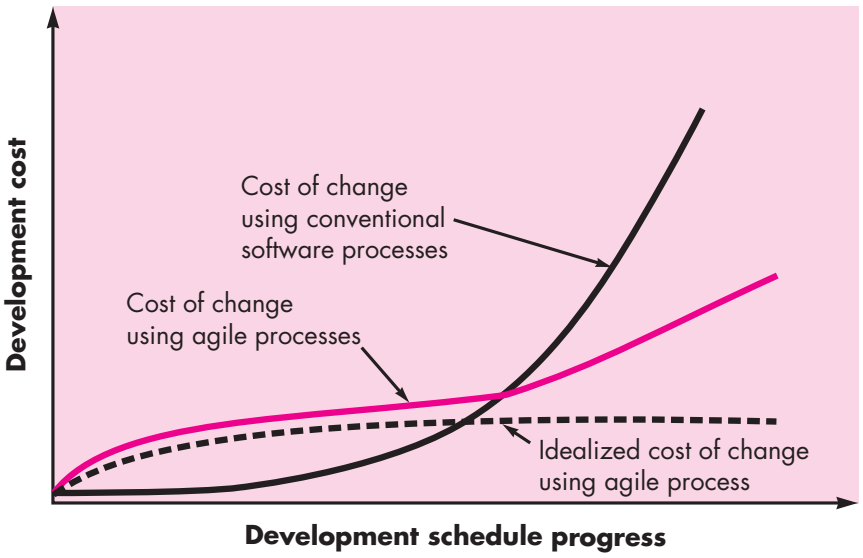

Figura 5.2: Costes de los cambios según las metodologías tradicionales vs ágiles.

monetario hasta que el sistema no está en una fase de despliegue. Sin embargo, en la gráfica de la derecha el comportamiento de las ágiles empieza a haber retornos monetarios a lo largo de todo el proyecto gracias a las pequeños incrementos software funcionales que el cliente puede probar. Estas gráficas no dejan de ser una simplificación de la realidad y las condiciones de pago del proyecto dependerá de las condiciones del contrato que hayan firmado las partes pero un comportamiento plausible es el explicado anteriormente.

Por último, se ha realizado la tabla 5.1 comparando 3 metodologías tradicionales (PMBOK, PRINCE2 e ICB) y 3 ágiles (XP, Scrum, FDD). Los aspectos que se van a comparar son las 10 áreas de conocimiento que se deben gestionar en un proyecto según PMBOK: integración, alcance, plazos, costes, calidad, RRHH, comunicación, riesgos, aprovisionamiento e involucrados. Nótese las siguientes consideraciones para la realización de la tabla comparativa.

- Para PMBOK el análisis es trivial al coger como base su propia estructura. La razón de escoger esta base es porque el autor cree que dichas áreas abarcan completamente las tareas de un proyecto.
- Para PRINCE2 se han analizado sus procesos y sus interrelaciones, enmarcándolas en las áreas más afines.
- Para ICB se han analizado sólo las competencias técnicas, dejando de lado las personales y las contextuales por ser más difíciles de encuadrar al ser más genéricas e involucrar aspectos subjetivos.
- Para las metodologías ágiles se ha intentado encuadrar en las distintas áreas de cono-

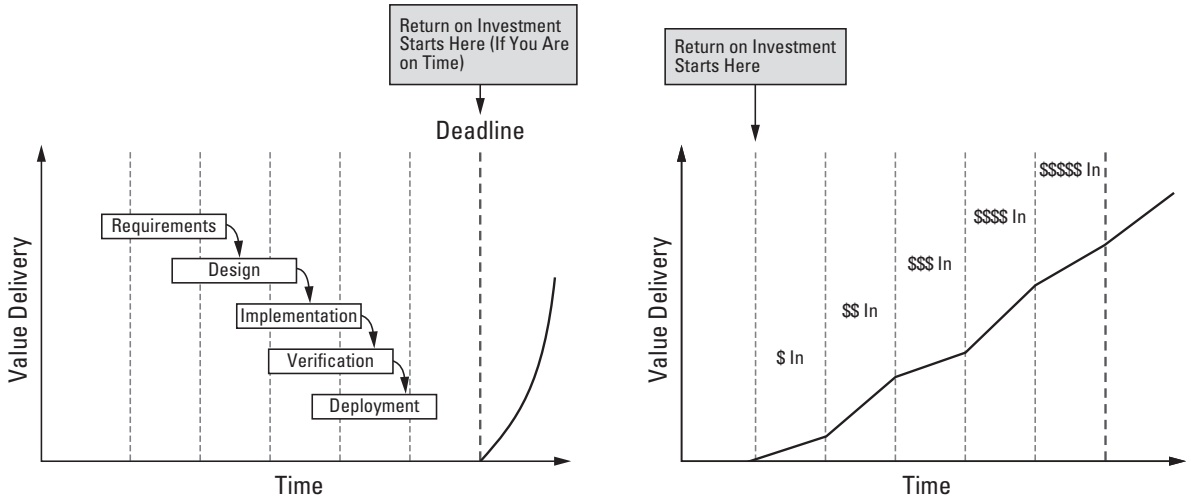

Figura 5.3: Retorno de la inversión (ROI) según las metodologías tradicionales vs ágiles.

cimiento sus principios, valores y buenas prácticas. Al no ser metodologías divididas en áreas si no principios generalistas, es algo subjetivo dónde situarlas.

- Se podría hacer un estudio más completo incluyendo el resto de metodologías pero por cuestiones de extensión y tiempo del TFM se ha optado por seleccionar 6 metodologías representativas del conjunto.

Tabla 5.1: Comparación de las metodologías PMBOK, PRINCE2, ICB, XP, SCRUM y FDD.

ÁREAS	PMBOK	PRINCE2	ICB	XP	SCRUM	FDD
Integración	• Desarrollar acta constitución del proyecto • Desarrollar el plan del proyecto • Dirigir la ejecución del proyecto • Monitorizar y controlar el trabajo del proyecto • Realizar control integrado de cambios • Cerrar el proyecto	• Emprender proyecto: enfocar el proyecto y realizar un resumen • Emprender proyecto: caso de negocio preliminar • Iniciar proyecto: crear el plan del proyecto • Iniciar proyecto: estrategia de gestión de la configuración • Dirigir proyecto: autorizar el inicio • Dirigir proyecto: autorizar el proyecto • Dirigir proyecto: autorizar cierre proyecto	• Éxito en la dirección de proyectos • Requisitos y objetivos del proyecto • Organizaciones de proyectos • Estructuras de proyectos • Arranque • Cierre	• Integración tan pronto y a menudo como sea posible • Propiedad colectiva del código • Medición de la velocidad del proyecto	• Verificar aprobación proyecto y su financiación en la fase de planificación • Validar herramientas e infraestructura en la fase de planificación • Fuerte gestión de cambios con el producto y sprint backlog • Refinamiento arquitectura sistema para apoyar los cambios	• Desarrollo del modelo general del sistema
Involucrados	• Identificar involucrados • Plan de involucrados • Dirigir relación involucrados • Controlar relación involucrados		• Partes involucradas • Comunicación • Controles e informes • Información y documentación			

Tabla 5.1: (continuación)

ÁREAS	PMBOK	PRINCE2	ICB	XP	SCRUM	FDD
Alcance	• Plan del alcance • Recopilar requisitos • Definir alcance • Crear la WBS • Verificar alcance • Controlar alcance	• Iniciar proyecto: documentación inicial del proyecto • Iniciar proyecto: refinar el caso de negocio • Gestión límite etapa: actualizar plan de proyecto • Gestión límite etapa: actualizar caso de negocio • Control etapa: revisar estado de la etapa • Control etapa: revisar estado del paquete de trabajo	• Requisitos y objetivos del proyecto • Alcance y entregables • Cambios • Controles e informes • Información y documentación	• Historias de usuario • Planificación de lanzamiento, pequeñas versiones	• Analizar y construir el modelo esquemático del proyecto • Elaboración la lista de backlog del proyecto • Elaboración del sprint de backlog del proyecto • Definición funcionalidades que se incluirán en cada versión • Seleccionar versión más adecuada para desarrollo inmediato • Revisar el progreso de items backlog asignados	• Analizar y construir el modelo esquemático del proyecto (paso 1) • Construir la lista de funcionalidades por áreas temáticas (paso 2)
Plazos	• Plan de plazos • Definir actividades • Secuenciar actividades • Estimar recursos de actividad • Estimar duraciones de actividad • Desarrollar calendario • Controlar calendario	• Gestión límite etapa: planear siguiente etapa • Gestión límite etapa: reportar final de etapa • Control etapa: tomar acciones correctivas	• Resolución de problemas • Tiempo y fases de los proyectos • Recursos • Controles e informes	• Liberar la planificación • Iterar la planificación	• Fijar la fecha y funcionalidades para cada versión • Iteraciones mensuales	• Determinar secuencia de desarrollo (paso 3) • Asignar paquetes trabajo a los jefes programadores (paso 3) • Asignar clases a los desarrolladores (paso 3)

Tabla 5.1: (continuación)

ÁREAS	PMBOK	PRINCE2	ICB	XP	SCRUM	FDD
Coste	• Plan de costes • Estimar costes • Determinar el presupuesto • Controlar los costes	• Iniciar proyecto: definir controles para el proyecto • Control etapa: reportar aspectos importantes	• Requisitos y objetivos de proyectos • Resolución de problemas • Coste y financiación • Controles e informes		• Estimación del coste de la versión en la fase de planificación	
Calidad	• Plan de calidad • Realizar aseguramiento de la calidad • Realizar control de calidad	• Iniciar proyecto: estrategia de gestión de calidad • Gestión entrega producto: entrega paquete de trabajo • Gestión entrega producto: aceptación paquete de trabajo	• Calidad • Controles e informes	• Énfasis en el testing • Sobre la base de la simplicidad • Uso de estándares de calidad y pruebas	• Revisión y ajuste de las normas con el que el proyecto se acordó • Reunión revisión del diseño • Reunión planificación de sprint • Reunión revisión de sprint. • Scrum diario	• Reuniones de revisión (todos los pasos) • Inspección de código y pruebas unitarias (paso 5)
Recursos humanos	• Plan de RRHH • Adquirir el equipo • Desarrollar el equipo • Dirigir el equipo	• Emprender proyecto: nombrar al ejecutivo y al director del proyecto • Emprender proyecto: diseñar y designar al equipo del proyecto	• Organizaciones de proyectos • Trabajo en equipo • Resolución de problemas • Recursos • Comunicación	• Rotación del personal en varios puestos • Programación en parejas • Buenas condiciones de trabajo (no horas extras)	• Nombramiento del equipo de proyecto para cada versión • Participación del equipo en reuniones de sprint • Participación del equipo en scrums diarios	• Nombrar equipo de diseño (paso 1) • Nombrar equipo de requisitos (paso 2) • Nombrar equipo de planificación (paso 3) • Nombrar equipo para cada funcionalidad (paso 3)

Tabla 5.1: (continuación)

ÁREAS	PMBOK	PRINCE2	ICB	XP	SCRUM	FDD
Comunicaciones	• Plan de comunicaciones • Dirigir las comunicaciones • Controlar las comunicaciones	• Iniciar proyecto: estrategia de gestión de la comunicación	• Información y documentación • Comunicación	• Usar lenguaje no técnico para explicar el proyecto • Cliente siempre disponible • Reuniones diarias • Uso de estándares	• Comunicación de las normas del proyecto al equipo • Reunión revisión del diseño • Reunión planificación de sprint • Reunión revisión de sprint. • Scrum diario	• Reuniones de revisión (todos los pasos)
Riesgos	• Plan de riesgos • Identificar riesgos • Analisis cualitativo del riesgo • Analisis cuantitativo del riesgo • Plan de respuesta a los riesgos • Monitorizar y controlar los riesgos	• Iniciar proyecto: estrategia de gestión de riesgos • Control etapa: capturar y examinar problemas y riesgos • Control etapa: reportar problemas y riesgos	• Riesgos y oportunidades • Resolución de problemas • Cambios • Controles e informes	• Crear prototipo para reducir riesgos	• Evalución de riesgos al inicio del proyecto • Revisar los riesgos en las reuniones de revisión	
Aprovisionamiento	• Plan de aprovisionamientos • Realizar aprovisionamiento • Administrar aprovisionamientos • Cerrar aprovisionamientos		• Resolución de problemas • Tiempo y fases de los proyectos • Recursos • Aprovisionamiento y contratos • Controles e informes			

Capítulo Sexto

Conclusiones finales

6.1. Conclusiones

Este TFM ha conseguido realizar una presentación general de la Ing. de Software, las metodologías de desarrollo de software que hay en la actualidad y una comparación ilustrativa entre ellas. No se ha podido profundidar en las muchas y variadas metodologías que hay tanto como le hubiese gustado al autor. Queda para posteriores trabajos o, incluso, una Tesis Doctoral, el ampliar este estudio con más información dando un enfoque más heterogéneo de la dirección y gestión de proyectos software así como realizar una comparación exhaustiva y desde distintas perspectivas que enriquezca el estudio.

Resumidamente se exponen algunas conclusiones u observaciones relevantes, a juicio del autor, que han quedado patentes mientras se realizaba el TFM.

- Se ha realizado una extensa descripción de por qué los proyectos software son diferentes al resto de proyectos de ingeniería enumerando sus particularidades y principales problemáticas. Esto da una justificación completa al estudio de los métodos de gestión de proyectos software por ser un elemento crucial para alcanzar el éxito.
- Las fronteras entre las metodologías no están delimitadas claramente. Las metodologías aquí llamadas tradicionales tienen características de las ágiles y viceversa. También se ha simplificado llamando metodologías a todas las formas de gestionar proyectos aquí tratadas cuando algunas de ellas no son metodologías propiamente dichas, sino más bien guías de buenas prácticas o directrices para gestionar proyectos.
- Se han encontrado dificultades para realizar la comparación entre las distintas metodologías. Esto es debido a que no es trivial encontrar elementos comunes a todas las metodologías y que, además, las definan completamente. Es complejo encontrar una base cuando se comparan metodologías de naturaleza diversa.
- A lo largo del trabajo se va afianzando la tesis de que no existe una metodología única y perfecta que garantice el éxito a cualquier tipo de proyecto. En el mejor de

los casos existirá una metodología que se adapte al tipo de proyecto, cliente, empresa y equipo que lo va a desarrollar.

- No se puede establecer ninguna superioridad entre las dos familias de metodologías confrontadas. Como ya se ha dicho en el punto anterior, dependerá del entorno del proyecto. Por lo tanto, hay que desterrar la idea de que las metodologías ágiles son la mejor opción siempre. Lo difícil es saber el punto medio recomendable para cada proyecto entre agilidad y la planificación tradicional.
- Una forma de conseguir una metodología que se ajuste a un proyecto sería coger una metodología específica y modificarla adaptándola a las necesidades del proyecto. Además se podría hibridar cogiendo otra/s metodología/s. Por ejemplo, Scrum/XP es una metodología híbrida muy popular per se podría ir más lejos hibridando metodologías de distintas familias. Por ejemplo, PMBOK/XP donde PMBOK suple las carencias en gestión de proyectos que tienen XP.

6.2. Líneas futuras

Vivimos una época emocionante donde el software está cambiando nuestra vidas en todos los aspectos: desde el ámbito laboral hasta la manera de relacionarlos socialmente pasando por los hábitos de consumo. Además, cada vez hay más sistemas críticos, tanto para las personas como para los Estados, controlados por sistemas informáticos: plantas energéticas, distribución de aguas, defensa y seguridad, hospitales, medios de transporte, etc... Estos macrosistemas deben de tener un software bien diseñado y construido porque un fallo tendría graves consecuencias. Consecuentemente, la exigencia en dichos sistemas aumenta cada vez más por lo que la presión para el equipo de desarrollo se incrementa.

Gracias a los avances en otros campos como, por ejemplo, el de los materiales, la electrónica o la medicina permitirán llevar el software a nuevos sectores creando nuevos sistemas que ayuden a las personas. Por tanto, la gestión de proyectos software será una tarea cada vez más crucial. Mientras no se automatice la creación de código y siga siendo un proceso intelectual y creativo realizado por humanos, la labor del director del proyecto seguirá siendo de suma importancia. Todavía queda muy lejos el que el ser humano pueda crear máquinas que tengan la suficiente inteligencia para desarrollar código en base a unas especificaciones enunciadas con la imprecisión y ambigüedad del lenguaje humano.

Como una línea futura para seguir con este trabajo, se podría analizar el uso de las metodologías desde un punto de vista práctico. Es decir, no realizar un trabajo meramente teórico sino que también se consulte bibliografía que recoja casos de éxito y experiencias reales de proyectos en los que se han utilizado determinadas metodologías. Por ejemplo, el SEI tiene artículos interesantes en el que tratan cómo se utilizaron las metodologías ágiles en macroproyectos reales dentro del Department of Defense (USA) (DOD), institución de referencia en la gestión y dirección de nuevas técnicas en proyectos de ingeniería.

El sector del desarrollo de software avanza muy rápido creando nuevas formas de trabajo utilizando las nuevas tecnologías disponibles, adaptándose a las necesidades del cliente y compitiendo con empresas a nivel global. El hombre siempre ha sigo capaz de adaptarse a su entorno y seguro que creará metodologías y herramientas cada vez más eficientes y adaptadas al tipo de proyectos que realice.

Parte II

ANEXOS

Apéndice A

Estándares del IEEE e ISO/IEC sobre gestión de proyectos software

En este apéndice se va a hacer un compendio de los estándares reconocidos por el IEEE y el ISO/IEC, siendo una prueba de que se ha obtenido un amplio consenso. El ISO/IEC JTC 1/SC 7 mantiene casi 200 normas sobre la Ingeniería de Software y el IEEE sobre 50. Las dos organizaciones llevan una década en un programa sistemático para coordinar e integrar sus colecciones.

En particular, aquí se resumirán los estándares que tratan de manera más o menos directa la gestión de proyectos software. Si se quiere consultar el resto de normas que involucra la Ing. de Software (requirements, design, construction, engineering process, engineering models and methods, configuration management, maintenance, quality, engineering professional practice y testing) se puede consultar el *Apéndice B del SWEBOK* ([5]) donde están agrupadas y resumidas. A continuación se enumeran las relaciondas con el Software Engineering Management:

ISO/IEC TR 19759:2005 Software Engineering - Guide to the Software Engineering Body of Knowledge (SWEBOK)

Es un norma de carácter general donde se identifica y describe la suma de conocimiento de la Ing. de Software.

IEEE Std. 1490-2011 Guide - Adoption of the Project Management Institute (PMI) Standard, A Guide to the Project Management Body of Knowledge (PMBOK Guide) - 5th Edition.

La Guía del PMBOK identifica los conocimientos de la Dirección de Proyectos generalmente reconocidos como buenas prácticas. "Generalmente reconocidos" significa que son aplicables a la mayoría de los proyectos, la mayor parte del tiempo y hay consenso sobre su valor y utilidad. "Buenas prácticas" significa que hay acuerdo general en que la aplicación de estas habilidades, herramientas y técnicas puede aumentar

las posibilidades de éxito de muchos proyectos. El PMI considera esta norma como una referencia de gestión del proyecto básica para sus programas y certificaciones de desarrollo profesional.

ISO/IEC/IEEE 16326:2009 Systems and Software Engineering - Life Cycle Processes - Project Management.

Especifica contenido normativo de los planes de gestión de proyectos de tipo software así como proyectos de sistemas intensivos en software. También proporciona asesoramiento sobre la aplicación de un conjunto de procesos de los proyectos que son comunes tanto para el ciclo de vida del software como del sistema.

IEEE Std. 16085-2006 (a.k.a. ISO/IEC 16085:2006) Standard for Systems and Software Engineering - Software Life Cycle Processes - Risk Management.

Define un proceso para la gestión de riesgo en el ciclo de vida, tanto del software como de sistemas. Es una tarea fundamental y crítica determinar continuamente la viabilidad de los planes del proyecto, mejorar la búsqueda e identificación de los posibles problemas que pueden afectar a las actividades del ciclo de vida, la calidad y rendimiento de los productos y también sirve para mejorar la gestión activa de proyectos.

IEEE Std. 15939-2008 Standard Adoption of ISO/ IEC 15939:2007 Systems and Software Engineering - Measurement Process.

Define un proceso de medición aplicable a la gestión de la Ing. de Software y sus sistemas. El proceso se describe a través de un modelo que define las actividades que se requieren para especificar adecuadamente la información requerida, como se aplicarán las medidas y los análisis y cómo determinar si los resultados del análisis son válidos. El proceso de medición es flexible, a medida y adaptable a las necesidades de los diferentes usuarios.

ISO/IEC/IEEE 26511:2012 Systems and Software Engineering - Requirements for Managers of User Documentation.

Los proyectos de software a menudo requieren el desarrollo de documentación de usuario. La gestión del proyecto, por lo tanto, incluye la gestión de dicha documentación. Esta norma especifica los procedimientos para la gestión de la documentación del usuario durante todo el ciclo de vida del software. Se aplica a las personas u organizaciones que producen conjuntos de documentación, a los que realizan un único proyecto con su documentación, a la documentación producida internamente, así como a la documentación contratada por organizaciones externos. Gran parte de la norma es aplicable a la documentación de sistemas que incluyen hardware así como software.

IEEE Std. 1062-1998 Recommended Practice for Software Acquisition.

Conjunto de prácticas útiles aplicables durante uno o más pasos en un proceso de adquisición de software. Se puede aplicar al software que se ejecuta en cualquier sistema informático, independientemente del tamaño, complejidad o criticidad del software.

ISO/IEC/IEEE 26512:2011 Systems and Software Engineering - Requirements for Acquirers and Suppliers of User Documentation.

Define el proceso de documentación desde el punto de vista del cliente como del proveedor. Cubre los requisitos de información utilizados en la adquisición de productos software: el alcance, documento de especificaciones, seguimiento del trabajo, solicitud de cambios. Incluye requisitos para las salidas de documentos principales. También se analiza el uso de un plan de gestión de la documentación y un plan de documentos a medida que surjan en los procesos de adquisición y suministro. Es independiente de las herramientas de software que pueden ser utilizados para producir la documentación. Gran parte de la norma es aplicable a la documentación de sistemas que incluyen hardware así como software.

ISO/IEC/IEEE 26515:2012 Systems and Software Engineering - Developing User Documentation in an Agile Environment.

Especifica la manera en que la documentación del usuario se puede desarrollar en los proyectos con desarrollo ágil. Está diseñado para ser utilizado en todas las organizaciones que utilizan el desarrollo ágil o están considerando la implementación de sus proyectos utilizando estas técnicas.

Apéndice B

Manifiesto Ágil

Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

Individuos e interacciones sobre procesos y herramientas

Software funcionando sobre documentación extensiva

Colaboración con el cliente sobre negociación contractual

Respuesta ante el cambio sobre seguir un plan

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Firmado el 17 de febrero de 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*.

Tras los cuatro valores descritos, los firmantes redactaron los siguientes, como los principios que de ellos se derivan:

- Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor.
- Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente.
- Entregar con frecuencia software que funcione, en periodos de un par de semanas hasta un par de meses, con preferencia en los periodos breves.

- Las personas del negocio y los desarrolladores deben trabajar juntos de forma cotidiana a través del proyecto.
- Construcción de proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realicen la tarea.
- La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la conversación cara a cara.
- El software que funciona es la principal medida del progreso.
- Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica enaltece la agilidad.
- La simplicidad como arte de maximizar la cantidad de trabajo que no se hace, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos que se autoorganizan.
- En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia.

El Manifiesto Ágil está disponible en http://agilemanifesto.org

Referencias

[1] *A guide to the project management body of knowledge (PMBOK guide), 5th edition*. Project Management Institute, 2013. ISBN-13: 978-1-935589-67-9.

[2] *Software Extension to the PMBOK Guide*. Project Management Institute, 2013. ISBN-13: 978-1-6282-5013-8.

[3] *IPMA Competence Baseline (ICB), version 3.0*. International Project Management Association, 2006. ISBN: 0-9553213-0-1.

[4] *Managing Succesful Projects with PRINCE2*. The Stationery Office, 2009. ISBN: 978-0-11-331059-3

[5] P. Bourque and R.E. Fairley, eds., *Guide to the Software Engineering Body of Knowledge (SWEBOK), version 3.0*. IEEE Computer Society, 2014. ISBN-13: 978-0-7695-5166-1. Disponible libre y gratuitamente en http://www.computer.org/portal/web/swebok

[6] *UNE-ISO 21500, Directrices para la dirección y gestión de proyectos*. AENOR, 2013.

[7] Ken Schwaber and Jeff Sutherland, *La Guía Definitiva de Scrum: Las Reglas del Juego*. 2013.

[8] ALLAN KELLY. *Changing software development: learning to be agile*. John Wiley & Sons Ltd, 2008. ISBN-13: 978-0-470-51504-4.

[9] DEAN LEFFINGWELL. *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley, 2011. ISBN-13: 978-0-321-63584-6.

[10] JAMES SHORE and SHANE WARDEN. *The art of agile development*. O'Reilly, 2008. ISBN-13: 978-0-596-52767-9.

[11] MIKE COHN. *Agile estimating and planning*. Prentice Hall, 2005. ISBN-13: 978-0-131-47941-8.

[12] ROGER S. PRESSMAN. *Software engineering : a practitioner's approach, 7th edition*. Mc Graw Hill, 2010. ISBN: 978-0-07-337597-7.

[13] IAN SOMMERVILLE. *Software Engineering, 9th edition*. Addison-Wesley, 2011. ISBN-13: 978-0-13-703515-1.

[14] POLLYANNA PIXTON, PAUL GIBSON and NIEL NICKOLAISEN. *The agile culture*. Addison-Wesley, 2014. ISBN-13: 978-0-321-94014-8.

[15] THE STANDISH GROUP. *CHAOS Manifesto 2013*. 2013. http://www.standishgroup.com

Artículos del Institute of Electrical and Electronics Engineers (IEEE):

[16] Aitken, Ashley; Ilango, Vishnu, *A Comparative Analysis of Traditional Software Engineering and Agile Software Development*. System Sciences (HICSS), 2013 46th Hawaii International Conference.

[17] Larman, C.; Basili, V.R., *Iterative and incremental developments. a brief history*. Computer, vol.36, no.6, pp.47,56, June 2003.

[18] Abrahamsson, P.; Warsta, J.; Siponen, M.T.; Ronkainen, J., *New directions on agile methods: a comparative analysis*. Software Engineering, 2003. Proceedings. 25th International Conference.

[19] Coram, M.; Bohner, S., *The impact of agile methods on software project management*. Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops.

[20] Sulaiman, T.; Barton, B.; Blackburn, T., *AgileEVM - earned value management in Scrum Projects*. Agile Conference, 2006.

[21] Fitsilis, P., *Measuring the Complexity of Software Projects*, Computer Science and Information Engineering, 2009 WRI World Congress on , vol.7, no., pp.644,648, 2009.

[22] J. Laurenz Eveleens and Chris Verhoef, *The rise and fall of the Chaos report figures*. IEEE Computer Society, 2010.

Artículos de otras fuentes:

[23] Boehm, B.W., *A View of 20th and 21st Century Software Engineering*, ACM Press, In Proc. ICSE?06, 2006.

[24] A. B. M. Moniruzzaman and Syed Akhter Hossain, *Comparative Study on Agile software development methodologies*. Journal CoRR, volume abs/1307.3356, 2013.

[25] Fitsilis, P., *Comparing PMBOK and Agile Project Management software development processes*. Computer Science and Information Engineering, 2009 WRI World Congress on, vol.7, no., pp.644,648, 2009.

Trabajos académicos:

[26] Ramiro Concepción Suárez, *Metodología de gestión de proyectos en las Administraciones Públicas según ISO 10006*, Tesis doctoral, Doctorado Interuniversitario en Dirección de Proyectos, Universidad de Oviedo, Octubre 2007.

[27] Pilar Rodríguez González, *Estudio de la aplicación de metodologías ágiles para la evolución de productos software*. Trabajo Fin de Máster, Facultad de Informática, Universidad Politécnica de Madrid, Septiembre 2008.

[28] Pedro Sáez Martínez, *Identificación y valoración de técnicas ágiles de gestión de proyectos software*. Trabajo Fin de Máster en Dirección de Proyectos, Universidad de Oviedo, Julio 2013.

[29] Luis Álvarez Puertas, *Oficina de Gestión de Proyectos Ágil: control y seguimiento de proyectos ágiles*. Trabajo Fin de Máster en Dirección de Proyectos, Universidad de Oviedo, Junio 2013.

Páginas web consultadas:

[30] Página de International Organization for Standardization (ISO): http://www.iso.org

[31] Página de Institute of Electrical and Electronics Engineers (IEEE): http://www.ieee.org

[32] Página de la Institute of Electrical and Electronics Engineers (IEEE) Computer Society: http://www.computer.org

[33] Página de Asociación Española de Normalización y Certificación (AENOR): https://www.aenor.es

[34] Página de Software Engineering Institute (SEI): http://www.sei.cmu.edu

[35] Página de Project Management Institute (PMI): http://www.pmi.org

[36] Página de International Project Management Association (IPMA): http://www.ipma.ch

[37] Página de Association for Project Management (APM): http://www.http://www.apm.org.uk

[38] Página de Project Management Association of Japan (PMAJ): http://www.pmaj.or.jp/ENG

[39] Página de Agile Alliance: http://www.agilealliance.org

[40] Página de Agile-Spain: http://agile-spain.org

[41] Página de Scrum Alliance: https://www.scrumalliance.org/

[42] Página de DSDM Consortium: http://www.dsdm.org

Repositorios:

[43] Repositorio de artículos científicos http://sciencedirect.com

[44] Repositorio de la Universidad de Oviedo: http://digibuo.uniovi.es/dspace

[45] Repositorio del Institute of Electrical and Electronics Engineers (IEEE): http://ieeexplore.ieee.org

[46] Repositorio del Computing Research Repository (CoRR): http://arxiv.org

[47] Vocabulario de Sistemas e Ing. de Software del IEEE Standard 24765:2010: http://www.computer.org/sevocab