9. Técnicas Ágiles de Desarrollo de Software

9.1. Introducción y objetivos

En este tema se estudian otras técnicas ágiles utilizadas en el desarrollo de *software,* además de *scrum*. Se estudia a fondo XP (*eXtreme Programming*), seguramente una de las técnicas ágiles más conocidas y estudiadas, que se encarga de llevar «al extremo» los principios del desarrollo iterativo generando nuevas versiones mejoradas del *software* varias veces al día, enviando el cliente cada poco tiempo nuevos incrementos funcionales del *software* y ejecutando todas las pruebas en cada nueva versión, aceptando la versión si todas las pruebas se ejecutan correctamente.

Junto a XP, se estudian los principios de lo que se conoce como *lean,* con su aplicación para el desarrollo de *software; kanban,* como un añadido en proyectos ágiles y su gran potencial como complemento de *scrum*; y otras técnicas como *crystal, feature driven development, adaptive software development, dynamic systems development method*o *scaled agile framework*.

Con el estudio de este tema pretendemos alcanzar los siguientes objetivos:

* Comprender la filosofía general de XP como caso paradigmático y bien conocido de aplicación de una técnica ágil.
* Conocer los valores, roles y prácticas recomendadas por XP.
* Conocer el objetivo, principios y características de la cultura *lean* y de la técnica *lean software development.*
* Conocer *kanban* y su combinación con *scrum*.
* Comprender y conocer otras técnicas ágiles como alternativas a *scrum* o XP.

9.2. eXtreme Programming

El primer proyecto desarrollado con programación extrema (*eXtreme Programming,* XP) comenzó el 6 de marzo de 1996 (Wells, 2013), y desde entonces ha demostrado su utilidad en infinidad de proyectos y en una gran variedad de industrias y tamaños de equipos de desarrollo.

La principal razón de su éxito radica en el énfasis que hace en la satisfacción del cliente y el trabajo en equipo. En el equipo de trabajo aparecen integrados todos los roles principales (managers, clientes y desarrolladores) en un proceso de comunicación de igual a igual, que se realiza, preferiblemente, siempre de manera oral.

En la Figura 1 se muestra un esquema del modelo de proceso general de XP como evolución del modelo en cascada.

Figura 1. Evolución del tradicional modelo en cascada (a), pasando por los modelos basados en ciclos iterativos de más corta duración (b), hasta el modelo de Programación extrema, que mezcla en cada iteración las cuatro actividades fundamentales, buscando pequeños incrementos del sistema a lo largo del proceso de desarrollo. Fuente: Beck, 1999.

En el modelo tradicional en cascada, el proceso comienza con la interacción con clientes y usuarios, obteniendo información sobre las características del sistema deseado. A continuación, se diseña el sistema, se implementa el código y, finalmente, se realizan las pruebas para garantizar que el sistema entregado es el correcto. Esto presenta, en ocasiones, algunos inconvenientes: o bien el usuario no conoce o no sabe expresar sus necesidades, o cambia de opinión, o simplemente desde el equipo no se entienden bien estas necesidades, de manera que el sistema final no es el correcto.

Cuando los problemas se detectan tarde puede resultar técnica o económicamente difícil solucionarlos.

Una manera de mejorar el proceso consiste en la adopción de un modelo iterativo que descompone la duración total del proyecto en varios ciclos consecutivos, respetando las mismas fases en cada uno de ellos. Este modelo (Figura 1.b) permite detectar con mayor antelación posibles errores, y resulta más flexible y adaptable al cambio.

Con la programación extrema (Figura 1.c) se lleva al límite este principio reduciendo al máximo la duración de las iteraciones. Se plantea un modelo con iteraciones muy cortas, con fases de diseño, implementación y pruebas solapadas entre sí, dedicando a las labores de análisis en cada iteración el tiempo estrictamente necesario para resolver los problemas correspondientes a ese incremento concreto. Se evita así el planteamiento de un diseño monolítico completo inicial del sistema, puesto que se asume que posibles cambios pueden hacer que este esfuerzo resulte inútil.

Cabe destacar que XP considera dedicar un tiempo a labores de análisis antes del comienzo del proyecto, definiendo de la manera más precisa posible el funcionamiento esperado del sistema. Beck propone una duración mínima de un mes para esta tarea, suponiendo un proyecto de diez personas de un año de duración (Beck, 1999). El resultado es una especificación inicial del sistema materializada en un conjunto de historias de usuario que redacta el cliente. A partir de ahí arranca el proyecto que se descompone en iteraciones, la implementación de cada una de las tareas y las correspondientes entregas *(releases).*

El flujo general de un proyecto en XP es el siguiente:

* El cliente elige el subconjunto más pequeño posible de historias de usuario, con coherencia interna y priorizadas en función del valor que aportan sobre el producto final, y define el alcance de cada *release*. Para estimar la cantidad de historias que es posible implementar, dialoga con el equipo de desarrollo en lo que se conoce como el «juego de la planificación» (mecanismo fundamental de planificación en XP).
* Cada *release* se divide en una o más iteraciones. En cada iteración se escoge un grupo de historias asociadas a la *release* y cada una de ellas se descompone en tareas que se asignan a los miembros del equipo.
* La implementación de las tareas comienza con la definición de las pruebas unitarias, que es posiblemente la práctica más importante que se encuentra en el corazón de XP. La definición de las pruebas, el diseño del incremento y la codificación hasta que todas las pruebas sean superadas son operaciones que se realizan siempre en parejas de programadores. En paralelo, el cliente está encargado de definir las pruebas de aceptación que validarán la superación de la iteración considerada y de la *release* en su conjunto.

El ciclo de vida en XP

La programación extrema propone cuatro actividades estructurales que definen el proceso de desarrollo: planificación, diseño, codificación y pruebas. Además, se sigue un enfoque eminentemente orientado a objetos, aspecto que se tiene en cuenta en las labores de diseño y codificación.

Un proceso inicial y necesario es la captura de requisitos, que se concreta formalmente en historias de usuario. Estas historias de usuario:

* Capturan los requisitos definidos por el cliente y contienen funcionalidades que aportan valor al producto desde el punto de vista del negocio y de los usuarios finales.
* Son creadas por el cliente y sirven como base para la definición de las pruebas de aceptación del sistema.
* Su propósito es similar al de los casos de uso, pero están más orientadas a realizar estimaciones de complejidad y tiempo de implementación durante la fase de planificación de la *release* (Wells, 1999). Este tiempo suele ser de una a tres semanas, si resulta superior es síntoma de que la historia debe descomponerse.
* Las escribe el propio cliente, evitando el empleo de lenguaje técnico y, generalmente, se escribe cada una en una tarjeta numerada, de manera que cada historia queda identificada.

En la Figura 2 se muestra un ejemplo sencillo de historia de usuario para un sistema de gestión de citas médicas en una clínica. La funcionalidad aparece descrita de manera precisa y comprensible en una única frase. En la parte posterior de la tarjeta, se puede incluir información adicional sobre la funcionalidad y su implementación esperada (Figura 3).

Figura 2. Ejemplo de cara frontal de una tarjeta de historia de usuario. Fuente: elaboración propia.

Figura 3. Ejemplo de cara posterior de una tarjeta de historia de usuario. Fuente: elaboración propia.

En la Figura 4 se muestra un esquema general del modelo de proceso en XP. También se representan algunas de las prácticas y herramientas recomendadas, que serán descritas en los siguientes apartados.

Figura 4. Modelo de proceso de Programación Extrema. Fuente: Pressman, 2010.

Fase de planificación

Se trata de planificar la siguiente entrega *(release)* y la siguiente iteración dentro de cada *release.*

El cliente prioriza aquellas historias que aportan mayor valor al negocio desde el punto de vista del cliente y negocia con el equipo de desarrollo su implementación en lo que se conoce como el juego de la planificación. En este punto, los programadores analizan las historias y las descomponen en tareas individuales que son necesarias para su implementación.

Una vez definidas las tareas, se asignan a los diferentes programadores que, a partir de este momento, trabajarán en parejas definiendo las pruebas unitarias, diseñando la solución, implementando e integrando el nuevo código.

Finalmente, podemos considerar que en XP las tareas de planificación constituyen una herramienta temporal, efímera, adaptable y que se realizan en los tres niveles fundamentales (Wells, 2013) que se resumen en la Tabla 1.

En la columna denominada «tiempo» se ha indicado la escala temporal habitual en la que se planifica cada uno de los niveles.

Tabla 1. Niveles de planificación en XP. Fuente: elaboración propia.

Descomposición de una historia de usuario en tareas

Una posible descomposición de la historia representada en la Figura 2 podría ser la siguiente:

* Diseñar la interfaz de usuario para inserción de nuevas citas (ocho días).
* Construir el modelo de datos que representa una cita individual y actualizar la base de datos (un día).
* Escribir la lógica del controlador que relaciona la interacción del usuario con el calendario y la modificación del modelo (dos días).

En este caso, la suma de horas correspondientes a estas tres tareas no coincide con la estimación inicial para la historia en su conjunto. Esto es algo comprensible, pues, en este caso, estamos realizando un análisis mucho más fino y, además, la previsión se realiza teniendo en cuenta las condiciones actuales del proyecto, en cuanto a grado de avance y recursos disponibles.

Para solucionar esta discrepancia encontramos dos alternativas:

* Adelantar el análisis y la descomposición en tareas en el momento en que se define cada historia de usuario, de manera que podamos asignar una estimación más precisa en la tarjeta. Este enfoque tiene dos problemas: exige adelantar actividades que pueden no ser necesarias (por ejemplo, porque la historia es modificada o simplemente desaparece) y, además, no considera posibles dependencias de implementación con otras tareas, que pueden estar más claras en la iteración concreta.
* Utilizar puntos de historia en las tarjetas de historia de usuario. Esta es otra manera de estimar la complejidad de implementación perfectamente válida, que simplemente pone en relación unas historias con otras respecto a la previsión de la dificultad de su implementación. Cumple en el objetivo de comparar las historias entre sí, pero evita la necesidad de ofrecer una estimación temporal precisa.

Diseño

En la fase de diseño, eminentemente orientada a objetos, es habitual la utilización de tarjetas CRC (clase-responsabilidad-colaborador). En la Figura 5 se muestra un ejemplo de tarjeta CRC asociado al ejemplo de la historia de usuario «creación de nueva cita».

Figura 5. Ejemplo de tarjeta CRC para la clase que gestiona las citas médicas. Fuente: elaboración propia.

En puridad, la fase de diseño no es simplemente la primera en cada iteración, sino que las labores de diseño se realizan continuamente, incluso durante el proceso de codificación, aplicando así las prácticas recomendadas de mantener el diseño lo más sencillo posible y la refactorización, como estudiaremos más adelante.

Codificación

El verdadero punto de arranque de la implementación en cada iteración es la definición de las pruebas unitarias. La propia definición de las pruebas ayuda a los desarrolladores a comprender mejor la tarea, a analizarla en profundidad y a plantear cualquier posible duda al cliente que, como veremos, está integrado en el propio equipo de desarrollo. La culminación de la implementación de la tarea será precisamente la comprobación de que las pruebas asociadas se ejecutan satisfactoriamente.

Todo el proceso de desarrollo aplica la práctica de programación en parejas y, a medida que las tareas se finalizan, el código se integra en el repositorio general, ejecutándose todas las pruebas para comprobar que las modificaciones no han introducido ningún tipo de problema o error.

Pruebas

Como ya se ha comentado, las pruebas unitarias son el elemento fundamental que guía y valida todo el proceso de desarrollo. Mientras que estas pruebas permiten comprobar la correcta implementación de cada una de las tareas y, por tanto, dan por finalizada la labor de los desarrolladores, se ejecutan en último momento las pruebas de aceptación, definidas por el cliente, que validan que la historia de usuario en su conjunto ha sido correctamente implementada.

Los cinco valores básicos

Beck establece cinco valores básicos que guían todas las actividades dentro de un proyecto desarrollado bajo XP (Beck, 2000):

* Comunicación: este es el primer valor propuesto por Beck, y ello refleja la importancia que se da a los procesos de comunicación en XP y en los procesos ágiles en general.

Se parte del reconocimiento de que muchos de los problemas encontrados en los proyectos se deben a fallos en la comunicación. Fallos en la comprensión de los requisitos al producirse malentendidos entre el cliente y un programador o analista, fallos en la comprensión de un cambio en el diseño o, simplemente, la ausencia de comunicación del cambio, etc. Estos problemas de comunicación, a su vez, pueden estar producidos por diversos factores, desde problemas interpersonales hasta una pobre gestión de la documentación, pasando por la variabilidad de estados de ánimo de los programadores y su capacidad para absorber información. A fin de cuentas, todos somos seres humanos.

La programación extrema favorece las comunicaciones interpersonales antes que la documentación por escrito, siempre que resulte posible. Para ello, introduce una serie de prácticas que, como veremos, no pueden ser aplicadas sin que exista comunicación de por medio (las pruebas unitarias, la programación en parejas, la involucración del cliente o la estimación de esfuerzos en el juego de la planificación).

* Simplicidad: la idea general consiste en construir aquello que como desarrolladores se nos pide, pero nada más. El objetivo es maximizar el rendimiento de la inversión realizada, utilizando los recursos estrictamente necesarios para conseguir satisfacer a nuestro cliente.

En términos prácticos, esto se traduce en realizar pequeños incrementos en la construcción del sistema final, abordando posibles problemas a medida que estos se presenten. Se trata también de eliminar elementos innecesarios en el proceso de construcción del *software,* manteniendo el diseño lo más sencillo posible.

En XP se considera que es preferible hacer hoy algo simple, aunque sea necesario modificarlo o mejorarlo mañana, que anticiparnos construyendo algo más complejo con características que pueden no ser necesarias.

Valorar la comunicación ayuda a que el equipo tenga claro qué debe hacerse y, sobre todo, qué no debe hacerse favoreciendo la simplicidad. Del mismo modo, un sistema sencillo facilita la comunicación, primero porque es más sencillo de explicar y segundo porque puede permitir un equipo de desarrollo más reducido.

* Retroalimentación: en XP se considera el optimismo como uno de los peligros inherentes a la práctica de la programación, mientras que el *feedback* es su tratamiento. Esta retroalimentación se produce a través de diferentes canales (el propio *software,* el cliente y los usuarios, y el resto de los miembros del equipo), y a diferentes niveles:
  + Corto plazo: los programadores dialogan con el cliente, analizando sus historias, estimando el esfuerzo necesario y escribiendo las pruebas que les ayudan a comprender la lógica del sistema y a analizar sus debilidades.
  + Medio y largo plazo: las pruebas funcionales permiten realizar un seguimiento de la evolución del sistema, detectando posibles problemas al introducir nuevas funcionalidades, y la pronta puesta en producción del sistema permite obtener información de la respuesta de los usuarios.

Como vemos, la retroalimentación está muy relacionada con los valores de comunicación y simplicidad.

Una buena comunicación facilita la retroalimentación, mientras que la retroalimentación proporciona información que enriquece el proceso de comunicación.

Por otro lado, es más fácil definir pruebas para sistemas sencillos, mientras que el proceso de definición de pruebas puede ayudar a razonar sobre aquellos elementos que no son imprescindibles.

* Valentía: este valor está estrechamente relacionado con la competitividad del mundo actual y la necesidad de ser permanentemente mejores que la competencia. Ello obliga, en ocasiones, a tomar decisiones drásticas; la modificación de la arquitectura ante un fallo o una vulnerabilidad puede hacer que muchos test dejen de funcionar. En ocasiones será necesario probar varias alternativas de implementación para encontrar la óptima, lo cual puede obligar a eliminar cientos de líneas de código inservibles u obsoletas.

Este coraje, valentía o disciplina, como prefiere llamarlo Pressman (2010), facilita que cada desarrollador tome en cada momento la decisión adecuada.

El valor del coraje adquiere pleno significado en combinación con los tres anteriores:

* + La comunicación permite encontrar nuevas oportunidades de mejora en el código.
  + La simplicidad favorece que podamos arriesgar más al modificar sistemas que son sencillos.
  + El coraje favorece la simplicidad porque anima a no desaprovechar cualquier oportunidad de simplificación que encontremos.
  + La retroalimentación nos permite ser más arriesgados, al saber que cualquier modificación realizada se validará inmediatamente a partir del conjunto de pruebas previamente definidas.
* Respeto: los cuatro valores anteriores se articulan en torno a la idea del respeto, tanto entre los participantes del proyecto como entre los participantes y el proyecto en sí mismo.

El respecto es un valor fundamental y necesario en una técnica que promueve la comunicación, el trabajo en equipo y el compromiso en la producción del sistema de la mayor calidad posible para el cliente.

A su vez, estos valores están relacionados con las prácticas que veremos a continuación.

Las doce prácticas básicas en XP

La programación extrema propone una serie de prácticas de programación que tienen en cuenta los valores en los que se apoya la metodología e intentan plasmarlos en técnicas concretas.

* El juego de la planificación: se trata de determinar de manera rápida el alcance de cada iteración combinando las prioridades de negocio con la factibilidad técnica en un proceso dialogado.

En cada iteración es el cliente quien decide cuál será el alcance de la implementación y fija los plazos basándose en las estimaciones realizadas por el equipo de desarrollo. Los programadores, por su parte, implementan solo aquellas funcionalidades definidas en las historias incluidas en la iteración.

En la Tabla 2 se resumen las principales consideraciones que deben evaluarse en este proceso, tanto por parte de los encargados de negocio como por parte de los miembros del equipo técnico, resultando evidente que ninguno de ellos por sí solo puede determinar la planificación de una iteración.

Tabla 2. Decisiones que deben afrontar los participantes del juego de planificación durante el diálogo que establecen. Fuente: adaptado de Beck, 2000.

* Las pequeñas entregas: se busca poner el sistema en producción cuanto antes, priorizando aquellas funcionalidades que aportan más valor. A partir de ese momento, se continúa mejorando el sistema, introduciendo cada vez un pequeño conjunto de mejoras en iteraciones muy cortas.

Se debe tener en cuenta que la nueva *release* debe tener sentido, constituir una aportación significativa y no debe implementar funcionalidades a medias, aunque con ello se consiga acortar el ciclo de desarrollo.

* La metáfora del sistema: se trata de guiar todo el proceso de desarrollo proporcionando una historia o descripción sencilla que captura el funcionamiento general del sistema, sus objetivos y ambición, de manera que todos los participantes puedan comprenderla.

Una famosa metáfora es la del «escritorio» de los modernos sistemas operativos con interfaz gráfica de usuario. Representa lo que, desde un enfoque más técnico, consideraríamos la arquitectura general del sistema, destacando los elementos fundamentales del producto al más alto nivel, pero omitiendo el vocabulario técnico, de manera que resulte comprensible para participantes con formación técnica o sin ella.

La metáfora del sistema

La metáfora es una herramienta poderosa para explicar la funcionalidad general de un sistema e incluso las principales características de su arquitectura, de manera que resulte fácilmente comprensible para cualquiera.

Un ejemplo típico es el de un servicio de atención al cliente *(call center)*. En este caso, la solución de las necesidades de un cliente pueden requerir dar un número determinado de pasos, hablando con diferentes asistentes y técnicos, humanos o virtuales. En este caso, el símil puede establecerse con una cadena de montaje física, donde existen distintas estaciones de trabajo con técnicos especializados en resolver problemas concretos.

* El diseño simple: el sistema debe mantener un diseño tan simplificado como sea posible en un momento dado, y cualquier complejidad adicional debe eliminarse en el momento en que sea descubierta.

El diseño correcto para el sistema en un momento dado es aquel que (Beck, 2000):

* + Permite ejecutar todos los test de manera satisfactoria.
  + No contiene lógica duplicada.
  + Contiene el menor número posible de clases y métodos.
* *Test-driven development* (desarrollo dirigido por pruebas): los programadores comienzan definiendo las pruebas unitarias del sistema, proceso que les permite comprender mejor los requisitos y sirve como hilo conductor del proceso de desarrollo.

El cliente está encargado de construir las pruebas funcionales que servirán para validar que las historias han sido correctamente implementadas.

En el contexto de XP, cualquier característica que no tenga al menos una prueba automatizada asociada,simplemente no existe. Esta definición de pruebas inicial aumenta la confianza que, tanto clientes como desarrolladores, tienen en el código y, al mismo tiempo aumenta la seguridad para afrontar posibles cambios imprevistos. Si la introducción del cambio causa algún problema en características previas, es posible detectarlo de manera prematura.

* La refactorización: es labor rutinaria de los programadores la reestructuración del sistema respetando la funcionalidad implementada, buscando siempre la simplicidad y flexibilidad y, por tanto, facilitando la comunicación y la retroalimentación de las características del sistema y su diseño.

Al mismo tiempo que se introducen nuevas características, se reflexiona sobre la posibilidad de reestructurar el programa, de manera que la ampliación resulte sencilla y, una vez implementada la funcionalidad, se busca el refinamiento y simplificación del código.

La refactorización del *software* es una técnica de ingeniería que trata de mejorar la estructura del código respetando su funcionalidad.

Martin Fowler, mantiene dentro de su página web (Fowler, 2022) un catálogo de ejemplos de refactorización. Uno de ellos es el conocido como «extracción de un método», que trata de agrupar un conjunto de líneas de código relacionadas entre sí en un único método, cuyo nombre describe la funcionalidad incluida (Fowler, 1999). De esta manera, el código no solo resulta más expresivo, sino que se permite la reutilización de esta función en otros lugares sin necesidad de repetir el código completo.

* La programación en parejas: idealmente, cualquier actividad de implementación de código la realizan dos programadores sentados en la misma máquina *(pair programming).*Mientras que uno de ellos codifica o diseña, el otro supervisa. El programador que escribe el código piensa en la mejor implementación mientras escribe, en tanto que el otro puede adoptar una postura más estratégica, reflexionando sobre las posibles implicaciones de la modificación o alternativas de mejora.

Los emparejamientos se modifican frecuentemente, cada pocas horas, de manera que esta práctica también facilita la comunicación y consigue que todos los miembros del equipo estén al tanto de todas las características del sistema.

* La propiedad del código compartido: cualquier miembro del equipo de desarrollo tiene la potestad y autoridad para modificar cualquier parte del código que estime oportuno, si en ello ve una oportunidad de mejora. De esta manera, todo el equipo es responsable del sistema en su conjunto. Esto tiene al menos dos implicaciones:
  + Todos los miembros del equipo deben tener acceso a todas las partes del código. Para ello es necesaria la utilización de algún sistema de control de versiones que permita que en todo momento cualquier participante pueda tener versiones actualizadas del código fuente y compartir las modificaciones que realice.
  + Todos los miembros del equipo tienen una visión general del código producido. No es necesario que todos ellos conozcan cada parte con la misma profundidad, pero al menos todos tienen algún nivel de conocimiento sobre cada componente.
* La integración continua: se debe disponer de la configuración apropiada para facilitar la integración y construcción del sistema (incluida la ejecución de las pruebas) tantas veces como sea necesario y, al menos, cada vez que una tarea sea finalizada. De esta manera, todo el nuevo código con sus modificaciones se integra y verifica cada pocas horas (al menos cada día) e, idealmente, se integra cada vez que un programador (o pareja de programadores) finaliza una tarea, por lo que resulta más obvio en quién recae la responsabilidad del problema, en caso de que algún test quede roto.
* La semana de 40 horas: una regla fundamental es no trabajar cada semana más de 40 horas y, por supuesto, no hacer horas extras o, al menos, no hacerlas durante dos semanas seguidas. Así, vemos cómo, desde el punto de vista teórico, XP se preocupa por el bienestar del programador (nombre que, en ocasiones, recibe esta práctica).

De hecho, el propio Beck (2000) hace referencia en su libro a la diferencia de culturas entre las empresas europeas y americanas. Se lamenta de que mientras en el viejo continente es habitual que los empleados tomen dos o tres semanas seguidas de vacaciones, en las empresas americanas rara vez se descansa más de algunos pocos días seguidos. Beck prefiere la práctica europea.

* La involucración del cliente: una práctica muy recomendada es la inclusión del cliente dentro del equipo y, si es posible, de un usuario *(cliente in situ)*. Se trata de que esté disponible en cualquier momento para resolver cualquier tipo de cuestión o incertidumbre.

Esta práctica tiene el inconveniente de que el usuario involucrado desatiende sus ocupaciones habituales para atender al equipo respondiendo dudas cotidianas. Se trata de buscar la manera de que el futuro usuario del sistema pueda continuar con su actividad laboral, al mismo tiempo que se encuentra disponible para las necesidades del proyecto. Con ello se persigue una mejor calidad del resultado final y la rápida resolución de cualquier tipo de incertidumbre por parte del equipo técnico.

* Empleo de estándares de programación: es importante que todo el proceso de codificación se ajuste a estándares, dependientes del lenguaje y plataforma utilizados, lo cual redunda en una mejor compresión y mantenibilidad del código.

Esta práctica se vuelve especialmente necesaria en el contexto de la aplicación de la programación en parejas (donde los programadores van rotando y modificando diferentes partes del código), teniendo en cuenta la propiedad compartida del código. Podría resultar inmanejable que cada uno de los desarrolladores aplicase sus propios convenios y prácticas de programación concretas.

Estándares de programación

Los estándares de programación, o convenciones de código, son conjuntos de reglas y buenas prácticas que permiten normalizar el código, facilitando su comprensión y mantenimiento.

Entre las normas más habituales podemos encontrar convenios para establecer los nombres de variables, métodos y archivos, el espaciado entre identificadores y símbolos, o la manera de escribir los comentarios y la documentación del código.

Existen normas de carácter general y otras más específicas de lenguajes o plataformas concretas. A continuación, se muestran algunos ejemplos de ellas:

* GNOME programming guidelines. Son un conjunto de normas y guías aplicables a los participantes en el desarrollo del proyecto GNOME y su ecosistema de aplicaciones. Se proporcionan indicaciones que abarcan desde reglas de estilo de programación en C, hasta la manera de generar la documentación del código (GNOME Project, 2020).
* Mozilla coding style guide. Ofrece un conjunto de estilos básicos y patrones que deben seguir los contribuyentes de este proyecto, haciendo énfasis en las buenas prácticas de desarrollo C/C++, IDL o en la gestión de errores (Mozilla, 2007).
* Java code conventions. La comunidad de programadores de lenguaje Java ha tenido siempre clara la importancia de las convenciones de código, puesto que en general el mantenimiento del software no suele ser realizado por el autor original, se aumenta la legibilidad y comprensión del código y, en el caso de distribuir el código fuente como producto, es importante mantener unos convenios de estructura (King et al., 1997).
* PEP 7 y PEP 8. Las PEP (Python Enhancement Proposals) son guías oficiales de la PSF (Python Software Foundation) que proporcionan a su comunidad de desarrolladores información sobre nuevas características del lenguaje, procesos o entornos de desarrollo. La PEP 7 es una guía de estilo para desarrollo en C (van Rossum y Warsaw, 2001), y la PEP 8 es una guía de estilo para desarrollo en Python (van Rossum, Warsaw, y Coghlan, 2013).

A continuación, en la lección magistral *Prácticas de XP* se realiza una revisión las doce prácticas principales que se proponen en la técnica de desarrollo XP para desarrollar un *software* de la mejor calidad posible.

Roles en XP

La programación extrema propone una serie de roles o perfiles específicos que deben participar en el proyecto de desarrollo.

La idea es que los equipos funcionan mejor cuando se definen perfiles concretos que aceptan la responsabilidad de tareas determinadas, consiguiendo una especialización que redunda en el beneficio de todos.

Programador *(programmer)*

Es el centro de XP. En el contexto de los valores y prácticas que aquí se proponen, el programador adquiere un rol más relevante y específico que en otras técnicas.

Además de las tareas habituales de programación, refactorización, definición de pruebas, etc., se espera que el programador posea (Beck, 2000):

* Habilidades comunicativas: facilitando la comunicación con el cliente o la programación en parejas.
* Hábito de la simplicidad: comprometiéndose a producir lo estrictamente necesario para complacer al cliente y manteniendo un diseño sencillo.
* Generosidad y respeto: admitiendo la propiedad compartida del código y el hecho de que otros miembros del equipo puedan modificar el código escrito por uno.
* Coraje: para afrontar los propios miedos a parecer poco inteligente o no lo suficientemente bueno, problemas a los que se enfrenta cualquier desarrollador en algún momento de su vida.

El cliente *(customer)*

Una característica fundamental de la metodología XP es que el cliente es considerado un miembro más del equipo. El programador sabe «cómo programar», mientras que el cliente o usuario sabe «qué se debe programar» (Beck, 2000) y se encarga de:

* Tomar las decisiones más estrechamente vinculadas con la visión estratégica y de negocio del producto.
* Escribir buenas historias de usuario.
* Definir las pruebas de aceptación que se ejecutan al final de cada iteración.

Así, el rol del cliente en un proyecto XP es complicado, pues está encargado de tomar decisiones, y se le exige tanto una visión estratégica y funcional del producto como conocimientos técnicos.

Encargado de pruebas *(tester)*

Este rol:

* Está encargado de ayudar al cliente con las pruebas de aceptación, sobre todo en aquellos casos en los que este último carece de los conocimientos técnicos.
* Define las pruebas unitarias que guían el proceso de desarrollo en cada iteración.
* No necesariamente es una figura independiente de la del programador, pero sí es necesario que alguien acepte la responsabilidad de controlar que todos los test están definidos y se ejecutan correctamente en cada integración.

Rastreador *(tracker)*

El *metric man* observa sin molestar y almacena datos históricos del proceso.

Su labor está estrechamente relacionada con la estimación de esfuerzo de las tareas, que está eminentemente basada en la experiencia. El rastreador analiza las estimaciones realizadas por los miembros del equipo y detecta posibles desviaciones sistemáticas. También tiene una visión general del proceso de desarrollo, anotando información sobre los resultados históricos de las pruebas, el número de defectos detectados o el progreso general de cada iteración, alertando de posibles retrasos.

Entrenador *(coach)*

Es responsable del proceso en su conjunto y tiende a mantenerse en un plano más secundario a medida que el equipo madura y sus funciones van perdiendo relevancia.

Especialmente al comienzo de un proyecto XP, con un equipo poco experimentado, debe asegurarse de que todos los participantes conocen su papel y saben trabajar con las prácticas adoptadas. También debe ser capaz de introducir nuevas prácticas que considere pertinentes, detectar el estado de ánimo del equipo y sugerir acciones de mejora en cuanto a diseño, implementación y pruebas.

Jefe de proyecto *(big boss*o *manager)*

Es el garante de que el proyecto finaliza cumpliendo los objetivos.

Su principal valor es el coraje, pues se espera de él que en momentos críticos tome decisiones importantes. Está encargado de organizar y moderar las reuniones, y de asegurar las condiciones óptimas para el desarrollo del proyecto. Su función está relacionada con el aseguramiento de que el desarrollo del proyecto se ajusta a lo planificado, pero no tanto con organizar el trabajo (definiendo qué tiene que hacer cada participante), pues esto se decide directamente a través del diálogo entre cliente y programador.

Consultor

Es un especialista externo al equipo que ayuda a resolver un problema puntual o cualquier imprevisto relacionado con su especialidad.

XP Industrial

XP Industrial —Industrial XP (IXP)— surgió como respuesta a la necesidad de escalar XP en organizaciones con más de 500 empleados (Kerievsky, 2005).

Intenta mantener el espíritu minimalista con pequeñas iteraciones y gran importancia de las prácticas de ingeniería, como propone XP tradicional, pero haciendo un esfuerzo en la mayor inclusión de la gerencia y la participación del cliente. Introduce seis nuevas prácticas y actualiza algunas de las tradicionales.

En IXP se introducen seis nuevas prácticas respecto de las propuestas en XP tradicional:

* Evaluación de factibilidad. Se trata de una actividad que realiza un experto en IXP para asegurar que la organización está preparada para aplicar esta metodología. Durante uno o dos días el experto se entrevista con posibles participantes y analiza si están preparados para acometer el desarrollo, tanto desde el punto de vista técnico como desde el punto de vista de la interiorización de los valores de XP.
* Comunidad del proyecto. Se extiende la idea de equipo a la idea de comunidad, que incluye, además del equipo técnico y al menos un miembro del lado del cliente, otros agentes (del equipo jurídico, de ventas y marketing, etc.). En una comunidad encontramos participantes más o menos involucrados, en función de sus responsabilidades y nivel de conocimientos. En este caso, al aumentar el número de participantes, es necesario definir previamente los papeles de cada uno y los canales de coordinación y comunicación.
* Calificación del proyecto. En esta actividad se trata de analizar el proyecto minuciosamente y contrastarlo con los objetivos generales de la organización para valorar su adecuación, necesidad y pertinencia. Entre otros aspectos, se analiza la comunidad del proyecto, la visión (futuro que deseamos alcanzar con el desarrollo) y la misión (la estrategia para alcanzar la visión).
* Gestión orientada a pruebas. Se trata de establecer métricas y pruebas que permitan validar en qué medida el proyecto es un éxito o un fracaso, más allá del aspecto puramente técnico, incluyendo objetivos de la organización más amplios.

Las pruebas de gestión son afirmaciones que indican un objetivo medible, acotado en el tiempo y con un resultado binario (se alcanza o no se alcanza el objetivo). Las buenas pruebas de gestión son SMART:específicas *(specific)*, medibles *(measurable)*, alcanzables *(achievable)*, relevantes *(relevant)* y temporalizables *(time-based)*.

Ejemplo de prueba de gestión

El enunciado de una prueba de gestión para una compañía como Apple y su servicio iTunes podría ser el siguiente (Industrial Logic, 2004):

«Nuestro nuevo servicio registrará al menos un millón de descargas de canciones durante el primer mes en producción».

Si durante la primera semana se consigue alcanzar este objetivo, entonces se considera que el test ha sido superado.

* Retrospectivas. Están orientadas a la mejora del propio proceso IXP y consisten en el análisis retrospectivo, después de cada entrega de un incremento, que permite aprender a partir del transcurso de la iteración.
* Aprendizaje continuo. Se incentiva que los participantes del proyecto aprendan nuevas herramientas y técnicas que puedan repercutir en la mejora del proceso global. Es una actividad de mejora más centrada en el nivel individual y el aprendizaje de conocimientos tanto técnicos como no técnicos. Una práctica tradicional que facilita este aprendizaje dentro del grupo es la programación en parejas. En el caso de IXP, las comunidades suelen tener sesiones de estudio en grupo cada una o dos semanas.

Además, IXP introduce algunas modificaciones sobre las prácticas tradicionales de XP, que en su mayoría están orientadas a incluir de manera más clara a clientes y gestores en la práctica diaria:

* SDD *(Storytest–Driven Development)* o desarrollo dirigido por historias. Es una mejora sobre la práctica TDD que trata de no comenzar a escribir código hasta que haya definida una prueba para cada una de las historias de usuario. Son las pruebas de aceptación del sistema.
* DDD *(Domain–Driven Design)* o diseño dirigido a partir del dominio. Es una mejora sobre la metáfora del sistema que busca formalizar el conocimiento que un experto tiene sobre el dominio de aplicación.
* Emparejamientos. Es una mejora sobre la programación en parejas que amplía esta manera de trabajar más allá del ámbito del desarrollo, hacia el coaching, la gestión o la definición de historias de usuario *(storytelling).*
* Usabilidad iterativa. Busca ser una mejora sobre la involucración del cliente, de manera que se evita plantear un diseño complejo desde el comienzo, evolucionando el mismo desde el punto de vista de la usabilidad a medida que se obtiene retroalimentación de los usuarios a través de las diferentes iteraciones.

Valoración crítica de XP

Los beneficios que persigue XP (a través de los principios, valores y prácticas concretas que propone) resultan indiscutibles. En la Figura 6 podemos ver un gráfico que resume las sinergias que se establecen entre las 12 prácticas fundamentales, donde unas pueden mostrarse más débiles y aparecen otras que las refuerzan.

Figura 6. Principales relaciones entre las prácticas de XP. Fuente: Fernández, 2012.

Por ejemplo, la importancia de la refactorización continua, buscando nuevas alternativas de mejora, podría resultar extenuante, ocupar demasiado tiempo o aumentar las posibilidades de que el sistema se vuelva inmanejable ante los continuos cambios, a menos que:

* Adoptemos la propiedad del código compartida, que facilita que tengamos acceso a todo el código y podamos valorar las implicaciones de los cambios.
* Se sigan estándares de codificación, de manera que no sea necesario mejorar el estilo antes de la refactorización y que los cambios producidos sean más comprensibles por otros miembros del equipo.
* Se programe en parejas, de manera que los cambios introducidos sean simultáneamente supervisados por otro compañero.
* Se mantenga un diseño simple, que facilite y al mismo tiempo promueva la refactorización.
* Se disponga de un sistema de integración continua y test automatizados, de manera que cualquier posible problema derivado de los cambios sea detectado de manera prematura.
* El programador se encuentre descansado y con la claridad de ideas suficiente para no cometer errores en la refactorización.

Como muchas de las conocidas como «técnicas ágiles», XP no está exenta de críticas. En concreto, se argumenta que mientras que muchas de las prácticas son beneficiosas, otras planteas problemas y el resto simplemente están sobrevaloradas (Rosenberg y Stephens, 2003).

Esta interdependencia entre las prácticas que acabamos de ver constituye su gran fortaleza y su principal debilidad, puesto que parece exigir que el correcto funcionamiento de XP implica la adopción de todas las prácticas, cosa que no siempre sucede en todas las organizaciones (bien sea por falta de recursos, o por simple desconocimiento).

Las principales críticas que afronta XP son las siguientes (Pressman, 2010; Stephens, 2003):

* Variabilidad de requisitos y alcance. El hecho de involucrar al cliente dentro del equipo de desarrollo puede hacer que los cambios de requisitos se soliciten de manera demasiado informal y que resulte difícil mantener el alcance del proyecto acotado. Los defensores de XP responden que este problema puede suceder con cualquier otra metodología, dependiendo de las características del proyecto.
* Expresión informal de los requisitos. Relacionado con el problema anterior (y en cierto modo en su mismo origen) está el hecho de que la única expresión de las necesidades del cliente son las historias de usuario y las pruebas de aceptación. Los críticos echan en falta una especificación formal, mientras que los defensores argumentan que la naturaleza cambiante de los requisitos en ciertos proyectos vuelve inoperable esta aproximación.
* Conflicto en las necesidades del cliente. Existen proyectos con varios clientes, con necesidades y objetivos que pueden estar en conflicto. XP no plantea una figura clara que sea capaz de moderar estas posibles discrepancias, puesto que el diálogo se establece directamente entre clientes y equipo de desarrollo, y sucede que esta labor de mediación está más allá de la autoridad y competencias de los desarrolladores.
* Carencia de un diseño formal de partida. Los críticos defienden que el desarrollo de sistemas complejos pasa por un análisis detallado previo que genere un diseño robusto fácil de mantener. Los defensores indican que el proceso incremental de desarrollo, con pequeñas entregas, está orientado a mantener la sencillez de un diseño que se va refinando y analizando en las sucesivas iteraciones, a través de las actividades de diseño parcial y refactorización.

Por otro lado, son muchos quienes consideran que XP es una técnica que ha caído en desuso. En el decimoquinto informe sobre el estado de las técnicas ágiles se reflejaba que solo un 1 % de las organizaciones consultadas empleaban XP como técnica de desarrollo, aunque un 6 % mencionaba utilizar una combinación de *scrum* con XP (Digital.ai, 2021).

Entre las razones que podemos encontrar para este declive están las siguientes, todas ellas relacionadas con su dificultad de adopción (Partogi, 2016):

* Está demasiado centrada en tareas de ingeniería. Mientras que el concepto tradicional de metodología de desarrollo se enfoca hacia las tareas de gestión, en XP se da una importancia fundamental a las prácticas de ingeniería del *software.* Muchos jefes de proyecto dejaron abandonadas las tareas de programación en los inicios de sus carreras (en algún caso nunca las realizaron), de manera que XP y su nivel de exigencia desde el punto de vista técnico les asusta. Este tipo de manager tradicional tiene difícil encaje en cualquiera de los roles que propone XP. Este tipo de perfiles se encuentra más cómodo con técnicas como *scrum*, en la forma de *scrum master* (aunque este rol no es en absoluto equivalente al de jefe de proyecto).
* Requiere de un nivel de inversión elevado. Esto se debe fundamentalmente al nivel de capacidad técnica esperado del equipo de desarrollo que eleva los costes de personal. Los programadores deben estar familiarizados con las prácticas de XP, ser capaces de analizar y comprender diferentes tipos de códigos, diseñar el sistema, definir y ejecutar pruebas… Esto no es habitual encontrarlo en programadores experimentados y, mucho menos, entre los recién graduados. En caso de no encontrar el perfil, aumentan los costes de formación. Los costes de infraestructura también son importantes, dada la necesidad de realizar pruebas automatizadas e integración continua. Todo ello sin mencionar lo difícil y costoso que es encontrar un buen coach de XP, por el perfil de «artesano de software» que se le demanda (Manifesto for Software Craftsmanship, 2009).
* Resulta irracional desde el punto de vista de negocio. Las personas que toman decisiones en las organizaciones están más próximas al mundo del negocio que al mundo de la ingeniería. Para ellos, prácticas como las pruebas unitarias (que implican trabajo adicional y más líneas de código) o la programación en parejas (que puede parecer un despilfarro de recursos) resultan simplemente irracionales.
* Es demasiado complejo. En comparación con otras técnicas, como Scrum, XP asigna demasiadas obligaciones al programador. Además, hace demasiado énfasis en las entregas continuas de nuevas versiones, cada semana a ser posible, lo cual puede obligar a limitar a una única historia el incremento generado.

Lo cierto es que, dada la complejidad y nivel técnico de exigencia que impone XP, muchas organizaciones prefieren comenzar con técnicas ágiles más sencillas y flexibles, como *scrum,* e ir introduciendo, progresivamente, aquellas prácticas de ingeniería de XP que resulten más interesantes en función del proyecto.

Scrum y XP

Como hemos visto, *scrum* no prescribe la utilización de prácticas de ingeniería concretas y se reduce a proporcionar un marco de trabajo general para gestionar el proceso de desarrollo de *software.* Sin embargo, es habitual que muchos equipos adopten progresivamente prácticas concretas, como las doce prácticas propuestas por XP (Beck, 2000).

*Scrum* se enfoca en las prácticas de organización y gestión, mientras que XP se centra más en las prácticas de programación.

Esa es la razón de que funcionen bien juntas: tratan de áreas diferentes y se complementan entre ellas (Kniberg, 2007).

Algunas de las prácticas de XP son tratadas directamente por *scrum* y se pueden ver como una intersección entre ambas como prácticas que mejoran la calidad, consiguiendo al mismo tiempo una productividad alta. Entre ellas podemos citar la programación en parejas, el empleo de estándares de programación, el desarrollo dirigido por pruebas, la refactorización o la propiedad del código compartido.

Figura 7. Scrum y XP. Fuente: Partogi, 2018.

9.3. Lean software development

De acuerdo con el Lean Enterprise Institute (LEI),*lean* es un término que se utiliza para tomar conciencia de que todo negocio ha de maximizar el valor que se le da al cliente a la vez que elimina lo que en lean se denomina despilfarro o desperdicio (del japonés *muda* y del inglés *waste*), es decir, se intenta reducir los recursos necesarios para dar ese valor al cliente (LEI, 2020).

Por tanto, para lean, el objetivo final de toda organización ha de ser el de seguir un proceso de creación de valor perfectamente definido que no presente ningún tipo de despilfarro y que permitirá obtener la excelencia en el valor que se proporciona a cada cliente. La identificación del flujo de valor para el cliente es uno de los desafíos de lean.

Womack y Jones (2012), definen el flujo de valor como el conjunto de acciones necesarias que harán pasar el producto de la organización por tres tareas que toda organización considera críticas:

* Solución de problemas. Esta tarea se inicia en la fase de concepción del producto, sigue durante el diseño detallado e ingeniería del producto y finaliza en su producción.
* Gestión de la información. Esta tarea comienza con la recepción del pedido hasta su entrega.
* Transformación física. Esta tarea abarca desde la materia prima del producto hasta su acabado, cuando se entrega al cliente.

La eliminación del despilfarro (*muda*) es un objetivo fundamental del pensamiento lean. El término muda engloba a su vez siete tipos diferentes de despilfarro (Poppendieck y Poppendieck, 2003):

* Sobreproducción. Este despilfarro hace referencia a la producción temprana o excesiva con respecto a lo que va a demandar el cliente. La sobreproducción, además, puede afectar a otros tipos de despilfarro, como, por ejemplo, un exceso de inventario.
* Inventario. Este despilfarro, consecuencia directa del anterior, conduce a un exceso de materia prima, *Work In Progress* (WIP) o productos terminados que no cuentan con un valor añadido para el cliente. Este tipo de despilfarro suele causar obsolescencia, daños sobre los productos, transporte y almacenamiento, debido principalmente a previsiones imprecisas, malos suministradores, excesivos tiempos muertos y crear productos sin que los demande el cliente.
* Procesamiento extra. Este tipo de despilfarro se debe al exceso de trabajo sobre un producto que hace que supere los estándares del cliente y que hace, por tanto, que ese extra de procesamiento sobre el producto sea innecesario ya que al cliente no le supone un valor añadido.
* Movimiento. Este despilfarro se debe a movimientos excesivos por parte de los empleados para poder llevar a cabo sus tareas como, por ejemplo, las herramientas que deben utilizar se encuentran demasiado lejos, el trabajo se ralentiza por falta de formación, o por no saber a dónde ir a preguntar por una mala organización de la empresa. Este tipo de despilfarro hará que se interrumpa el flujo de producción y, en consecuencia, el tiempo de producción se verá incrementando.
* Transporte. Este tipo de despilfarro se debe a movimientos innecesarios entre partes de distintos procesos. También incrementa el tiempo de producción (así como el *work in progress*) lo que puede afectar incluso a la calidad final del producto.
* Esperas. Este tipo de despilfarro se debe a la espera por parte de los empleados para disponer de la maquinaria o de las piezas necesarias para realizar su trabajo.
* Defectos. Este tipo de despilfarro se debe a la fabricación de productos con defectos que no van a pasar el control de calidad. Hace que aumenten los costes, ya que será necesario rectificar el producto para que obtenga la calidad esperada.

Aparte del despilfarro, se identifican también en *lean* otros dos tipos de desperdicios que se deberían eliminar y que están asociados a los procesos de las organizaciones:

* Mura (irregularidad). Este término hace referencia a la poca homogeneidad y a la inconsistencia que se da en los resultados obtenidos, en lo que respecta a calidad y volumen de los productos fabricados.
* Muri (trabajo *tensionante*). Este término hace referencia a trabajos que se imponen debido a una mala organización en la empresa, tales como: la realización de tareas peligrosas, hacer trabajar más rápido de lo habitual, llevar cargas físicas muy pesadas, etc. Muri también hace referencia a las malas condiciones del puesto de trabajo, lo que afectará al rendimiento y buen funcionamiento de la organización.

Conviene aclarar que *lean* no es ninguna metodología, tal y como se entiende en el sentido convencional, por lo que no va a decir qué es lo que se debe hacer exactamente para conseguir la excelencia en el valor que se le da al cliente.

El término lean se refiere más bien a la síntesis de un conjunto de principios, un pensamiento o una filosofía a seguir a la hora de construir productos de cualquier tipo. Algunos de los principios más importantes que recoge *lean* son los que se indican a continuación (Burlton y Matthijssen, 2013; Garzás, Enríquez de S. y Irrazábal, 2013):

* Valor que se da al cliente y calidad. Definir qué es lo que representa valor para el cliente y centrarse en las actividades que proporcionan dicho valor. En este proceso se procurará eliminar todos los defectos, así como detectar y solucionar todos los problemas lo más pronto posible.
* Eliminación del despilfarro. Las organizaciones y los procesos que tienen implantados están llenos de despilfarro, por lo que hay que eliminar todo aquello que resulta prescindible y que no aporta valor al cliente (sobreproducción, tiempos de espera, inventarios, transportes innecesarios, etc.).
* Mejora continua. Todos los procesos han de seguir un flujo continuo, es decir, han de estar en movimiento y ser dinámicos para que se pueda añadir valor de una manera continua. Este principio de *lean* conocido como «kaizen» parte de la idea de que todo es susceptible de ser mejorado. En *lean* se asume que la perfección no existe, pero hay un esfuerzo de superación continua para intentar alcanzarla. Por ello, todos los esfuerzos del personal involucrado en los procesos se ven dinamizados, para que los procesos puedan mejorar con el mínimo coste posible y sin poner en riesgo la estrategia de la organización ni su competitividad en el mercado.
* Procesos «pull». Producir solamente lo necesario, de acuerdo a lo que el cliente necesita y evitando bloquear recursos (personas, infraestructura, etc.) cuando su demanda no es inmediata. El concepto «pull» se basa en el hecho de que el volumen de producción ha de ser acorde a la demanda del cliente (es decir, necesidades reales donde el cliente es el motor de la producción) y no construir producto a la espera de que el cliente lo compre (procesos «push»).
* Flexibilidad. Capacidad para producir de una manera rápida diferentes tipos de productos, manteniendo la eficiencia a pesar de que el volumen de producción se vea reducido.
* Relación a largo plazo con los proveedores. Firmar acuerdos con los proveedores en los que se permita compartir información y asumir los riesgos de los costes.

Aunque el concepto de *lean* se suele considerar como un sinónimo del método de fabricación de la empresa automovilística Toyota, conocido como Toyota Production System (TPS), realmente este término no fue acuñado por los japoneses, sino por tres autores estadounidenses, James P. Womack, Daniel T. Jones y Daniel Roos, que en los años noventa utilizaron el término *lean* para describir, en su libro *The Machine That Changed the World* (Womack, Jones y Roos, 2007), el proceso de negocio que seguía esta empresa japonesa desde los años cincuenta.

En este sentido, resulta interesante comentar también que, para su aplicación en Japón, se fijaron en el modelo de producción en masa de vehículos, establecido por Henry Ford, con el que consiguió que sus coches estuvieran al alcance de más gente bajando el tiempo y los costes de producción, en base a una cadena de montaje más eficiente, ordenada y con piezas estandarizadas fabricadas en serie.

Taiichi Ohno fue el creador del sistema de producción de Toyota (TPS) y autor del libro *Toyota Production System: Beyond Large-Scale Production* (Ohno, 1988). La estrategia definida por Ohno para construir un sistema de producción libre de despilfarro y reducir, por tanto, los gastos de producción, estaba basada en tres principios fundamentales:

* Construir estrictamente lo necesario.
* Eliminar todo aquello que no genere valor para el cliente.
* Si algo no va bien, parar la producción y solucionar los problemas.

Sin embargo, aunque todo en *lean* parece ideal (la solución perfecta y más o menos sencilla de implementar) conviene resaltar que son muchas las empresas en todo el mundo que fracasan en su intento de implantar *lean* por seguir un camino equivocado. La resistencia al cambio; el giro de 180° que muchas organizaciones han de realizar en su modo de trabajar, con todo el volumen de trabajo adicional que esto conlleva; hasta que empieza a funcionar y sumado al el esfuerzo para la mejora continua que nunca acaba, pueden hacer que la cultura *lean* acabe fracasando estrepitosamente.

Burlton y Matthijssen (2013), establecen cuatro factores de éxito que se han de considerar a la hora de implantar esta cultura en una organización:

* Compromiso por parte de la dirección. La dirección debe asumir el camino que va a iniciar cuando se decide a adoptar *lean.* En este aspecto, la dirección ha de estar dispuesta a realizar cambios profundos en su forma de hacer las cosas, especialmente a nivel de dirección, donde se requiere más involucración, colaboración y deseo de compartir, sin perder en ningún momento la estrategia de negocio de su organización.
* Focalización en el valor que se le da al cliente y la calidad. La organización ha de saber qué es exactamente lo que para el cliente representa valor y qué no lo es, la eficiencia y la calidad en el trabajo debe primar sobre los costes requeridos. En este sentido, hay que tener en cuenta que siempre que se haga lo que desea el cliente, es decir, siempre que los productos «se hagan bien», que no viene a decir lo mismo que «estar bien hechos», la organización se ahorrará dinero con total seguridad.
* Implicación de todo el personal que hace el trabajo real. Cuando la dirección realice cambios, en la forma de trabajar, en la unidad organizativa de la empresa, etc., deberá tener siempre en cuenta el involucrar y mantener informadas a todas las personas que realmente están realizando el trabajo de fabricación de los productos, ya que son quienes tienen el mayor conocimiento sobre qué es lo que puede aportar valor al cliente y dónde está el despilfarro. Los cambios por realizar para adoptar una cultura *lean* no se deben imponer a las personas sin ningún tipo de justificación ni de explicación. Si todos los afectados por la adopción de *lean* entienden el objetivo a alcanzar, hay respeto, reconocimiento y van todos juntos en el mismo camino, la probabilidad de conseguir el éxito se ve gratamente incrementada.
* Organización por procesos. De cara a poder implantar de manera exitosa el pensamiento *lean* en la forma de trabajar, la definición correcta de los procesos es un requisito imprescindible. Sin embargo, en la mayoría de las empresas, especialmente en aquellas dedicadas a proporcionar servicios, los *workflows* de los procesos de negocio brillan por su ausencia. De acuerdo a Marshall (2000), un *workflow* sería de manera simplificada un modelo de un proceso de negocio concreto que define el qué (objetivo del proceso), cómo (actividades), quién (responsables) y cuándo (orden) de un proceso de negocio real.

Las técnicas que permiten eliminar el despilfarro, así como las que mejoran el flujo de valor de la organización, son importantes para aumentar el valor que se le proporciona al cliente, pero no las únicas, ya que las técnicas orientadas a visualizar los procesos, garantizar la calidad, medir el rendimiento y mantener una comunicación continua, de cara a mejoras, también son muy importantes.

Una técnica muy vinculada al pensamiento *lean,* y que se utiliza en las organizaciones para la resolución de problemas, es lo que se conoce como ciclo de mejora continua de deming o plan-do-check-act (PDCA). El ciclo de *deming* o PDCA fue introducido en Toyota por el autor que le dio nombre, el Dr. W. Edwards Deming, un estadístico norteamericano que definió los siguientes pasos como método de mejora en la fabricación de productos:

* Plan (planificar). En esta etapa se ha de responder a las preguntas: ¿qué se debe hacer?, ¿cuándo?, ¿quién debe hacerlo?, ¿cómo?, ¿utilizando qué?
* Do (hacer). En esta etapa se llevan a cabo las tareas programadas y se documentan las acciones realizadas.
* Check (verificar). En esta etapa se determina si las tareas llevadas a cabo han dado los resultados esperados y se documentan las conclusiones.
* Act (actuar). En esta etapa se ajustan los planes basándose en toda la información recogida y se documenta el proceso.

Aparte del enfoque de mejora continua, en el proceso de creación de los productos existe también otro enfoque de especial interés conocido como *total productive mainteinance* (TPM), que se encarga de la gestión de la calidad durante el proceso de mantenimiento de los productos que fabrica una organización. Uno de los pilares de TPM es lo que se conoce como los principios 5S. Los principios 5S definen una serie de disciplinas que se han de seguir cuando se mantiene un producto o de cara a su mantenimiento (Martin, 2009):

* *Seiri* (organización en el sentido de ordenación). Este principio establece que una organización ha de saber en todo momento dónde se encuentra cada cosa, es decir, que se coloque en el lugar que le corresponde.
* *Seiton* (orden en el sentido de sistematización). Muy vinculado al anterior, este principio establece que hay que definir un lugar para cada cosa y cada cosa ha de estar en su lugar.
* *Seiso* (limpieza). Este principio establece que el puesto de trabajo ha de mantenerse limpio y libre de cables, basura, chatarra, etc.
* *Seiketsu* (estandarización). Relacionado con el anterior, este principio establece que la organización determina qué es lo que se va a hacer para mantener el puesto de trabajo limpio.
* *Shutsuke* (disciplina). Este principio establece que hay que tener la disciplina de seguir todas las prácticas, estándares, etc. que se hayan definido en la organización, para realizar el trabajo y tener la voluntad de cambiar en la forma de trabajar si la aplicación de las prácticas, guías, etc. refleja que no se estaba haciendo bien y están especificando una nueva forma de llevarlo a cabo.

De lo explicado hasta ahora, resulta inevitable relacionar todo lo que establece *lean* a la ingeniería de *software* (eliminar despilfarro, observar el lugar de trabajo para detectar errores y proponer mejoras, principios 5S, etc.). Por ejemplo, los principios 5S aplicados al desarrollo de *software* podrían tener la siguiente interpretación (Martin, 2009):

* *Seiri* (organización en el sentido de ordenación). Aplicado al desarrollo de *software* se podría seguir, por ejemplo, con una alta cohesión de cada componente y un nombrado coherente de los elementos que forman parte del código, para facilitar su posterior localización en caso de cambios en las funcionalidades del sistema.
* *Seiton* (orden en el sentido de sistematización). Aplicado al desarrollo del *software,* en este caso, se referiría a que cada porción de código debe localizarse en el lugar que se le asignó, donde el programador espera encontrarlo, y en caso de no ser así entonces hay que refactorizar el código para su reubicación.
* *Seiso* (limpieza). Aplicado al desarrollo de *software* se focalizaría, por ejemplo, en eliminar todos los comentarios del código que sean redundantes y que no sirvan para entender la historia que se cuenta a través del código que nos explica cómo está implementado el sistema.
* *Seiketsu* (estandarización). Aplicado al desarrollo del *software* se trataría de que todo el equipo de desarrollo lea y siga los libros de referencia que se hayan establecido como estándares para el desarrollo del *software* de la organización (guías de estilo, buenas prácticas en construcción de *software*, etc.).
* *Shutsuke* (disciplina). Aplicado al desarrollo de *software* se trataría, por ejemplo, de cuando un programador ha de seguir una norma de estilo para el nombrado de los métodos de las clases del sistema y se da cuenta que es distinto a como lo venía haciendo hasta ahora. El programador no ha de ignorar la norma y deberá cambiar todo su código de acuerdo a lo que se describe en el documento correspondiente. Aquí la premisa sería que hacer el código legible es tan importante como hacerlo ejecutable.

Mary Poppendieck y Tom Poppendieck son los autores que, a través de su libro *Lean Software Development,* realmente popularizaron el término «lean» aplicado al desarrollo de *software* (Poppendieck y Poppendieck, 2003). El desarrollo ágil, por ejemplo, tiene también como objetivo eliminar todo el despilfarro posible que suele acompañar el proceso de construcción de *software.* La Tabla 3 muestra la similitud del despilfarro en el desarrollo de *software* con los siete tipos de despilfarro definidos por *lean*que se han mencionado anteriormente y que están vinculados a la fabricación de productos físicos.

Tabla 3. Equivalencias de tipos de despilfarro entre fabricación y desarrollo de software. Fuente: Poppendieck, M. y Poppendieck, T., 2003.

Siguiendo la Tabla 3, el despilfarro en el desarrollo de *software* viene definido de la siguiente manera (Poppendieck y Poppendieck, 2003, p. 4-8):

* Trabajo incompleto. Este despilfarro traerá como consecuencia que el *software* que está sin terminar se quede obsoleto y haya que replantearse un nuevo desarrollo para el sistema en su totalidad, rehaciendo lo que ya estaba implementado.
* Procesos extras. En el desarrollo de *software* se suele generar mucha documentación en papel que hay que entregar al cliente y que realmente no le aporta ningún valor (tampoco resulta de valor para el desarrollo del sistema en sí), pero que simplemente se entrega al cliente porque así está definido, como «entregables», en los pliegos del contrato. Por tanto, hay que valorar qué documentación exactamente tiene valor para el cliente de cara al proyecto.
* Características extras. Muchas veces parece una buena idea incluir características extras al sistema pensando que son sencillas de incluir y que al cliente le van a gustar, o bien, porque el programador, en algunas ocasiones, quiere probar una nueva tecnología en el sistema para ver cómo funciona, pero no realmente porque el sistema solicitado por el cliente así lo requiera. Sin embargo, hay que ser conscientes de que, a la hora de desarrollar *software,* hay que realizar trazas sobre cada elemento incorporado, compilar, integrar, probar cada vez que se modifique una línea de código, diagrama, modelo, etc., y por supuesto, cualquier característica extra que se haya incluido tendrá que ser mantenida durante toda la vida del sistema.
* Intercambio de tareas. El asignar personas a varios proyectos a la vez es una fuente de despilfarro muy significativa. Está demostrado que el tener a una persona en más de un proyecto hace que se multipliquen las interrupciones y el intercambio de tareas de un proyecto a otro, lo que va a retrasar cada proyecto de manera importante incumpliendo los plazos estimados y establecidos inicialmente.
* Esperas. Durante un proceso de desarrollo de *software,* las esperas por cosas que se necesitan para continuar el trabajo constituyen también una de las grandes fuentes de despilfarro. Retrasos en el inicio del proyecto, retrasos en la contratación del personal para el proyecto, retrasos debido a una documentación excesiva de los requisitos, retrasos en las revisiones y aprobaciones, retrasos en las pruebas y retrasos en los despliegues constituyen un despilfarro importante a la hora de construir *software.*
* Movimiento. En este caso cabe plantearse el movimiento que se ha de llevar a cabo para obtener respuesta cada vez que un programador tiene una duda sobre algún aspecto de la implementación. Hay que considerar si se tiene a mano personal en la organización que desarrolla el *software,* para poder resolver todas las dudas técnicas, o si hay que preguntarle al cliente, o a algún representante del cliente, si tiene que irse a otra planta del edificio para comprobar cómo han ido las pruebas sobre el código que ha implementado.

Es por ello por lo que se recomienda que todo el equipo de desarrollo (programadores, los que realizan las pruebas del *software,* representantes del cliente que podrían pertenecer a la misma organización que desarrolla el *software,*etc.) se encuentren ubicados en una misma sala. Pero no son solamente las personas las que tienen que moverse de cara, en su caso, a resolver dudas y otros aspectos vinculados al desarrollo del *software,* también los requisitos se mueven cuando pasan de los analistas a los diseñadores, de ahí a los programadores y después a los que realizan las pruebas. Cada traspaso o movimiento que se realiza de un artefacto puede generar despilfarro en el proyecto.

* Defectos. Dependiendo del impacto del defecto detectado, y del tiempo que se tarde en detectar el defecto en el desarrollo del *software,* se podrá generar más o menos despilfarro. En este sentido, no tiene nada que ver con la gravedad del defecto, ya que un defecto crítico que se ha detectado rápidamente (por ejemplo, en tres minutos) no va a generar mucho despilfarro. Sin embargo, un defecto mínimo, por simple y sencilla que sea su resolución, si transcurre mucho tiempo hasta su descubrimiento, por ejemplo, varias semanas, puede constituir una fuente importante de despilfarro para el proyecto.

A pesar de las similitudes, en muchos aspectos, entre el pensamiento *lean* asociado a TPS y el *lean* aplicado al desarrollo de *software,* lo que ocurre, como ya se ha demostrado y citado en incontables ocasiones, construir *software* no es igual que construir coches, por lo que también se dan diferencias entre el desarrollo de *software* y la producción en el pensamiento *lean,* tal y como se ilustra en la Tabla 4.

Tabla 4. Diferencias entre producción y desarrollo de software. Fuente: Poppendieck, y Poppendieck, 2003.

Una de las diferencias fundamentales que se muestra en la Tabla 4, y que requiere un poco más de detalle, es la que se refiere a la calidad. La calidad, vista desde la producción de productos físicos, está definida en función de si es conforme a los requisitos que se especificaron en el diseño o la receta.

De esta manera, por ejemplo, para elaborar un plato, se sigue la receta, y la calidad vendrá en función de si se han seguido correctamente los pasos y se han utilizado los ingredientes adecuados (otra cosa sería la calidad del servicio, por ejemplo, a la hora de servir el plato en un restaurante, pero este aspecto ya entraría en otro contexto distinto que no se va a tratar en este tema).

Sin embargo, la calidad en el *software* depende de la integridad percibida y de la integridad conceptual(Poppendieck y Poppendieck, 2003). La integridad percibida depende del cliente y se obtiene como el conjunto de lo que sería funcionalidad, usabilidad, fiabilidad y coste del producto. La integridad conceptual hace referencia a que todos los conceptos vinculados al *software* se encuentren distribuidos de una manera cohesiva, es decir, se mantiene un equilibrio entre flexibilidad, mantenibilidad, eficiencia y un comportamiento reactivo.

Mary Poppendieck y Tom Poppendieck resumen en siete los principios aplicables de *lean* al desarrollo de *software* (Garzás, Enríquez de S. y Irrazábal, 2013) (Poppendieck y Poppendieck, 2003):

* Eliminar despilfarro. Este principio se basa en eliminar todo aquello que no genera valor para el cliente, con la misma filosofía del *lean* de TPS.
* Amplificar el aprendizaje. Este principio se basa en que el desarrollo de *software* es un proceso de aprendizaje continuo en el que, por ejemplo, a través de reuniones frecuentes y cortas del equipo de desarrollo con el cliente, o sin el cliente, se pueda ir aprendiendo sobre la funcionalidad que ha de implementar el sistema y la mejor forma de llevarla a cabo.
* Retrasar las decisiones lo más que se pueda. Este principio se basa en que hay que retrasar las decisiones todo lo que se pueda para controlar mejor la incertidumbre en muchos aspectos vinculados al *software.*
* Realizar entregas lo más rápido posible. Este principio se basa en que hay que hacer entregas del *software* lo más rápido posible para poder ir recibiendo *feedback* del cliente e ir orientando mejor el proyecto.
* Dar más valor al equipo. Este principio se basa en que los roles en el proyecto deben cambiar. Así, los directivos aprenderán a escuchar y valorar al equipo de desarrollo (y en este caso no se está refiriendo al aspecto económico), que les explicarán las acciones que se podrían llevar a cabo en el proyecto de cara a la implementación y sugerencias de cómo mejorarlo.
* Crear la integridad. Este principio se basa en la importancia de disponer de un sistema de integración continua para que el desarrollo sea fácil de mantener, mejorar y reutilizar.
* Visualizar todo el conjunto. Este principio se basa en la idea de enfocarse en el flujo completo de valor hacía el cliente y en la entrega de un producto completo. Hay que tratar de evitar que el foco de atención recaiga sobre un elemento en particular, ya sean los plazos, los costes o los procesos.

9.4. Kanban

*Kanban* tiene su origen en una palabra japonesa cuyo significado es ‘tarjetas visuales’ y fue utilizada por primera vez en la compañía Toyota. Taiichi Ohno se fijó en la forma de trabajo de los supermercados norteamericanos a la hora de reponer los productos en las estanterías. La idea era que una vez que se producía la demanda de un coche se utilizaran las tarjetas para saber todas las piezas que se necesitaban para su construcción, cuántas estaban disponibles y cuántas faltaban.

De esta manera se podía planificar la fabricación del coche, controlar el inventario y coordinar los distintos pasos del proceso de la fabricación del vehículo siguiendo la filosofía *lean* del *just-in-time*, es decir, de cara a aumentar la productividad en la fabricación de los productos, el objetivo es producir exclusivamente los elementos que se necesitan, en su cantidad justa y en el momento que se necesitan, eliminando todo despilfarro.

La primera aplicación del sistema *kanban* en el desarrollo de *software,*vinculado de forma muy estrecha con el pensamiento *lean,* la llevó a cabo David Anderson, como presidente de la empresa Modus Cooperandi (Anderson, 2009).

La técnica kanban hace énfasis en la entrega continua de resultados, teniendo en cuenta que se deben evitar saturaciones y cuellos de botella según las capacidades del equipo de desarrollo.

Se basa en tres principios:

* Visualización del flujo de trabajo: ofrece una herramienta fundamental, el tablero kanban, que permite obtener información visual sobre cada tarea y sus relaciones con un solo golpe de vista.

Se descompone el trabajo en tareas que se desplazan de izquierda a derecha, y en columnas se muestra el estado del trabajo (pendiente, en progreso, finalizado, en la configuración más básica) como se puede ver en la Figura 8.

* Limitación de la capacidad de trabajo en proceso (*work in progress*): busca equilibrar la carga de trabajo en cada momento, de manera que el equipo no quede saturado. En cada columna del tablero no se puede superar un volumen de trabajo prefijado.

Figura 8. El tablero Kanban. Fuente: Vila, 2016

* Flujo continuo: cuando una tarea se finaliza, la que le sigue en orden de prioridad inmediatamente se pone en desarrollo. No se trabaja por lotes de trabajo, como sucede en *scrum* con sus *sprints,* sino que todas las tareas pendientes son susceptibles de ser comenzadas en cualquier momento del desarrollo.

Para el desarrollo de *software,* *kanban* al igual que *scrum* utiliza un desarrollo incremental, historias de usuario, tareas, etc., pero en este caso, además, hace uso de un tablero, el denominado «tablero *kanban*», que va a permitir a cada miembro del equipo de desarrollo obtener una visualización completa del estado de implementación del *software.*

La idea consiste en un tablero que se divide en una serie de columnas que reflejan el estado del proyectoen lo que a tareas se refiere. Básicamente: tareas que hay que hacer, tareas pendientes, tareas que se están realizando y tareas que se han terminado. Los más puristas del *kanban* añaden una columna más que refleja el estado de ánimo de cada miembro del equipo de desarrollo, normalmente con el uso de caritas. De esta manera, las tarjetas *kanban* se irán colocando en cada columna según corresponda.

Las Figuras 9 y 10 muestran ejemplos de tableros *kanban* utilizados en distintos proyectos *software.*

Figura 9. Ejemplo de tablero Kanban. Fuente: Rocha, 2021.

Figura 10. Ejemplo de tablero Kanban. Fuente: Kanbanize, 2022.

9.5. Scrumban

Cuando una organización se decide a adoptar prácticas ágiles la opción más común es utilizar *scrum*acompañado de *kanban*, ya que ambos enfoques se complementan muy bien, aunque por separado presenten aspectos bien diferenciados, tal y como se resume en la Tabla 5.

Tabla 5. Características de *scrum* y *kanban.*Fuente: elaboración propia.

La manera más simple de combinar ambas técnicas consiste en aprovechar las capacidades de visualización del proceso que ofrece *Kanban*, con el tablero *kanban,* como podemos ver en los ejemplos anteriores (Figuras 9 y 10).

En *scrumban* se tiende a adoptar un modelo de proceso con flujo continuo, como propone *kanban*, de manera que la pila del sprint ya no es necesaria, y todas las tareas pendientes están agrupadas en la columna inicial del tablero (Vila, 2016).

También, se puede utilizar esta técnica del tablero y las caritas en la reunión de retrospectiva de la iteración llevada a cabo (en *scrum,* la conocida como reunión *sprint retrospective*), para poder mostrar así, de manera gráfica, el grado de satisfacción en distintos aspectos, tal y como se ilustra en la Figura 11, donde se valora la técnica, el ecosistema y la filosofía empleada en la iteración.

Figura 11. Tablero para la reunión Sprint Retrospective. Fuente: Wallet, 2011.

9.6. Otras técnicas ágiles

*Crystal methods*

Se trata de una familia de técnicas creadas por Alistair Cockburn que se centran en el equipo, entendido como el conjunto de personas que deben desarrollar el *software.*

Se parte de la base de que el desarrollo de aplicaciones es una serie de juegos de cooperación, con recursos limitados, donde la comunicación juega un papel fundamental. En este sentido, es fundamental el tamaño del equipo, puesto que no es lo mismo organizar un grupo pequeño, de tres desarrolladores, que uno más grande, de veinte. Por tanto, en función del tamaño del equipo, aparecen las diferentes variantes de Crystal (Fernández, 2012):

* *Crystal clear,* para equipos de menos de ocho personas.
* *Crystal yellow,* para equipos de diez a veinte personas.
* *Crystal orange,* para equipos de veinticinco a cincuenta personas, etc.

Las distintas técnicas comparten fundamentos comunes, mientras que los roles, patrones del proceso y las prácticas, son particulares en cada variante. Las prioridades comunes a todas las técnicas son las siguientes:

* Seguridad en el resultado del proyecto: busca obtener un resultado razonable en función de las prioridades particulares de cada proyecto y los recursos disponibles.
* Eficiencia en el proceso de desarrollo: aspecto que, una vez más, está estrechamente relacionado con el aprovechamiento de los recursos disponibles.
* Aceptación de las convenciones: hace referencia a la comodidad que sienten los desarrolladores al adoptar las normas impuestas por la técnica. Esta es una característica importante, que surge de la comprobación empírica de que en muchas organizaciones las técnicas impuestas no siempre son aceptadas con gusto (y muchas veces son obviadas) por los propios trabajadores.

Además, se definen siete propiedades fundamentales que cualquier proyecto de desarrollo debería poseer para tener ciertas garantías de éxito (Lowe, 2016). Todas ellas se aplican a proyectos de cualquier tamaño, pero las tres primeras se consideran fundamentales e ineludibles:

* Entregas frecuentes. Para Cockburn (2004), la propiedad más importante de cualquier proyecto es la entrega de productos funcionales a los usuarios finales cada poco tiempo. En proyectos de carácter general se establece un intervalo máximo de cuatro meses (y preferiblemente dos), mientras que en desarrollos web el espacio entre entregas se reduce a una semana.
* Mejora reflexiva. El equipo debe dialogar y reflexionar sobre su propia práctica para proponer acciones de mejora.
* Comunicación osmótica. El concepto de ósmosis hace referencia a la absorción de información en segundo plano, aunque no se produzca una comunicación formal, por la simple escucha de lo que sucede en el espacio de trabajo. Por este motivo los equipos deben estar localizados físicamente en el mismo lugar.
* Seguridad personal. Se debe facilitar que todos los integrantes del equipo tengan libertad para expresar sus opiniones sin temor a represalias.
* Focus. Cada miembro del equipo debe contar con la capacidad, tiempo y herramientas para concentrarse en lo que debe y puede hacer en cada momento, sin interrupciones ni distracciones.
* Acceso directo a usuarios expertos. Se trata de obtener retroalimentación temprana y de calidad por parte de los usuarios finales del sistema.
* Un entorno tecnológicamente apropiado. El equipo debe contar con herramientas que permitan las pruebas automatizadas o la integración frecuente.

*Feature driven development* (FDD)

Esta técnica se basa en iteraciones de duración muy corta, siempre inferior a dos semanas. Todas las tareas de análisis, diseño e implementación se centran en cumplir una serie de características (*features*) del *software,* que deben cumplir las siguientes propiedades (Fernández, 2012):

* Ser sencillas y poder ser desarrolladas en poco tiempo (entre uno y diez días).
* Representar una aportación para el cliente y su negocio.
* Se deben expresar según el patrón <acción><resultado><objeto> (por ejemplo, «calcular el importe total de un pedido»).

FDD hace énfasis en las tareas de diseño y considera un modelo de proceso con cinco pasos (Palmer, 2009; Palmer y Felsing, 2001), representados en la Figura 11:

* Desarrollo de un modelo de dominio general. El equipo, en colaboración con el cliente, desarrolla un modelo general del sistema de manera iterativa bajo la supervisión de un arquitecto jefe. Este proceso permite, a los expertos en el dominio de aplicación y al equipo de desarrolladores, compartir información y establecer un lenguaje común.

El modelo inicial da una representación amplia del problema, sus objetos e interacciones, y va adquiriendo profundidad y refinamiento a través de las sucesivas iteraciones.

* Construcción una jerarquía de características. Se descompone la funcionalidad en características sencillas. Cada una de ellas debe poder ser implementada en menos de dos semanas (y preferiblemente entre uno y tres días).

No es el modelo el que guía el desarrollo, sino estas características. Las características se organizan en una jerarquía de tres niveles (áreas del dominio, actividades de cada área y características concretas).

* Planificación por característica. Se analizan las características en función de su prioridad, dependencias y complejidad, y se define un orden de implementación. Se agrupan en paquetes y cada uno es asignado a un programador jefe, que se responsabiliza y especializa en ese conjunto de características.
* Diseño por característica. Tras la planificación inicial, comienza el proceso iterativo.

En cada iteración, el programador jefe selecciona un conjunto de características de su paquete y se elabora un diseño más detallado, y una descomposición en clases que se distribuyen entre los programadores. Así, la responsabilidad sobre cada clase es individual y la propiedad del código no es compartida (como sucede en XP), buscándose la especialización.

* Desarrollo por característica. Durante el resto de la iteración cada programador implementa y prueba el código, y, tras un proceso de inspección, se produce la integración en el repositorio general.

Figura 12. Modelo de proceso en FDD. Fuente: adaptada a partir de Palmer y Felsing, 2001.

*Adaptivesoftware development* (ASD)

Esta técnica, propuesta por Jim Highsmith (2000), parte del convencimiento de que las necesidades del cliente van a cambiar siempre, tanto durante el proyecto como tras su finalización. Más que de una metodología, estamos hablando de un conjunto de procedimientos que permite a las empresas alcanzar una cultura de adaptación.

Sus objetivos son los siguientes (Fernández, 2012; Highsmith, 2002):

* Concienciar a la organización sobre la necesidad de trabajar con el cambio y no contra el cambio.
* Desarrollar proyectos iterativos de gestión del cambio y facilitar la colaboración entre las personas.
* Marcar una estrategia de desarrollo rápido sin olvidar el rigor y la disciplina necesarios.

ASD plantea un modelo de proceso con las tradicionales fases (planificación, diseño y construcción) sustituidas por un modelo más dinámico y abstracto: especulación, colaboración y aprendizaje (Highsmith, 2002), como vemos en la Figura 12:

Figura 12. Fases y prácticas en el ciclo de vida de ASD. Fuente: adaptada a partir de Tutorials Point, 2022.

* Fase de especulación. El término «especulación» reconoce la naturaleza incierta de los sistemas complejos, y anima a la exploración y la experimentación. Asume la existencia de incertidumbres y, aunque la planificación no se abandona, debemos estar dispuestos a apartarnos en cualquier momento de planes predefinidos. Aquí encontramos dos prácticas:
  + Iniciación del proyecto. Definir la misión del proyecto y sus objetivos, entendiendo las restricciones, estableciendo la organización del proyecto, identificando y esbozando requisitos, estimando el tamaño y el alcance, e identificando los riesgos principales.
  + Planificación adaptativa. Con la información de alcance y los recursos disponibles se define un marco temporal para el proyecto. Dentro de esta duración total, se establece el número de iteraciones y la duración de cada una. Cada una de ellas queda asociada a un tema general u objetivo de manera que pueda suministrar un conjunto de características relacionadas, demostrables al cliente para su revisión al finalizar.
* Fase de colaboración. En ASD se parte de que los sistemas complejos no se construyen, evolucionan. Además, estas aplicaciones requieren de la adquisición, análisis y utilización de grandes cantidades de información, no manejables por una única persona, y cuya comprensión requiere a menudo de capacidades y conocimientos específicos. En este contexto las habilidades comunicativas del equipo son fundamentales.
  + Desarrollo concurrente del sistema, aplicando técnicas de ingeniería orientada a componentes. El equipo técnico, de carácter multidisciplinar, se concentra en la producción del *software,* mientras que los jefes de proyecto se encargan de facilitar la colaboración y el desarrollo de actividades concurrentes.
* Fase de aprendizaje. En proyectos complejos, más que en ningún otro caso, se pueden cometer equivocaciones. Es necesario aprender del proceso y de nosotros mismos como equipo, empleando técnicas como las retrospectivas y *focus groups* con clientes. Aquí encontramos dos prácticas fundamentales:
  + La revisión de calidad, tanto desde la perspectiva del cliente (empleando *focus groups*) como desde un punto de vista técnico (empleando la programación en parejas, las revisiones técnicas o la revisión de la arquitectura al final de cada iteración).
  + Preguntas, respuestas y *release*. Dentro de la fase de aprendizaje es vital dedicar un tiempo a que el equipo reflexione y analice sus propias prácticas para identificar fallos y puntos de mejora. También se debe analizar el estado global del proyecto, evaluando el grado de ajuste a la planificación inicial y adoptando las acciones correctivas necesarias.

*Dynamic systems development method* (DSDM)

Sus orígenes se remontan a 1994, cuando se desarrolló para crear una herramienta RAD (*rapid applications development*) unificada, mediante la implementación de un *framework* para el proceso de desarrollo de *software.* DSDM se basa en tres principios (Fernández, 2012):

* Nada es construido a la perfección a la primera.
* Regla 80-20: el 80 % de funcionalidades se implementan con el 20 % del tiempo y lo restante consume el 80 % del tiempo.
* Resulta improbable que se conozcan los requisitos del sistema desde el primer día.

DSDM se basa en cinco fases de desarrollo, aunque únicamente las tres últimas fases son iterativas:

* Estudio de viabilidad.
* Estudio del negocio.
* Modelado funcional.
* Diseño y construcción.
* Implementación.

*Scaled agile framework* (SAFe)

*Framework*, creado por Dean Leffingwell, para organizar grandes empresas. Su principal objetivo consiste en ayudar a implementar la agilidad en la empresa, a nivel de la organización, y no sólo a nivel de equipo, algo a lo que, por ejemplo, está más enfocado Scrum (Garzás, 2013; SAFe, 2022).

SAFe está basado en principios ágiles y lean, y para su implantación en una empresa se distinguen tres niveles de abstracción que hay que coordinar entre sí:

* Nivel de equipo. Se define cómo organizar a cada equipo que interviene en el desarrollo software. SAFe propone que los equipos utilicen una combinación de técnicas de *scrum* y XP.
* Nivel de programa.
* Nivel de portfolio. Se define lo que más valor aporta a la organización y se emplean los principios de lean, además de utilizar tableros *kanban* para organizar las tareas de este nivel.

Cada uno de estos niveles posee un *backlog* (*team backlog, program backlog* y *portafolio backlog*), en el que se incluyen «historias» a distinto nivel de abstracción en función de cada nivel: historias de usuario (a nivel de equipo), *features* (a nivel de programa) y epopeyas (a nivel de portafolio).

*Agile modeling*

*Agile modeling* (AM), es una técnica basada en la práctica para el modelado y la documentación efectiva de sistemas basados en *software.*

En un nivel alto, AM es una colección de catorce prácticas básicas, entre ellas: participación activa de los *stakeholders,* modelado en cada iteración, TDD y documentación continua.

En un nivel más detallado, AM es una colección de cinco valores (comunicación, simplicidad, retroalimentación, coraje y humildad), principios y prácticas para modelar *software,* que se pueden aplicar en un proyecto de desarrollo de *software* de una manera efectiva y ligera (Ambler, 2022a).

*Agile unified process* (AUP)

*Agile unified process* (AUP) es una versión simplificada de *rational unified process* (RUP). Describe un enfoque simple y fácil de entender para desarrollar *software* usando técnicas y conceptos ágiles, pero manteniéndose fiel a RUP (Ambler, 2022b).

AUP aplica características ágiles, como el desarrollo dirigido por pruebas, el desarrollo dirigido por modelos ágiles (AMDD), la gestión de cambios ágil y la refactorización de bases de datos para mejorar la productividad.

9.7. Relación entre técnicas ágiles

En muchas ocasiones es posible hacer uso de técnicas y enfoques ágiles en conjunto. Esto quiere decir que, por ejemplo, no es necesario pensar, por separado, solo en base a *lean* o a *scrum*, sino que se pueden combinar.

*Scrum* indica una manera de concretizar y aplicar los principios del enfoque ágil, con comunicación fluida, simplicidad o entregas continuas. Sigue una técnica concreta indicando las reuniones a tener, cuánto deben durar, qué roles habrá en el proyecto, los artefactos, etc. Por otro lado, la técnica XP también propone agilismo con su concreción y, a la vez, tanto en *scrum* como XP, el equipo de desarrollo puede organizar el trabajo utilizando *kanban*, lo que permite visualizar las tareas en curso con facilidad.

Inclusive, sobre cualquiera de las combinaciones, se puede aplicar la filosofía *lean* de no realizar tareas que no aporten valor, de tratar de no tener despilfarros o desperdicios y de maximizar el valor del producto resultante. Por tanto, la unión de varias técnicas y enfoques ágiles es realmente útil en los proyectos a desarrollar.

Todo el enfoque aquí mencionado se engloba en el denominado pensamiento sistémico, en el que se trata de dar la mejor solución a un problema completo, haciendo una selección de mejores prácticas.

Por ejemplo, imaginemos un proyecto organizado en *sprints* de tres semanas, se aplica *scrum*, se usa un tablero *kanban* para gestionar tareas, se aplican algunas prácticas de la técnica XP y todo esto, en conjunto, hace que haya el menor desperdicio posible y que se aumente el valor final del proyecto resultante.

La unión de las diferentes técnicas y enfoques ágiles tiene grandes ventajas en el pensamiento del agilismo, la cuales se pueden concluir de siguiente manera:

* Las técnicas ágiles proponen minimizar la documentación y maximizar resultados tangibles.
* En una propuesta ágil la cuestión menos clara al inicio del proyecto es el alcance, pues se pueden dar cambios continuos.
* El liderazgo ágil se basa en la capacidad de autogestión de las personas con su trabajo, y no en un jefe de proyecto.
* El manifiesto ágil consiste en un documento con cuatro valores y doce principios usados como base para la filosofía de trabajo ágil, dando lugar a técnicas tales como: *scrum, kanban* o XP.
* Los valores ágiles consideran a las personas más importantes que los procesos, los productos más importantes que los documentos, la colaboración preferible a la negociación y la respuesta a cambios mejor que el seguir un plan predefinido.
* Algunos principios ágiles incluyen entregas en períodos cortos de tiempo, dar importancia a la simplicidad, potenciar la excelencia técnica o promover la comunicación cara a cara.*Scrum* se considera la técnica ágil más utilizada, basada en reuniones o ceremonias; *sprints* con duración de entre una y cuatro semanas; desarrollo iterativo e incremental; y tiene en consideración las figuras de *scrum master*, *product owner* y equipo de desarrollo.
* La técnica XP propone un conjunto de prácticas de ingeniería para el desarrollo de *software* ágil, contemplando entregas en periodos cortos.
* *Kanban* consiste en el uso de un tablero o panel de tareas para visualizar el trabajo en desarrollo, realizando un seguimiento de avance fluido y limitando el trabajo en curso.
* El enfoque *lean* propone hacer solo lo que sea útil, no generar despilfarros o desperdicios, evitando perder el tiempo en trabajos que no aportan valor real.
* Las técnicas ágiles pueden usarse en conjunto, seleccionando y adaptando las partes convenientes de cada enfoque para un proyecto concreto.

Más aún, se puede llevar a cabo la combinación de un enfoque tradicional adoptando ideas de una técnica o enfoque ágil.

Si lo tradicional está de un lado y lo ágil del otro, se puede tratar de encontrar un punto medio que resulte acorde a las necesidades, para así gestionar de manera adecuada el proyecto deseado. En este caso, podemos encontrar distintas situaciones de mezcla entre lo tradicional y lo ágil, tales como:

* Considerar planificaciones usando diagramas de Gantt y hacer uso de un tablero de trabajo *kanban*cada dos semanas, reflejando la distribución detallada del trabajo y siendo visible fácilmente por todos.
* Tener un proyecto basado en hitos con fechas ya fijadas, debido a que se requiere así al tratarse de un trabajo a integrar con otros proyectos o empresas, y que cada hito sea ejecutado como un proyecto ágil corto, basado en iteraciones.
* Gestionar un proyecto con enfoque tradicional, planteando los requisitos como historias de usuario, con especificaciones sencillas y criterios de aceptación totalmente claros por cada requisito.

En resumen, es bueno saber que no es necesario pensar solo en términos de hacer uso de enfoques tradicionales o ágiles por separado, ni en seguir una técnica única.

Más bien se trata de encontrar una buena mezcla que resulte adecuada al proyecto en cuestión, sacando lo mejor de cada enfoque y técnica, de manera que realmente se usen los principios como herramientas para el fin de gestionar un proyecto correcta y eficientemente.