TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Aplicación de patrones de diseño al desarrollo de software ágil

Autor
Christian Andrades Molina

Director
Manuel Isidoro Capel Muñon

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 6 de Septiembre de 2018

Aplicación de patrones de diseño al desarrollo de software ágil

Christian Andrades Molina

Palabras clave: desarrollo ágil, arquitectura de software, patrones, diseño, metodologias ágiles, atributos de calidad, manifiesto ágil, desarrollo iterativo, documentación.

Resumen

El desarrollo de software ágil se encuentra entre las tendencias del sector gracias a sus ventajas en términos de flexibilidad y evolución, priorizando la comunicación entre partes implicadas frente a la documentación. Ha surgido como un complemento a la forma de trabajar ya existente o incluso como una alternativa viable a dia de hoy convirtiendose sin duda en una visión que choca frontalmente con una forma de gestionar proyectos basada en el ciclo de vida en cascada del software. Sin embargo, toda solución ágil a un mecanismo lento pero fiable puede presentar problemas a largo plazo, en este caso el estancamiento que presentan en estas metodologías respecto del diseño arquitectónico. Una realidad que puede llegar a provocar problemas en el desarrollo y en el alcance de determinados atributos de calidad.

El proposito principal de este proyecto es definir un enfoque metodológico basado en el diseño a partir de patrones arquitectónicos y que pueda ser aplicable a cualquier metodología ágil para el desarrollo de un sistema software complejo. Para su definición abarcaremos toda la historia del desarrollo de software, las metodologias ágiles existentes hasta la fecha, análisis de patrones, principios de desarrollo, alternativas arquitectónicas y una propuesta final.

El objetivo final será disponer de unas bases consistentes para mejorar el desarrollo de software hacia una alternativa que fusione las ideas propias del agilismo con la propuesta más tradicional, comprobando que el diseño de la arquitectura y la documentación son posibles en metodologías ágiles.

Application of design patterns to agile software development.

Christian Andrades Molina

Keywords: agile development, software architecture, patterns, design, agile methodologies, quality attributes, agile manifest, iterative development, documentation.

Abstract

The development of agile software is one of the trends in the sector thanks to its advantages in terms of flexibility and evolution, prioritizing communication over documentation. It has emerged as a complement to today's way of working or even as a viable alternative to the present day, becoming without doubt a vision that clashes with a slower and more bureaucratic way of managing projects. However, any agile solution to a slow but reliable mechanism can present long-term problems, especially the stagnation in these methodologies around architectural design. This could lead to problems in the development and scope of certain quality attributes.

The main purpose of this project is to define a specific development methodology for the design with architectural patterns that can be applied to any agile methodology for the development of a complex software system. In order to explain this, we will cover the entire history of software development, the agile methodologies available to date, pattern analysis, development principles, architectural alternatives and a final proposal.

The final aim will be having a consistent basis to improve software development towards an alternative that combines the ideas of agility with a more traditional proposal, proving that the design of architecture and documentation are also possible in agile methodologies.

Yo, **Christian Andrades Molina**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75897720c, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Christian Andrades Molina

Granada a 6 de Septiembre de 2018 .

D. **Manuel I. Capel Tuñón**, profesor del **Departamento de Lenguajes y Sistemas Informáticos** de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Aplicación de patrones de diseño al desarrollo de software ágil***, ha sido realizado bajo su supervisión por **Christian Andrades Molina**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 6 de Septiembre de 2018 .

El director:

Manuel I. Capel Tuñón

Agradecimientos

En primer lugar a mis padres por haberme dado la oportunidad de cursar esta carrera en Granada, sin ellos habría sido imposible, pero sobre todo por apoyarme tanto en los mejores como los peores momentos. Jamás han fallado.

A mi tutor, por haberme dado muy buenas explicaciones y orientarme adecuadamente con referencias y correcciones muy detalladas.

Y por último a todos los compañeros con los que he compartido estos años de carrera y que he ido conociendo conforme avanzaba.

Índice general

Índice de figuras

Capítulo 1

Introducción del proyecto

1.1. Introducción

El desarrollo de software ágil continúa actualmente su difusión en la industria y, por tanto, determinadas tecnologías actuales como TDD, Continuous Integration, etc., que han sido introducidas desde hace ya una década por las prácticas de "Extreme Programming", se usan actualmente de forma generalizada en la industria del software. La programación extrema es la metodología de desarrollo más destacada de los procesos ágiles, siendo de las primeras en aplicar las características fundamentales del desarrollo ágil como la simplicidad o la adaptabilidad.

Actualmente, el desarrollo de sistemas software complejos se realiza mediante la aplicación de un conjunto de técnicas de programación y técnicas de prueba del software normalmente, que se gestionan de una forma ágil (Scrum, Kaizen, Six Sigma, etc.), pero que no se pueden denominar propiamente como "metodologías ágiles" de desarrollo de software. Existe en la actualidad un recelo en el uso de patrones de diseño en el desarrollo de software ágil por parte de los entornos más adaptados a un diseño emergente por la contradición de su uso al cambio continuo planteado en los procesos ágiles. Estos proponen arquitecturas de carácter evolutivo; es decir, no se diseñan completamente antes de comenzar a escribir código como era habitual si se aplica el modelo de desarrollo de software tradicional lineal. No obstante, el rechazo a los patrones no es una idea totalmente generalizada, y dentro del ámbito de desarrollo de software los patrones de diseño sí son aceptados.

Por tanto, tenemos por una parte el concepto de arquitectura de software, el cual nos proporciona una visión global de la estructura del proyecto y sirve para predecir la calidad del software, con las ventajas lógicas de un proceso de alto nivel en términos de reutilización y evolución; por la otra

tenemos el uso de metodologías ágiles que ofrecen eficacia y flexibilidad en el desarrollo de dicho software. El objetivo del proyecto es tratar de unificar ambos conceptos en una única metodología, aportando un método para conseguir la integración de las características de ambas ideas, demostrando así que partir de una base sólida, propiciada por el uso de patrones de diseño y arquitectónicos, es la mejor práctica para el desarrollo de un producto software.

Nuestro propósito final es incluir las actividades de diseño de arquitecturas de software en una metodología ágil conocida y que pueda ser aplicado a otras metodologías de tendencia agilista actuales, sin dejar de lado las consideraciones de calidad, riesgo y costos asociados a la definición de una arquitectura de software previa al resto de tareas de desarrollo del producto software.

1.2. Motivación

El presente proyecto se dispone como una propuesta de enfoque correcto para abordar el problema de compensar la carencia de un diseño arquitectónico, previo al desarrollo del proyecto, que, generalmente, se puede observar dentro de las técnicas desarrollo de software ágil actuales. Pretendemos mostrar en este trabajo que ambos enfoques no son incompatibles.

Para conseguir el objetivo anterior, este proyecto quiere proponer un patrón o patrones de diseño arquitectónico que permitan integrar su diseño en las tareas de desarrollo de software en el menor tiempo posible y no sea incompatible con las características del movimiento ágil, que suele ser la tendencia que ha adquirido mayor interés actualmente.

Las metodologías ágiles, al contrario de las más tradicionales, que cuentan con un enfoque más predictivo, recelan del uso de un diseño que implique prever necesidades futuras mediante la implementación de un diseño arquitectónico completo [35].

Por otra parte contar con las arquitecturas, en el desarrollo de software, están cobrando cada vez mayor importancia ya que su uso propicia determinados atributos de calidad, y mitiga riesgos, en el software que se desarrolle posteriormente. Esto ha motivado que se propongan distintas opciones para la integración de métodos, técnicas y patrones arquitectónicos en procesos de desarrollo agilistas. Sin embargo, estos patrones arquitectónicos parecen propiciar la necesidad de invertir grandes cantidades de tiempo para corregir la arquitectura software de un sistema en respuesta a requerimientos cambiantes del mismo, debido a que la primera no fue diseñada para tal fin.

1.3. Objetivos

Los objetivos que vamos a tratar de alcanzar en el desarrollo de este proyecto son los siguientes:

- **Apoyar los aspectos generales relevantes en las buenas prácticas de desarrollo de software**, a través de la integración del uso metodologías clásicas con implicación fundamental del diseño arquitectónico y la implantación de metodologías ágiles, que se utilizan cada vez más en la actualidad.
- **Examinar de manera crítica las discrepancias de desarrolladores actuales respecto del modelo de desarrollo tradicional lineal**, ya que los primeros están a favor de un desarrollo ágil puro, que puede conllevar una toma irreversible de decisiones tempranas en el diseño del software.
- **Analizar con detalle los patrones arquitectónicos existentes**, ya que su uso propicia los atributos de calidad del diseño obtenido y el producto software final.
- **Seleccionar un conjunto de patrones de diseño y arquitectónicos que permitan su implementación sobre un entorno de desarrollo ágil.**
- **Implantar la arquitectura construida a partir del conjunto de patrones** utilizando para ello una metodología de desarrollo ágil, tal como Scrum.

1.4. Estructura del proyecto

Introduciremos en primer lugar un marco histórico que aporte los hitos más importantes y los conceptos relevantes del desarrollo de software. Haremos un inciso en los conceptos de desarrollo ágil y arquitecturas de software como contexto a una metodología completa que aporte ambas ideas. Posteriormente nos centraremos en el debate sobre el diseño arquitectónico sobre metodologías ágiles para acabar desarrollando el nuevo método a partir de patrones de diseño y su implementación y validación correspondiente.

Capítulo 2

Metodología ágil y la arquitectura de software

2.1. Historia del desarrollo de software

Desde los inicios, el desarrollo de software se ha convertido en un procedimiento complejo y en ocasiones tedioso debido a los tiempos interminables para completar las distintas etapas del proyecto, que establece el modelo de desarrollo tradicional lineal del software. Cambios en la idea inicial o incluso una previsión errónea de la duración de dichas etapas generaba costes, sobretodo en tiempo y dinero, que ocasionaban que un proyecto se convirtiera en un dolor de cabeza. Esta forma de trabajar se alineaba con el modelo de ciclo de vida 'desarrollo en cascada' del software. Esta metodología de desarrollo partía de un progreso lineal, elaborando las distintas etapas secuencialmente: requisitos, diseño, implementación, verificación y mantenimiento. Cada fase da lugar a la siguiente, pero solo una vez finalizada la anterior, convirtiéndose en una forma de trabajo muy estructurada y fácil de implementar pero a su vez lenta y sin margen de error. Es decir, cualquier error generado en alguna de sus etapas podría provocar un paso atrás importante. Sin embargo, los avances lógicos de tiempo provocaron que esta manera de desarrollar software no aportara lo necesario para nuevos proyectos que requerían más flexibilidad y adaptación.

El desarrollo ágil o las distintas metodologías orientadas a la agilización del proceso, se remonta a finales de los años 30:

1. En 1939, Walter Shewhart -conocido como "El padre de la calidad"- publica en su libro Statistical Method from the Viewpoint of Quality Control el proceso de mejora continua y trabajo iterativo e incremental a través de ciclos cortos de "Planificar, Hacer, Analizar y Actuar (plan-

do-study-act)". Un concepto que luego fue difundido por Deming.

2. En 1948, el ingeniero Taiichi Ohno comenzó a desarrollar Kanban en Japón (Toyota), un sistema que pone en marcha el "método del Supermercado", debido a que la idea fue prestada por las grandes tiendas de mercadeo masivo. Kanban en su traducción literal del japonés es "señal visual" o "tarjeta". Los trabajadores usaban una tarjeta para marcar los pasos en el proceso de fabricación. Se trataba de un primer paso de acercamiento a un desarrollo controlado.

3. En 1951, William Deming, contribuyó a la expansión del método de desarrollo IIDD (Iterative and Incremental Design and Development) a través de la divulgación del 'Planificar, Hacer, Analizar y Actuar'.

4. En 1955, comenzó a usarse el proceso IIDD en el desarrollo de software.

5. En los años 70 y 80, el modelo de desarrollo en cascada tomo sus primeros pasos, definiéndose su nombre y sus fases.

6. En 1986, se registra el uso del término SCRUM como analogía del rugby en el desarrollo en un artículo titulado "The New Product Development Game" de la Harvard Business Review.

7. En 1992, Alistair Cockbur publica 'Los Métodos Crystal', un punto de partida para la evolución de las metodologías de desarrollo de software conocidas hoy en día como movimiento ágil.

8. En 1993, Bill Opdyke presenta el concepto de 'Refactorización', una técnica para la reestructuración de piezas de código existente, alterando su estructura interna sin afectar su comportamiento con el exterior

9. A partir de 1995, comenzaron a surgir numerosas ideas de metodologías que apostaban por un desarrollo que se distanciará del estandar como la Programación por Parejas, un método de desarrollo de sistemas dinámicos, Scrum, el desarrollo adaptativo del software, un primer acercamiento a la metodología XP, etc.

10. En el año 2001, Bob Martín reunió a otros 16 líderes del movimiento ágil en Snowbird, Utah (US), para escribir el 'Manifiesto Ágil' [4], que engloba las metodologías que hasta ese momento se les conocía como "Metodologías de Desarrollo de Software de peso liviano". Más tarde formaron la alianza ágil, que promovía el uso del desarrollo ágil en aplicaciones. Este manifiesto compuesto por cuatro valores y doce principios, estaba formado por los siguientes integrantes: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas.

Figura 2.1: Los 17 integrantes del Manifiesto Ágil

2.2. ¿Qué es el 'Desarrollo Ágil'?

Una vez definidos los pasos dados por la industria en el desarrollo de software, con un predominio inicial de iniciativas orientadas al modelo de desarrollo en cascada y un cambio posterior hacia mecanismos que agilizaran este proceso, surge lo que es comúnmente conocido a día de hoy como el desarrollo de software ágil. Este concepto promueve la realización del trabajo en equipos cuya organización y disciplina es autónoma, permitiendo que la comunicación entre sus integrantes sea total en cada una de las fases del proyecto. Esto a su vez conlleva una revisión permanente del mismo ante posibles cambios, evitando, de esta forma, realizar reuniones en procesos finales del trabajo sin la seguridad de que el resultado sea cumplir todos los requisitos del software que se está desarrollando.

En cada iteración del ciclo de vida del software se incluyen tareas de planificación, análisis de requisitos, diseño, codificación, pruebas/test y documentación. En cada paso el equipo de desarrollo añade pequeños elementos totalmente funcionales y que cumplen con todos los requisitos previos, obteniendo un conjunto consistente y sin errores.

Respecto de la localización y composición de los equipos de trabajo ágil, estos se localizan en oficinas abiertas, llamadas 'plataformas de lanzamiento' que están formados por revisores, escritores de documentación y ayuda, diseñadores de iteración y directores de proyecto.

Existen varias metodologías de desarrollo, las cuales se basan en el Manifiesto Ágil, basado en la experiencia de lo que funciona y no funciona en dicho campo de la Ingeniería de Software. Estas comparten la idea de un desarrollo iterativo, comunicativo y con la premisa de eliminar el mayor número de trabas que consuman recursos innecesarios y que, a su vez, poseen varias diferencias para llevarlos a la práctica. En definitiva, las metodologías ágiles aportan dinamismo y flexibilidad a proyectos con el mínimo impacto.

2.2.1. Principios fundamentales del 'Desarrollo Ágil'

La publicación del manifiesto dio origen a 12 principios a cumplir en cualquier desarrollo de software ágil:

1. La prioridad del grupo de trabajo es satisfacer las necesidades del cliente mediante la entrega temprana y continua de productos de valor que cubran una o varias necesidades.

2. Aceptación de que los requisitos iniciales pueden variar durante el desarrollo sin afectar a la escala de tiempo y fechas de entrega. La inclusión de nuevos requisitos que mejoren el producto hasta la fecha, son bienvenidas y pueden ser aprovechadas para proporcionar ventaja competitiva al cliente.

3. La entrega frecuente (entre 2 semanas y 2 meses) de software funcionante es fundamental. Es la base fundamental de la metodología.

4. La forma de obtener una medida o indicador del progreso es el funcionamiento óptimo del software.

5. Los procesos ágiles promueven el desarrollo sostenible. Se debe mantener un ritmo constante aplicando la regla 80/20.

6. Cooperación entre miembros de equipo en entornos compartidos. Tanto lideres del proyecto como desarrolladores deben ejercer sus tareas interaccionando entre ellos de forma cotidiana.

7. Comunicación cara a cara. Priorizar las reuniones periódicas entre clientes y colaboradores frente al desarrollo de una extensa documentación.

8. Mantener la motivación y la confianza del equipo. Para que el proceso de elaboración del proyecto obtenga la mayor calidad posible, es indispensable trabajar sobre un grupo de personas motivadas y que promuevan un clima de solidaridad y confianza.

9. Buscar la excelencia en todos los ámbitos posibles. Un buen diseño junto a una excelencia técnica mejora la agilidad. El producto final se verá recompensado.

10. Las tareas deben de ser lo más sencillas posibles. Es fundamental maximizar la cantidad de trabajo no realizado. Si la tarea adjudicada posee demasiada complejidad, es posible dividirla en iteraciones más pequeñas.

11. Autogestión de los equipos, tanto en organización como en funciones. Los equipos deben reflexionar sobre cómo pueden ser más efectivos para ajustar las prioridades y, consecuentemente, conseguir un procedimiento más útil. Así mismo, también podría existir una figura que monitorice el trabajo del equipo, pero sin excederse en sus funciones.

12. Adaptarse a las circunstancias imprevistas que puedan ocurrir durante el desarrollo.

2.2.2. Ventajas

Si optamos por el uso de una metodología ágil, el progreso del proyecto obtendrá una serie de ventajas como consecuencia de que cada uno de sus elementos puede desarrollarse de forma paralela. Entre dichos elementos podemos encontrar:

- Mayor implicación del cliente en el proceso de desarrollo. El cliente podrá observar como avanza el proyecto e, incluso, aportar su opinión ofreciendo ideas en las distintas reuniones entre el equipo y él. Esto provoca una mayor satisfacción del mismo al verse involucrado a lo largo de las distintas etapas del desarrollo.

- La entrega de productos parciales para valorar los recursos utilizados y los requisitos cumplidos permiten un mejor seguimiento y control. El producto final es la suma de todas sus partes sin errores.

- Respuesta rápida a cambios e incluso en la predicción de ellos. Al ser un proceso evolutivo, los miembros de equipo pueden valorar su trabajo y mejorar aquellas áreas más vulnerables.

- Uno de los aspectos más importantes es la reducción de costes y tiempo mediante la eliminación de tareas innecesarias. Ofrecen una distinción más clara de las tareas en las que se deben focalizar los esfuerzos y permiten identificar errores en etapas tempranas.
- Las metodologías ágiles ofrecen un enfoque proactivo de los miembros del equipo en la búsqueda de la excelencia del producto. El trabajo colaborativo fomenta que el grupo posea una mejor organización de su trabajo.
- La temporización del proyecto es más realista que en otras metodologías de trabajo. El progreso de desarrollo se va decidiendo fase a fase, siendo consciente el equipo del trabajo realizado y del que queda por hacer.

En resumen. Con una metodología ágil, se prevé obtener mejores resultados, en menos tiempo y con mayor capacidad de adaptación, que si utilizamos un modelo desarrollo lineal, consiguiéndose un mayor control del cliente sobre su producto, una mejora sustancial de calidad del mismo y brinda al equipo la posibilidad de reaccionar sin problemas a cambios de última hora.

2.2.3. Inconvenientes

Si bien el uso de metodologías ágiles proporciona una serie de ventajas frente a otras formas de desarrollo, como las que se basan en el ciclo de desarrollo en cascada, existen varios factores que pueden provocar que su utilización, con respecto a un modelo tradicional lineal, sea contraproducente:

- El uso de un equipo poco experimentado: a causa de la inexperiencia del equipo, unida a un cliente impaciente, podría crear problemas durante un desarrollo de este tipo si se cuenta con tiempos de desarrollo medidos. Un buen conocimiento de la praxis de esta metodología por parte del equipo es esencial para obtener éxito.
- Esa insuficiencia de conocimiento es otro de los problemas que se pueden generar. Una posible solución sería impartir cursos donde profundizar en los mecanismos que caracterizan un desarrollo ágil y ejemplificarlo llevando a cabo proyectos sencillos en la práctica.
- Si el cliente no participa activamente en el proyecto, provoca que los requisitos a cumplir en cada etapa sean cada vez más difusos. Su im-

plicación es esencial en un desarrollo ágil para estructurar el trabajo y conseguir resultados correctamente.

- La aplicación de una metodología ágil promueve que los miembros del equipo aspiren a conseguir la máxima calidad posible del producto, asumiendo en ocasiones riesgos innecesarios porque no calibran bien los medios que se disponen para obtenerlo con éxito. Si no se emplean técnicas de gestión de riesgos dentro del desarrollo ágil, los problemas no tardarán en aparecer durante el proceso de desarrollo.

- Tomar una actitud egoísta, ya sea motivada por un nivel de experiencia mayor que el resto de miembros del equipo o por imprudencia personal, provoca la toma de decisiones individualmente que no cuenta sin el apoyo del resto del grupo. Esto es una violación clara de los principios del desarrollo ágil.

- La comunicación entre miembros es uno de los pilares fundamentales en este tipo de desarrollo. Si existe algún problema en este sentido, como comunicación en idiomas muy distintos o la lejanía entre desarrolladores, se terminarán produciendo problemas de difícil manejo.

- Calidad inferior a la esperada. Si decidimos llevar a cabo un proyecto mediante el uso de una metodología ágil, su correcto uso, normalmente, conlleva una mejora de la calidad del producto. Sin embargo, esto no podemos ver como una resultado generado automáticamente como consecuencia de utilizar un método ágil. Para propiciarla, el grupo de trabajo debe elaborar una serie de condiciones de prueba que permitan alcanzar la calidad deseada del sistema final.

- Síndrome del Burn Out. Trabajar con métodos ágiles puede aumentar el estrés de algunos de los desarrolladores debido a los exigentes requisitos solicitados de cada uno en cada fase. Esta circunstancia puede detectarse a tiempo y, por tanto, hay que mantener los ojos bien abiertos antes de que se vean afectados los resultados y cunda la desmotivación entre el resto del equipo.

- Intentar abarcar demasiado en una sola iteración, este error de planificación, a parte de generar un estrés extra, puede provocar un caos en el ciclo de desarrollo dando lugar a etapas sin principio ni final. Una perniciosa consecuencia de este defecto consiste en la denominada "vuelta atrás", es decir, tener que volver a recuperar los procesos ya considerados como terminados. Es preciso controlar que no se llegue a este punto.

- Falta de documentación. Al no haber documentación es el código (junto con sus comentarios) lo que se entiende como documentación del

software. La comprensión holística del sistema en este caso se queda en las mentes de los desarrolladores. Las metodologías ágiles no plantean alternativas para mantener accesible con facilidad la documentación de los proyectos. Simplemente indican la manera cómo se llevarán a cabo las diferentes acciones durante el proceso de desarrollo, evitando generar documentos que considera innecesarios al finalizar cada etapa. Respecto a este inconveniente, debemos mencionar que la elaboración de una arquitectura software completa, aún evolucionando conforme avanza el proceso de desarrollo, puede solucionar satisfactoriamente la objeción de no contar con documentación del software.

Finalmente, comentaremos que, en cualquier caso, probar el software es la mejor forma de averiguar si ya está preparado para cumplir con los requerimientos de un proyecto apoyado por los métodos ágiles. No obstante, el abuso de esta práctica, sin que el equipo de desarrollo tenga experiencia previa en la prueba de software, no es lo más conveniente.

2.3. Metodologías ágiles más conocidas

Actualmente se conocen diferentes metodologías relacionadas con el enfoque ágil para entregar productos de calidad en el menor tiempo y costo posible. A día de hoy la metodología ágil más popular para la gestión de proyectos es Scrum, pero también existen otros métodos, entre ellos, cabe mencionarse a los siguientes:

2.3.1. Agile Unified Process (AUP)

ASD (Adaptive Software Development) es una técnica que fue desarrollada por Jim Highsmith y Sam Bayer a comienzos de 1990, que incorpora el principio de la adaptación continua, o sea, adaptarse al cambio y no luchar contra él. De acuerdo con el principio del desarrollo ágil, no existe un ciclo estático sino que cada etapa puede ser modificada al tiempo que otra es ejecutada. Las características que definen esta técnica son las siguientes: se trata de un método iterativo, tolerante a los cambios, que se guía por los riesgos, orientado a los componentes del software y los revisa para aprender de los errores.

El ciclo utilizado por ASD es conocido como: Especular-colaborar-aprender, el cual está supone realizar un constante aprendizaje y establecer una colaboración entre desarrollador y cliente. Estos tres componentes se definen de la siguiente manera:

Especular:

- Establecemos los objetivos y metas a cumplir, teniendo en cuenta los riesgos del proyecto en cuestión.
- Intentamos obtener una estimación del tiempo del proyecto para llevarlo a cabo dentro de lo razonable.
- Decidiremos el número de iteraciones (fases) del proyecto, teniendo en cuenta las funcionalidades que el cliente ha de realizar al finalizar cada etapa.
- Repetición de los tres pasos anteriores, si es necesario, para llegar a un punto en común.

Colaborar:

- Proceso intermedio del proyecto en el cual se desarrolla gran parte del trabajo mediante cooperación entre los miembros del equipo y con el resto de las partes implicadas.

Aprender:

- En la última etapa se recoge toda la información generada durante el desarrollo del proyecto, para su aprendizaje por parte del equipo, tanto la información positiva como negativa. Existen cuatro tipos de aprendizaje que se puede llegar a tener en este punto:
- Calidad del producto desde el punto de vista del cliente, mediante su valoración con una perspectiva funcional.
- Calidad del producto desde el punto de vista de los desarrolladores, mediante su valoración con una pesperctiva técnica.
- Funcionalidad desarrollada y evaluación de lo aprendido durante el trabajo, mediante su valoración por los integrantes del equipo.
- Estado del proyecto, obteniendo un punto de partida común a partir de todo lo anterior, para comenzar la construcción de la siguiente fase, este estadio pasa a convertirse en el inicio de un bucle de desarrollo proactivo.

Ventajas :

- Se adquiere un modo de trabajo eficaz, con un aprendizaje continuo de los errores cometidos.

- Promueve la colaboración entre compañeros.
- A partir de la información obtenida, se mejora sustancialmente el software.
- Apunta hacia el Rapid Application Development (RAD), el cual enfatiza velocidad de desarrollo para crear un producto de alta calidad.

Desventajas : Como principal objeción, tendremos:

- Los errores cometidos o los cambios no previstos con anterioridad, podrían pasar factura en la calidad y costo final del proyecto.

2.3.2. Extreme Programming (XP)

La programación extrema es un enfoque de la ingeniería de software propuesto por Kent Beck, autor de la obra inspiradora de dicho enfoque: "Extreme Programming Explained: Embrace Change" (1999). XP es uno de los procesos ágiles más importantes del desarrollo de software, que pone el acento en las relaciones personales de los miembros del equipo como clave para el éxito en el desarrollo de un software; para conseguirlo, promueve el trabajo en equipo, se preocupa por el aprendizaje de los desarrolladores, y propicia un buen clima de trabajo.

La metodología XP se basa en cuatro valores:

- **Simplicidad**, proponiendo la realización de etapas funcionales más cortas, refactorizar el código (restructurar el código fuente) para mejorar la facilidad de comprensión del código o cambiar su estructura y manejar una documentación consensuada por todos sin excederse en su tamaño.
- **Comunicación**, definiendo las necesidades del cliente y transmitiéndolas a un grupo de trabajo pequeño pero constituido por miembros capaces y comunicativos.
- **Realimentación**, esta característica se basa en una metodología del tipo „prueba-error" y en la capacidad de compartir conocimiento entre miembros de equipo.
- **Decisiones tomadas con valentía**, proponiendo el uso de las mejores prácticas posibles de Ingeniería de Software en el desarrollo.

Existen doce prácticas básicas que deben seguirse para llevar a cabo una programación extrema, acorde a su función, que serían las siguientes:

1. Equipo completo.
2. Planificación del trabajo de cada desarrollador, con revisiones continuas.
3. Uso de un test del cliente para validar cada versión producida.
4. Implementación de pequeñas versiones.
5. Diseño simple y en continua mejora.
6. Organizar en parejas a los programadores.
7. Desarrollo guiado por las pruebas automáticas.
8. Integración continua de las mejores propuestas.
9. El código es de todos.
10. Normas de codificación comunes.
11. Frases cortas para definir partes del programa de forma rápida.
12. Ritmo sostenible, sin parones de tiempo excesivos.

Ventajas :

- Reducción en la tasas de errores y programación más organizada.
- El programador consigue una mayor satisfacción en su trabajo.

Desventajas : Como principal objeción, tendremos:

- Resulta recomendable exclusivamente en proyectos con menor tiempo de desarrollo debido a las altos costes en caso de fracasar.

2.3.3. Kanban

Kanban nació en 1948 a manos del ingeniero Taiichi Ohno en Japón, dentro de la compañía Toyota para controlar el avance del trabajo en el contexto de una línea de producción. Es decir, no es una técnica aplicable al desarrollo de software, propiamente, sino que su objetivo es gestionar de manera general cómo se van completando tareas dentro de una línea de producto, pero en los últimos años se ha utilizado en la gestión de proyectos de desarrollo de software.

Al igual que otros desarrollos ágiles, Kanban se basa en un desarrollo incremental mediante la división del trabajo en varios partes. La característica diferencial con otras metodologías es su concepto visual. A partir de un tablero Kanban, los miembros de todos los equipos pueden visualizar el trabajo y optimizar el flujo entre ellos. Las tareas en las que se divide el proyecto se escriben en un post-it y se pega en una pizarra o se crea una sección en una tabla virtual. Cada post incluye una descripción del mismo y el número de horas estimada. La tabla se divide en el número de procesos por los que pasará la tarea.

Por tanto, ofrece una perspectiva distinta a otras opciones, permitiendo realizar un seguimiento del trabajo, identificar cuellos de botella y obtener una lectura fácil mediante indicadores visuales.

Ventajas :

- Se reducen los tiempos de entrega, ciclos de producción y planificación.
- Se incrementa la productividad y la fiabilidad de las entregas parciales.
- Es más fácil llevar un control del sistema informático, mediante elementos visuales.

Desventajas : Como principal objeción, tendremos:

- No se definen las iteraciones y se limitan las tareas que se pueden realizar por cada fase

2.3.4. Feature Driven Development (FDD)

En este caso, hablamos realmente de una meta-metodología, es decir, se trata de un enfoque metodológico que necesita ser adaptado a cada caso concreto.

FDD con sus siglas en inglés Feature Driven Development es un enfoque ágil para el desarrollo de sistemas. FDD fue desarrollado por Jeff De Luca y Peter Coad, éste último es un viejo gurú de la orientación a objetos. El libro que mejor describe la metodología es 'A Practical Guide to Feature-Driven Development'.

Como ocurre con otras aproximaciones al desarrollo ágil, FDD se enfoca en iteraciones cortas que van produciendo cada vez más funcionalidad tangible del sistema objetivo. Incluye una monitorización permanente durante cada una de las fases del proyecto y sus autores afirman que resulta práctico para el desarrollo de sistemas críticos.

La utilización de FDD define 5 procesos fundamentales. Los tres primeros se consideran los procesos iniciales y dentro de ellos, los dos primeros que se llevan a cabo secuencialmente, definen el modelo global. Los tres últimos se iteran para cada requisito.

- **Proceso 1**. Desarrollar el modelo global (Develop overall model): FDD no hace énfasis en la recolección y administración de los requerimientos. Los expertos del dominio presentan un informe llamado walkthrough (o "revisión sistemática") realizado para que los miembros del equipo de diseño y el Arquitecto de Software Jefe sean informados mediante una descripción de alto nivel del sistema. Para realizarlo, se divide el dominio global en varias áreas y se especifica cada una de ellas detalladamente. Posteriormente, se obtendrá un modelo de objetos para cada área y, posteriormente, se obtiene el modelo global.
- **Proceso 2**. Construir una lista de características (Build feature list) a partir de los modelos de objetos, requerimientos y walktroughts. Se presenta una lista de funcionalidades por área ofrecidas al cliente y se consigue una lista general dividida en subconjuntos según el tipo de funcionalidad.
- **Proceso 3**. Planificar (Plan by feature) las funciones según una prioridad.
- **Proceso 4 y 5**. Diseñar (Design by feature) y construir (Build by feature): este proceso iterativo incluye tareas tales como inspección del diseño, codificación, testing unitario, integración e inspección del código. Luego que la iteración llega a su fin se realiza un main build (o "encuadernado") en el cual se integra toda la funcionalidad deseada. Dicho encuadernado se realiza mientras comienza la siguiente iteración.

Ventajas :

- Trabajo en conjunto entre el cliente y el equipo

- Minimiza los costos frente a cambios.
- No se malgasta dinero y tiempo en soluciones complejas que no se ajusten a los requisitos establecidos.
- Se busca la simplicidad y la excelencia técnica.

Desventajas : Como principal objeción, tendremos:

- Como otras metodologías, la documentación carece de protagonismo.
- Por tanto, el código se vuelve dificil para ser reutilizado.
- Esta enfocado a proyectos críticos en iteraciones cortas.

2.3.5. Lean Software Development (LSD)

El término Lean Software Development (o „Desarrollo de Software Ajustado“) se acuñó por primera vez como título de una conferencia organizada por la iniciativa ESPRIT de la Unión Europea, en Stuttgart, Alemania, en octubre de 1992. Independientemente, al año siguiente, Robert “Bob” Charette en 1993, planteó el concepto de “Lean Software Development” como parte de su trabajo para investigar mejores formas para administrar los riesgos en proyectos de software. Al igual que Kaban, la metodología Lean también tiene su origen en Toyota.

Que Lean podría aplicarse al desarrollo de software es una idea que se estableció muy temprano, solo 1 o 2 años después de que el término se utilizara asociado a tendencias en procesos de fabricación e Ingeniería Industrial.

El desarrollo de software bajo la filosofía Lean se puede resumir en siete principios:

1. Eliminar desperdicios/restos como código generado sin una funcionalidad solicitada.
2. Amplificar el aprendizaje mediante una mentalidad de aprendizaje continuo.
3. Tomar decisiones lo más tarde posible debido a posibles cambios de última hora del cliente en sus requisitos.
4. Entregar lo antes posible, cumpliendo uno de los principios del desarrollo ágil de realizar entregas periódicas.

5. Potenciar el equipo mediante la participación de los desarrolladores en tomas de decisiones.
6. Crear la integridad mediante un sistema de integración continua que incluya pruebas automatizadas, builds o pruebas de usabilidad.
7. Visualizar todo el conjunto a través de la consigna "Pensar en grande, actuar en pequeño, equivocarse rápido y aprender con rapidez".

Ventajas :

- Eliminación de componentes residuales que aceleran el proceso de desarrollo del software
- Entregas periódicas con un mayor perfeccionamiento en las funcionalidades.
- El equipo de desarrollo tiene un mayor control.

Desventajas : Como principal objección, tendremos:

- Alta dependencia en la cohesión y disciplina del equipo para llegar a conseguir objetivos comunes.
- El cliente debe saber que necesita en su proyecto y reaccionar tomando las decisiones adecuadas.

2.3.6. Open Unified Process (OpenUP)

OpenUP (Open Unified Process) es un método de desarrollo de software propuesto por un conjunto de empresas quienes donaron el proceso en 2007 a la Fundación Eclipse. Se trata de un marco de trabajo de fuentes abiertas, útil para poder utilizar el núcleo de Rational Unified Process (RUP). Actualmente, OpenUP se encuentra bajo una licencia de software libre. Esta metodología ofrece un desarrollo de software que cumpla con un conjunto mínimo de los elementos de RUP, descartando elementos como el tamaño del equipo o la seguridad. Por tanto, OpenUP puede ser utilizado como una metodología complementaría a otra más completa (por ejemplo, RUP) si es necesario. La mayoría de los elementos de OpenUP están declarados para fomentar el intercambio de información entre los equipos de desarrollo y mantener un entendimiento compartido del proyecto, sus objetivos, alcance y avances.

Entre los principios que conforman el proceso, encontramos:

- Colaboración entre miembros para coordinarse y compartir intereses y conocimiento.
- Llegar a un equilibrio en las prioridades para conseguir el máximo beneficio.
- Centrarse en la arquitectura desde un inicio para reducir riesgos y minimizar el trabajo extra.
- Desarrollo evolutivo para obtener retroalimentación.

Ventajas :

- Posee las mismas ventajas que ofrece un desarrollo ágil.
- Es adaptable como metodología secundaria a otra principal.

Desventajas : Como principal objección, tendremos:

- Se centra en un estructura mínima, sin capacidad para profundizar.

2.3.7. Dynamic Systems Development Methods (DSDM)

DSDM (Dynamic Systems Development Methods) es un método que provee un framework para el desarrollo ágil. Fue desarrollado en Reino Unido durante los años 90 por un conjunto de proveedores y expertos en IS (Ingeniería de Software) denominados como el consorcio de DSDM. Esta compañía es una organización no lucrativa independiente, que posee y administra el marco de trabajo. La primera versión fue terminada en enero de 1995 y publicada en febrero de 1995. La versión actualmente en uso (abril de 2006) es la versión 4.2.

Es una metodología de desarrollo de software originalmente basada en la metodología RAD (Desarrollo Rápido de aplicaciones) [5], un proceso de desarrollo de software creado inicialmente por James Martin en 1980. En DSDM se lleva a cabo un procedimiento reiterativo e incremental cuyo objetivo es entregar el proyecto a tiempo y dentro del presupuesto indicado, para lo cual se tienen en cuenta que los requisitos pueden cambiar.

Actualmente está compuesto por los siguientes ocho principios:

1. Focalizarse en las necesidades del cliente y por tanto del proyecto, ofreciendo entregas funcionales durante el desarrollo.

2. Entregar a tiempo el trabajo es sumamente importante. Se considera primordial no fallar en las fechas.

3. La colaboración entre los miembros del equipo es indispensable para ser más eficientes.

4. Jamás comprometer la calidad. El nivel de calidad debe ser acordado al principio y cumplirse al final sin comprometer las fechas estimadas.

5. Establecer un equilibrio entre el problema y las soluciones planteadas.

6. Entrega frecuente, revisión y la realimentación de los entregables, forma parte del enfoque de desarrollo de la solución.

7. Comunicación continua y fluida. Priorizar las conversaciones cara a cara.

8. Demostrar un control sobre el trabajo, mostrando el progreso realizado en forma de entregas y con transparencia.

Ventajas :

- La calidad del producto mejora a partir de la participación de los usuarios.
- Su mejor atributo es la rapidez de procesos.
- Reduce los costos de proyectos
- Permite una mayor reutilización de código al contrario que otras metodologías.

Desventajas : Como principales objeciones, tendremos:

- La participación activa de los usuarios es necesaria o el desarrollo puede estar comprometido.
- Es complejo de aplicar y entender.

2.4. Scrum

Scrum es un marco de trabajo para el desarrollo y el mantenimiento de productos complejos que permite la implementación de metodologías ágiles, pero no se considera como una forma de desarrollo ágil propiamente dicha. De hecho, se pueden usar muchos marcos de trabajo para implementar una metodología ágil, como por ejemplo Kanban; sin embargo, Scrum posee unas connotaciones únicas que se derivan de su compromiso con la realización del trabajo a través de iteraciones breves. Hoy en día podemos encontrar Scrum en multitud de empresas de diferentes sectores como:

- Software/Hardware: Adobe, Citrix, IBM, Intel, Microsoft o Softhouse.
- Industria del entretenimiento: Blizzard, Crytek, Ubisoft o EA.
- Internet: Amazon, Google o Yahoo.

2.4.1. Origen de Scrum

El origen de Scrum se remonta a principios de los años 80, a partir de un estudio realizado en 1986 por Hirotaka Takeuchi y Ikujiro Nonaka en la obra ' The New New Product Developement Game' [6]. En él, se analizaban los procesos de desarrollo utilizados en las principales empresas de manufactura tecnológica de productos exitosos en Japón y en los Estados Unidos: Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M y Hewlett-Packard.

Estos equipos de desarrollo de renombre partían de requisitos generales para lanzar al mercado productos en un tiempo inferior a lanzamientos anteriores. En el análisis, Takeuchi y Nonaka comparaban esta forma de trabajar con la colaboración para avanzar en el rugby en forma de melé (una formación típica de este deporte), o "Scrum" en inglés, que es el término que acabó recibiendo esta metodología.

Realmente no es un método de trabajo que se considere exclusivo de una área en concreta, sino que es capaz de adaptarse a cualquier proyecto cuyos requisitos sean inestables y para los que se requieran rapidez y flexibilidad por partes iguales.

John Scumniotales y Jeff McKenna concibieron, ejecutaron, y documentaron el primer Scrum para ser utilizado en desarrollo ágil de software en 1993 utilizando el estudio de gestión de equipos de Takeuchi y Nonaka como base. En 1995 se normalizó el proceso gracias a que Ken Schwaber, que presentó "Scrum Development Process" en OOPSLA 95 (Object-Oriented

Programming Systems and Applications Conference) como un marco de reglas para desarrollo de software basado en los principios de Scrum y que él mismo había empleado en el desarrollo de Delphi, así como también lo hizo Jeff Sutherland en su empresa Easel Corporation. Posteriormente, en 2001 se escribiría lo que hoy en día se conoce como el 'Manifiesto Ágil'.

2.4.2. ¿Qué es Scrum?

Scrum, como hemos señalado anteriormente, no es un proceso o una técnica para construir productos o una metodología ágil como tal, sino que es un marco de desarrollo ágil que se caracteriza por ser ligero, fácil de entender, pero a su vez extremadamente difícil de llegar a dominar.

En Scrum se utilizan una serie de buenas prácticas para poder llegar a trabajar colaborativamente en equipo, apoyándose unas a otras, y que son seleccionadas a partir de un estudio sobre las mejores prácticas de equipos altamente productivos. Como marco de trabajo dentro del desarrollo ágil, en Scrum se cumplen los principios descritos como la entrega de productos parciales (ver sección 2.2.2), adopción de una estrategia incremental en lugar de una planificación completa, basar la calidad del resultado en la valoración del equipo o solapar las fases de desarrollo en vez de llevarlas a cabo de manera secuencial.

Su utilidad es diversa, resultando ser indicado para proyectos en entornos complejos donde los requisitos pueden experimentar cambios y el obtener flexibilidad en el proceso de desarrollo resulta fundamental. Además puede aplicarse para evitar situaciones como las siguientes:

- Las entregas se alargan demasiado.
- Los costes se disparan o la calidad no es aceptable.
- Falta de capacidad de reacción ante la competencia.
- La moral de los equipos es baja y la rotación alta.

O bien se necesita:

- Identificar y solucionar ineficiencias sistemáticamente.
- Trabajar utilizando un proceso especializado en el desarrollo de producto.

El marco de trabajo Scrum está compuesto de equipos con diferentes roles para sus miembros, eventos, artefactos y reglas asociadas. Todos estos elementos se relacionan a partir de las reglas Scrum.

2.4.3. Scrum en la teoría

Scrum se basa en la teoría de control de procesos empírica o empirismo, que es una variante de la teoría estadística útil en el estudio de procesos estocásticos [33]. El empirismo asegura que el conocimiento procede de la experiencia y de la necesidad de tomar decisiones basándose en lo que se conoce. Scrum emplea estas ideas, dentro de un enfoque iterativo e incremental, para conseguir optimizar la predictibilidad y el control del riesgo en el desarrollo de un proyecto de software.

Los tres principios que promueve Scrum son:

- Transparencia: todos los aspectos significativos del proceso deben ser visibles para los responsables del resultado.
- Inspección: deben inspeccionarse frecuentemente los artefactos que se producen con la aplicación de Scrum y el progreso hacia un objetivo.
- Adaptación: si los resultados obtenidos no se están acercando al objetivo, entonces se ha de revisar el proceso o el material deberá ser ajustado.

Scrum fija una serie de iteraciones de longitud fija llamadas sprints. Los hitos (consecución de un sprint), que se van alcanzando frecuentemente, proporcionan al equipo una sensación de progreso continuo y tangible. Este marco de trabajo propone cuatro protocolos para la inspección y adaptación del proceso, contenidos dentro del sprint:

- **Reunión de Planificación del Sprint :** se decide qué hay que terminar en el siguiente sprint.
- **Scrum Diario :** reuniones de 15 min para poner al día al equipo.
- **Revisión del Sprint :** reunión para demostrar lo obtenido en ese sprint.
- **Retrospectiva del Sprint :** valoración global del sprint completado con medidas para mejorar el siguiente.

2.4.4. Composición de un equipo Scrum

La composición de un equipo Scrum es ligeramente distinta al de un proyecto que siga el ciclo de desarrollo en cascada. Tenemos un dueño de producto ("product owner"), al equipo de desarrollo ("development team") y un experto en Scrum ("Scrum Master"). Entre los desarrolladores encontramos evaluadores, diseñadores e ingenieros de operaciones, que gracias a un sistema de funciones cruzadas, tienen la posibilidad de influir en la toma de decisiones a la hora de aportar soluciones por parte de cargos encargados de la dirección.

Product Owner : responsable de maximizar el valor del producto y el trabajo del equipo. Son los encargados de entender los requisitos empresariales y de mercado para, de esta forma, priorizar las distintas tareas del trabajo a desarrollar. El dueño de producto es el único responsable de la gestión de la lista del producto (o „product backlog"). La gestión del dueño de producto supone el elaborar los elementos de la lista de producto, ordenarlos, optimizar el trabajo desempeñado, asegurar que la lista de producto sea visible y transparente, y asegurarse de que el equipo de desarrollo la entiende.

Este rol lo desempeña normalmente una sola persona y el resto de personas deben respetar sus decisiones.

Development Team : los equipos de desarrollo en Scrum son profesionales encargados de desempeñar el trabajo de entregar productos de manera incremental y son expertos en prácticas de desarrollo sostenible. Los equipos de Scrum más eficaces mantienen una relación estrecha entre sus miembros, se encuentran en la misma ubicación y están compuestos por entre cinco y siete miembros.. Entre sus características más importantes tenemos:

- Autoorganización de su propio trabajo.
- Multifuncionalidad.
- Todos han de ser desarrolladores, independientemente de la tarea que desempeñen en el desarrollo del proyecto.
- No existen sub-equipos, todo es un conjunto.
- Todo recae sobre el equipo.

Scrum Master : el experto en Scrum es el responsable de asegurar que todo el equipo de desarrollo entiende el trabajo y lo hace ajustándose a la teoría, prácticas y reglas de Scrum.

Un experto en Scrum es eficaz si conoce bastante bien el trabajo realizado por su equipo y puede ayudarlo a optimizar el flujo de entrega de deliverables

durante el proceso de desarrollo. Ha de planificar el uso de los recursos necesarios (tanto humanos como logísticos) para organizar los plazos de los sprints, las reuniones rápidas, la revisión de sprints y las retrospectivas de sprints. Los expertos en Scrum también intentan resolver los obstáculos y distracciones del equipo durante el desarrollo, aislándolo de interrupciones externas siempre que sea posible.

2.4.5. Eventos de Scrum

En Scrum tenemos una serie de eventos predefinidos que permiten minimizar las reuniones destinadas a poner en común lo conseguido y que se mantienen entre los miembros del equipo. Estos eventos son bloques de tiempo con una duración máxima ya fijada.

Entre los elementos de Scrum, se puede mencionar a los sprints, que son bloques de tiempo de un mes o menos donde formalizamos el proyecto de manera incremental. Su ejecución comienza justo al terminar el sprint anterior. Cada sprint está compuesto por eventos. La falta de alguno de estos da como resultado una reducción de la transparencia y se considera una oportunidad perdida para inspeccionar y adaptarse.Es decir, podríamos considerar un sprint como un mini-proyecto de tiempo reducido y con un objetivo claro.

El protocolo que se sigue cuando seguimos un enfoque Scrum se resume en los siguientes elementos:

- **Daily Scrum :** con el fin de sincronizar sus actividades y crear un plan para las próximas 24h, cada día que dura un sprint, se realiza una reunión de 15 minutos de duración sobre el estado del proyecto. Este plan conlleva los siguientes pasos:

  1. Comenzar la reunión puntualmente.
  2. Solo hablan los integrantes del proyecto.
  3. Duración máxima de 15 minutos.
  4. Celebrar la reunión a la misma hora y en el mismo lugar.

  Entre las preguntas que pueden surgir, encontramos:

  1. ¿Que se hizo ayer para ayudar al equipo en el desarrollo del sprint?
  2. ¿Qué haremos hoy para ayudar en ese proceso?
  3. ¿Existen impedimentos para cumplir este objetivo?

- **Sprint Goal :** se trata de ofrecer al equipo de desarrollo cierta flexibilidad con respecto a la funcionalidad implementada en el sprint. Los elementos seleccionados de la'lista de producto' ofrecen una función coherente, que podría constituir el objetivo del sprint.
- **Sprint Planning Meeting :** en cada inicio de un sprint, se lleva a cabo esta reunión con el equipo completo. La reunión de planificación de sprint tiene un máximo de duración de ocho horas para un sprint de un mes. Para sprints más cortos, el evento es normalmente más breve. En este se pretende:

  1. Seleccionar el trabajo que se hará.
  2. Preparar el Sprint Backlog (o "lista de cosas pendientes").
  3. Determinar el trabajo que se realizará durante este sprint.

- **Sprint Review :** reunión que se lleva a cabo al final de cada sprint con el objetivo de realizar un análisis para inspeccionar el incremento que se ha desarrollado en este y reajustar la lista de trabajos pendientes, si es necesario. Su duración comprende una hora por cada semana durante la duración del sprint. Entre las tareas a realizar, encontramos las siguientes:

  1. Revisar el trabajo que fue completado y no completado.
  2. Presentar el trabajo completado a los interesados.
  3. El dueño de producto informa de los elementos de la lista de producto que han sido terminados y no terminados.
  4. A partir de ese sprint finalizado, se construye un plan de futuro.

- **Sprint Retrospective :** se trata de una etapa que se realiza justo después de la revisión del sprint y antes de mantener la reunión de planificación. Se lleva a cabo una retrospectiva del propio sprint, en la cual todos los miembros del equipo dejan sus impresiones sobre lo que se acaba de terminar. El propósito de la retrospectiva del sprint es llevar a cabo una mejora continua del proceso de desarrollo del producto.

2.4.6. Documentos de Scrum

- **Product Backlog :** La lista de producto es una lista ordenada de todo lo que hay que hacer para terminar el producto, y es la única fuente de la que se obtienen los requisitos a cumplir por cualquier cambio que

vaya a realizarse. Enumera todas las características, funcionalidades, requisitos, mejoras y correcciones que constituyen los cambios que han de ser hechos sobre el producto para elaborar entregas futuras.Se trata de un documento de alto nivel para todo el proyecto que nunca estará completo. Dicho documento evolucionará conforme avancemos en el desarrollo del producto, comenzando a partir de una lista de requisitos mínima.

- **Sprint Backlog :** La lista de trabajos pendientes del sprint es el conjunto de elementos de la lista de producto actualmente seleccionados para completar el sprint actual, más un plan para entregar al equipo que describe cómo se van a implementar los requisitos durante el sprint. Por lo general los requisitos se subdividen en tareas, a cada una de las cuales se le asignará unas horas de trabajo, aunque ninguna tarea puede tener una duración superior a 16 horas. Si la duración de una tarea resulta ser superior a 16 horas, deberá ser dividida en otras menores. Las tareas incluídas en el sprint backlog nunca son asignadas, sino que los miembros del equipo se responsabilizan individualmente de cada una de ellas del modo que les parezca más adecuado.

- **Burn Down Chart :** es una gráfica pública que mide la cantidad de requisitos, aún pendientes de satisfacer, en el backlog del proyecto, al comienzo de cada sprint. Su representación suele ser lineal, dibujándose una línea que conecta cada sprint que ya se haya completado, hasta llegar al punto que se considere la finalización del proyecto. Un desarrollo satisfactorio debería mostrar una línea descendente y la inclusión de nuevos requisitos durante el proceso provocaría la aparición de una sublinea de conexión entre 2 sprints con pendiente ascendente.

2.5. ¿Qué es la arquitectura de software?

La arquitectura de un producto software podría entenderse con la analogía de la formación de un equipo de fútbol en el terreno de juego. A través de una serie de pasos, según la arquitectura elegida, definimos la forma de trabajar que poseen los componentes de un sistema.

Existen numerosas definiciones de lo que es Arquitectura de Software, que tratan de ser la estándar, pero la más conocida formalmente es la expresada en el libro 'Software Architecture in Practice, Third Edition' [22]:

"La Arquitectura de Software de sistema de computación es un conjunto de estructuras de software, que comprenden: unidades de implementación,

aspectos dinámicos y la correspondencia con entornos de ejecución, instalación, desarrollo y organización del sistema."

Otras definiciones nos hablan de tomas de decisiones tempranas por parte del equipo de desarrollo del proyecto, pero específicamente referidas a componentes estáticos de la arquitectura, lo cuál no es precisamente de aplicación en nuestro caso, sobre todo porque en el contexto de aplicación de técnicas ágiles todas las decisiones son dinámicas.

Partimos de los requisitos del cliente, los cuales tendremos que analizar y gestionar para producir un diseño inicial. Este diseño debe ser una representación fiel de la arquitectura que usemos en nuestro software. Por ejemplo, si observamos nuestro dibujo arquitectónico de software, deberíamos de poder intuir qué tipo de aplicación va a ser construida. No es lo mismo una aplicación que controla un hospital que una aplicación de un cajero automático, ya que cada una tendría un dibujo arquitectónico distinto. Nuestro diseño, además de cumplir con los requisitos del cliente, debe manejar una serie de disyuntivas debido a conflictos razonables dentro del desarrollo del proyecto, que son consecuencia de la aplicación de técnicas, principios de diseño o del uso de un hardware recomendable para cada caso. Es necesario asimilar todo lo aportado por el cliente para conseguir producir en un diseño robusto y práctico.

2.5.1. ¿Por qué es tan importante en el desarrollo?

La complejidad que ha aportado los avances tecnológicos y de software son relevantes, especialmente, en el desarrollo de software. Los clientes solicitan una serie de tareas cuya aplicación se debe poder manejar y la gran mayoría de estas conllevan interconectividad y suelen estar orientadas a su implementación en entornos específicos, como la nube.

Eoin Woods es un arquitecto de software que trabaja desde principios de los 90 en una gran variedad de campos del desarrollo de software y expuso la siguiente definición de lo que él entendía como AS:

"La arquitectura de software es el conjunto de decisiones de diseño que, si se realizan incorrectamente, pueden provocar la cancelación de su proyecto."

Una elección incorrecta de una arquitectura podría llevarnos, como menciona Eoin, a un trágico final. Sin embargo, una toma de decisiones adaptada eficazmente sobre un proyecto es el camino correcto hacia un resultado óptimo y que cumpla los requisitos. En el libro mencionado anteriormente se examinan una serie de razones que explican porqué es tan relevante el uso de una AS:

1. Según qué arquitectura, se propiciarán o inhibirán unos u otros atributos de calidad.

2. Hace posible la predicción temprana de los atributos de calidad mediante análisis de la arquitectura.

3. Un análisis previo de la arquitectura seleccionada ofrece las cualidades esperables del sistema objetivo.

4. Constituye la mejor documentación que propicia la comunicación entre las partes interesadas en el proyecto.

5. La arquitectura supone la base de las decisiones más importantes durante el proceso de desarrollo porque son las más difíciles de modificar posteriormente, debido a su pronta aplicación. Por consiguiente, esta es una de las características que se desprenden de la definición de la AS que interfiere de una manera importante con las características del desarrollo ágil.

6. La arquitectura define un conjunto de restricciones de implementación del software en etapas posteriores del desarrollo del proyecto.

7. Puede servir para determinar la estructura de una organización o viceversa, dicha estructura podría condicionar la elección de la AS.

8. Se utiliza como una base fundamental del denominado „prototipado rápido“ del software.

9. Definir una arquitectura u otra permite al responsable razonar sobre el coste y la planificación del proyecto como un todo.

10. Se puede entender como un modelo reutilizable y transferible conforme con el núcleo de una línea de producto.

11. Se centra más en el ensamblaje de componentes que en su creación individualizada.

12. Al centrarse en un único diseño, se canaliza la creatividad de los desarrolladores para poder reducir la complejidad del diseño.

13. Se considera una base para entrenar a nuevos miembros del equipo de desarrollo.

2.5.2. Elementos y estructuras de una arquitectura

Las arquitecturas, con el fin de facilitar su comprensión, se componen en estructuras y vistas, facilitando en cada caso la estructura a utilizar y la vista a adoptar.

- Una vista representa un conjunto de elementos arquitectónicos, compuestos por los desarrolladores del sistema, y que están relacionados entre sí.
- Una estructura representa un conjunto de elementos, ya sea de software o hardware.

En general, entenderemos "vista" como la representación de un conjunto coherente de elementos arquitectónicos y las relaciones que existen entre ellos. Una vista podría ser la representación de una estructura del sistema objetivo, tal como por ejemplo:

1. Una estructura de módulo es el conjunto de los módulos del sistema y su organización.
2. Una vista de módulo es la representación de esa estructura, documentada según una plantilla en una notación elegida, y utilizada por algunas partes interesadas del sistema.

2.5.2.1. Estructuras arquitectónicas

1. **Estructuras modulares :** se define la estructura modular del sistema objetivo como un conjunto de unidades de código que, normalmente, se compilan independientemente y que contienen todo lo necesario para ejecutar sólo un aspecto de la funcionalidad del sistema. Las preguntas a responder con esta estructura son la responsabilidad funcional primaria de cada módulo, qué otros elementos usará ese modulo, si usa otro software (y sus dependencias) y cuál es esta relación entre módulos.
2. **Estructuras componente-conector (CandC) :** define como se debe estructurar el sistema como un conjunto de elementos, que incluyen los componentes y los conectores. Los componentes son los servicios, clientes, servidores, etc. y los conectores representan a los mecanismos de comunicación entre componentes. Estas estructuras responden a las siguientes preguntas: cuáles son los principales componentes y como

interaccionan, qué datos compartidos existen, cuáles son las partes del sistema a replicar, qué progreso siguen los datos a través de cada componente, qué partes van a funcionar en paralelo y si es posible cambiar la estructura del sistema y cómo hacerlo.

3. **Estructuras de despliegue o asignación :** incorporan decisiones sobre cómo el sistema se relacionará con estructuras en su entorno que no son de software exactamente (como CPU, sistemas de archivos o redes). Respondemos en este caso a preguntas como: qué procesador va a ejecutar cada elemento de software, cómo se organiza en directorios durante el desarrollo y la asignación de cada elemento a los equipos.

2.5.2.2. Vistas

1. **Visión estática :** describe los componentes de la arquitectura.
2. **Visión funciona :** describe el funcionamiento de cada uno.
3. **Visión dinámica :** describe el comportamiento de ellos a lo largo del tiempo y el tipo de interacción entre sí.

Su representación puede expresarse tanto en lenguaje natural como mediante el uso de lenguajes creados específicamente para ello, tales como son los diagramas de estado, diagramas de flujo de datos, etc. El estándar actual es UML.

2.6. Ciclo de desarrollo en un arquitectura

Independientemente de la metodología utilizada (ágil, por ejemplo), proyecto adjudicado o grupo de trabajo asignado, existen una serie de pautas que preceden a la construcción del sistema, divididos en etapas.

El código que implementa aspectos relacionados con atributos de calidad generalmente se encuentra disperso en una gran cantidad de módulos del sistema. Toma esta toma de decisiones se llevan a cabo de forma temprana. Es así que realizar cambios en atributos de calidad de forma tardía es una cuestión compleja y problemática.

Por ello, este desarrollo precede a la construcción del sistema y está dividido en las siguientes etapas: requerimientos, diseño, documentación y evaluación.

1. **Definir los requerimientos :** La primera etapa se encarga de crear un modelo de requerimientos, los cuales guiarán posteriormente el diseño de la arquitectura. Para ello, debemos capturar/identificar, documentar y especificar los requerimientos. Entre estos tenemos los atributos de calidad, que juegan un papel importante y otros como los funcionales primarios y las restricciones del sistema.

2. **Diseño de la Arquitectura :** La etapa de diseño abarca el desarrollo central de la misma y el proceso más complejo. En ella, definimos las estructuras que compondrán la arquitectura. La creación de estas estructuras se hace en base a patrones de diseño, tácticas de diseño y elecciones tecnológicas. El diseño debe buscar por encima de todo cumplir con los requisitos que influencian a la arquitectura y se deben asignar las responsabilidades de lo componentes que comprenderán la arquitectura del software en cuestión.

3. **Documentación del diseño :** En el momento en el que el diseño de la arquitectura ha 'finalizado', debemos documentar los aspectos fundamentales, capturando las decisiones tomadas con el fin de poder comunicarlo a otros involucrados dentro del proceso de desarrollo del software. Esta documentación involucra representar las estructuras utilizadas a partir de vistas, las cuales están compuestas por diagramas e información adicional.

4. **Validación :** Una vez definido el diseño de la arquitectura, es conveniente evaluarlo a partir de la documentación realizada en la etapa anterior, obteniendo posibles errores o riesgos que produzcan problemas a largo plazo. La ventaja de analizar el diseño construído en una etapa temprana de desarrollo es sin duda el costo de corrección de los defectos identificados, una gran ventaja.

Aunque hayamos enumerado las etapas, cada una de ellas pueden necesitar retroceso. Es decir, es posible que, una vez alcanzada y finalizada la validación para la corrección de errores, se necesite volver a una etapa anterior. Un esquema que represente este proceso seria el siguiente:

Figura 2.2: Representación del ciclo de desarrollo de una arquitectura.

2.7. Arquitecturas más comunes

Ante el reto de elegir la arquitectura correcta para nuestro proyecto, es necesario conocer los diseños ya establecidos, los cuales son ampliamente usados en la industria. Entre ellos, lo más conocidos son:

2.7.1. Arquitectura cliente-servidor

La arquitectura cliente-servidor es un tipo de modelo de diseño de software que consiste en uno o varios usuarios realizando peticiones a un programa. En este diseño, los usuarios son denominados como demandantes (clientes) y el programa como proveedor de recursos o servicios, conocido como servidor. El cliente realiza una petición y el servidor le da respuesta. Aunque esta idea se puede aplicar a programas que se ejecutan sobre un solo ordenador es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de ordenadores.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores, aunque son más importantes las ventajas de tipo organizativo debidas a la centralización de la gestión de la información y

la separación de responsabilidades, lo que facilita y clarifica el diseño del sistema.

Entre las características que posee este tipo de arquitectura, podemos encontrar:

- Las tareas del cliente y servidor tienen diferentes requerimientos en cuanto a recursos de cómputo.

- La arquitectura es una relación entre procesos ejecutados en ordenadores distintos.

- Un servidor atiende a muchos clientes al mismo tiempo, siendo una relación que puede ser de muchos a uno.

- El cliente es activo al solicitar peticiones y el servidor pasivo al esperar estas.

- Una de las principales ventajas de esta arquitectura es la posibilidad de conectar clientes y servidores independientemente de sus plataformas.

- El concepto de escalabilidad tanto horizontal como vertical es aplicable a cualquier sistema Cliente-Servidor. La escalabilidad horizontal permite agregar más estaciones de trabajo activas sin afectar significativamente el rendimiento. La escalabilidad vertical permite mejorar las características del servidor o agregar múltiples servidores.

- El servidor es un proceso que puede residir en el mismo equipo que el proceso cliente o en un equipo distinto a lo largo de una red.

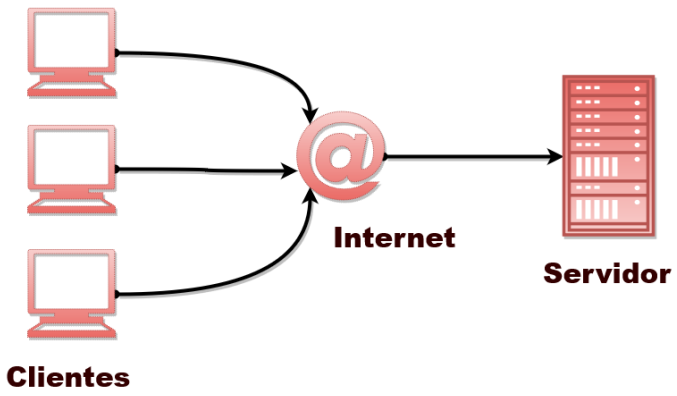

Figura 2.3: Esquema de la arquitectura Cliente-Servidor.

2.7.2. Programación por capas

La programación por capas es un modelo de desarrollo de software en el que el objetivo primario es el desacoplamiento lógico de las partes que componen un sistema software: capa de presentación y capa de negocios. Esto permite crear distintas interfaces sobre una misma base de datos. La ventaja principal de este estilo es que el desarrollo se puede llevar a cabo en varios niveles y, en caso de que sea necesario algún cambio, solo afectará al nivel requerido sin tener que revisar entre el código fuente de otros módulos. Además, permite distribuir el trabajo de creación de una aplicación por niveles

El término çapa"hace referencia a la forma como una solución es segmentada desde el punto de vista lógico. Tenemos 3 tipos:

- **Capa de presentación :** Llamada también como capa de usuario, presenta el sistema al usuario, comunicándole información y capturando sus datos mediante un filtrado de entrada. Es comúnmente conocida como interfaz gráfica, y como tal, debe ser amigable y entendible. Esta capa se comunica exclusivamente con la de negocio.

- **Capa de negocio :** La capa de negocio incluye una serie de programa que se ejecutan, reciben peticiones del usuario y envían sus respuestas

tras el proceso. Aquí se establecen las reglas a cumplir. Su comunicación es abierta tanto a la capa de presentación para recibir solicitudes como a la capa de datos para almacenar o recuperar datos.

- **Capa de datos :** Capa donde los datos residen y se consultan. Se compone de uno o varios gestores de bases de datos, los cuales reciben, almacenan y recuperar solicitudes desde la capa de negocio.

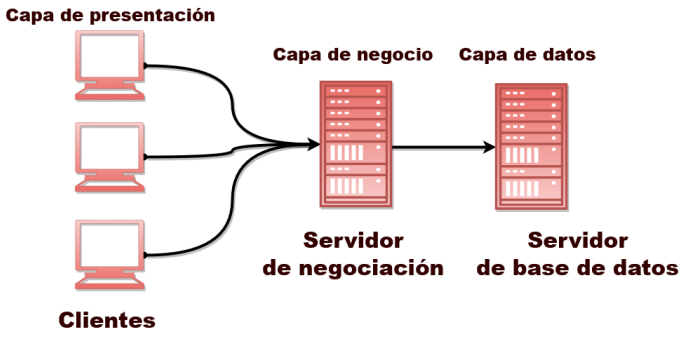

Figura 2.4: Esquema de la arquitectura por capas.

2.7.3. Modelo vista-controlador (MVC)

El modelo Vista-Controlador es un patrón de arquitectura de software que separa datos y lógica de negocio de una aplicación de su representación y el módulo encargado de los eventos y comunicaciones. Es decir, están explícitamente separados y anejados por tres tipos de objetos, cada uno especializado para un conjunto de tareas.

El objetivo primordial del patrón es dar soporte a los modelos funcionales y mapas mentales de la información relevante para los usuarios, permitiendo un modelo que facilite la consulta y manejo de los mismos.

- **Capa de Modelo :** Esta capa implementa la lógica de negocio, formado por un conjunto de clases que representan la información del mundo real. Así, por ejemplo, un sistema de administración de datos geográficos tendrá un modelo que representara la altura, coordenadas de posición, distancia, etc. sin tomar en cuenta ni la forma en la que esa información va a ser mostrada ni los mecanismos que hacen que

esos datos estén dentro del modelo, es decir, sin tener relación con ninguna otra entidad dentro de la aplicación.

- **Capa de la Vista :** La capa de vista representa los datos del modelo al usuario. Es responsable del uso de la información de la cual dispone para producir cualquier interfaz de presentación de cualquier petición que se presente. Se puede tener varias vistas asociadas a un modelo. De esta manera, por ejemplo, se puede tener una vista mostrando la hora del sistema como un reloj analógico y otra vista mostrando la misma información como un reloj digital.

- **Capa del Controlador :** La capa del controlador se encarga de interpretar las peticiones de los usuarios. Responde a las peticiones de los usuarios a partir del modelo y la vista. Interactúa con el modelo a través de una referencia al propio modelo.

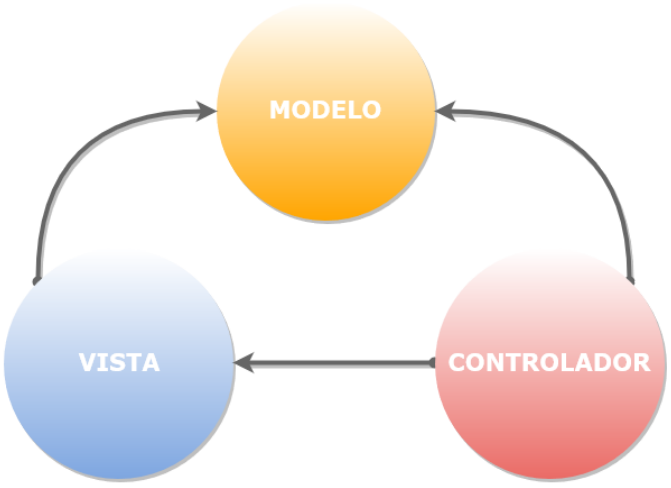

Figura 2.5: Esquema de MVC.

2.7.4. Arquitectura orientada a servicios (SOA)

Una arquitectura orientada a servicios es una metodología estándar del sector cuya definición es abierta, orientado el marco de trabajo desde un punto de vista a los servicios. Ha de contener la lógica de negocio y los datos de todos los sistemas informáticos de las empresas. Existen varias definiciones como:

- **IBM :** "Un modelo de componentes que interrelaciona las diferentes unidades funcionales de las aplicaciones, denominadas servicios, a través de interfaces y contratos bien definidos entre esos servicios. La interfaz se define de forma neutral, y debería ser independiente de la plataforma hardware, del sistema operativo y de los lenguajes de programación utilizados. Esto permite a los servicios, construidos sobre sistemas heterogéneos, interactuar entre ellos de una manera uniforme y universal."

- **Microsoft :**"La Arquitectura SOA establece un marco de diseño para la integración de aplicaciones independientes de manera que desde la red pueda accederse a sus funcionalidades, las cuales se ofrecen como servicios."

Abordan problemas de diseño hacia sistemas de software más complejos, separando la lógica de integración de negocio de la implementación, para que el desarrollador de integración pueda centrarse en ensamblar una aplicación integrada en lugar de hacerlo en los detalles de la implementación.

Cumplen la siguiente serie de principios:

1. Bajo acoplamiento.

2. Contrato de servicio.

3. Autonomía.

4. Abstracción.

5. Reusabilidad.

6. Composicionalidad.

7. Ausencia de estado.

8. Facilidad de descubrimiento.

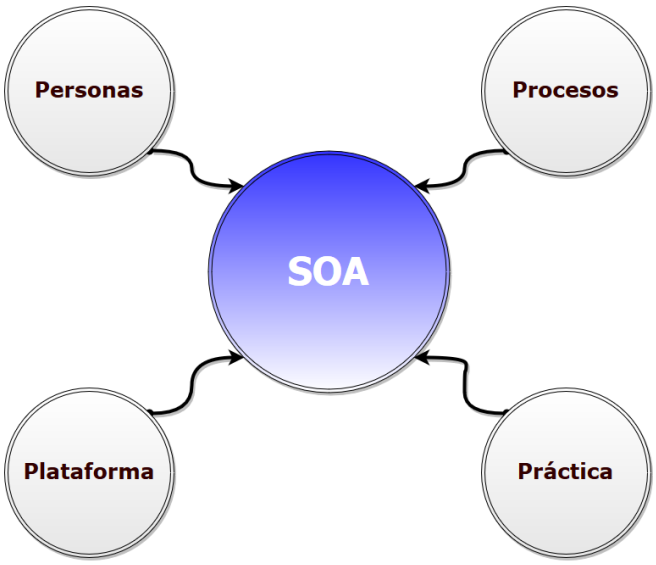

Figura 2.6: Esquema de SOA.

2.7.5. Arquitectura dirigida por eventos (EDA)

La Arquitectura dirigida por eventos, Event-driven architecture o EDA, es un patrón de arquitectura software que fomenta la producción, detección, consumo de, y reacción a eventos en sistemas y aplicaciones que transmiten eventos entre componentes software. El concepto de "evento" se refiere a un cambio de estado en un sistema que provoca posteriormente la transmisión de un mensaje o notificación del cambio. Por otro lado, el término evento es frecuentemente usado también para denotar el mensaje de notificación en sí mismo, lo cual puede llevar a algún tipo de confusión.

Un sistema dirigido por eventos está compuesto típicamente de emisores de eventos (o agentes), consumidores de eventos (o "sink" en inglés), canal de eventos, motor de procesamiento de eventos y reglas de negocio y actividades derivadas.

Construir aplicaciones y sistemas alrededor de una arquitectura dirigida por eventos permite a estas aplicaciones y sistemas ser construidos de una manera que facilita un mayor grado de reacción, debido a que los sistemas dirigidos por eventos están, por el diseño, más normalizados para entornos no predecibles y asíncronos.

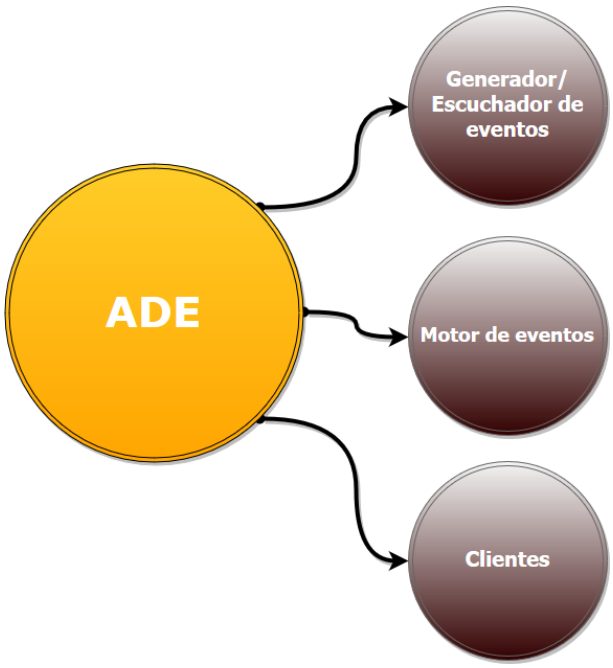

Figura 2.7: Esquema de ADE.

2.8. ¿Es necesario el diseño trabajando con metodologías ágiles?

Si bien en el desarrollo de software existen numerosos métodos, desde los más tradicionales como el modelo de cascada o de espiral con un enfoque predictivo, hasta las impuestas a día de hoy como metodologías ágiles, existen corrientes de pensamiento que inciden en la inclusión de las arquitecturas de software dentro del enfoque metodológico ágil del desarrollo de software.

Las AS favorecen, por encima de todo, los requisitos no funcionales, ligados a los atributos de calidad como la seguridad, el rendimiento o la escalabilidad. Sus componentes más importantes son los patrones arquitectónicos y de diseño, constituyendo un conjunto de principios que proporcionan un marco para el diseño, proveen el diseño de la aplicación, mejoran la partición y ayudan a definir las características básicas y de comportamiento del sistema objetivo. Cada uno de estos aspectos se desarrollará en apartados posteriores.

¿Por qué la pregunta en el título acerca de si la introducción del diseño arquitectónico en metodologías ágiles favorecen el producto final? Desde un

punto de vista ágil, no es precisamente necesario el diseño. En la idea de la agilidad de software, sus principios no permiten que existan decisiones inamovibles y prematuras en el largo plazo, ya que el desarrollo debería ser evolutivo y no condicionado por un primer diseño fijo.

En los documentos 'Code as Desing: Three Essays' [20] (compuesto por 3 ensayos publicado en diferentes fechas) y 'Less is More with Minimalist Architecture' [35], escritos por Jack W. Reeves, Ruth Malan y Dana Bredemeyer respectivamente, apostaban por el uso del código como fuente de diseño. En el segundo libro citado podemos encontrar la idea resumida:

"La arquitectura y la gobernanza aportan un elemento de control que puede perjudicar a las personas llevándolas hacia el camino equivocado. Las decisiones arquitectónicas en caso de fallo deben enfocarse ahí donde el precio a pagar es más alto para maximizar la probabilidad de éxito. El enfoque minimalista de la arquitectura implica clasificar los requisitos arquitectónicos de mayor prioridad y luego hacer lo mínimo posible para lograrlos. Es decir, se ha de mantener la decisión de contar con la arquitectura lo más pequeña posible, mientras se asegura que el personal técnico cumple con las prioridades clave del sistema."

La perspectiva de algunos desarrolladores es que el diseño no debe existir como una etapa de desarrollo más, ni influir formalmente en otras fases o elementos relevantes como tal. Su existencia es meramente informal, aportando ideas durante el progreso del software. El engorro de completar y mantener una documentación es una de las desventajas de no utilizarlo.

Por contra, a partir de este proyecto queremos probar justo lo contrario: priorizar el producto no significa exclusivamente prescindir de un diseño y una documentación. Si buscamos la definición de "abstracción" podemos encontrar la siguiente:

"La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?". "

Esto es precisamente lo que un diseño previo junto a una documentación acorde con éste nos puede aportar. El código nos ofrece información referente a la estructura interna de la aplicación desarrollada, pero no de los elementos que lo componen. La abstracción es el mecanismo para ello: no significa mantener un alto formalismo en la especificación, no significa mantener grandes y engorrosos documentos, ni planificar y prever todo el diseño en forma previa, sino que podamos simplificar la documentación sobre la arquitectura del sistema, es decir, hace posible no tener que prescindir de hacer arquitectura y diseño del sistema software que pretendemos desarrollar.

Por tanto, tener la documentación mínima y necesaria para entender la arquitectura del sistema de forma ágil significa tener en cuenta los ajustes

arquitectónicos durante todo el desarrollo para posibilitar la obtención de una buena arquitectura y esto significa hacer buena ingeniería.

Capítulo 3

Patrones arquitectónicos y atributos de calidad

Para la elaboración del proyecto, debemos profundizar en el uso de patrones y sus atributos asociados:

3.1. ¿Qué son los patrones?

Los patrones se ofrecen como una forma de solucionar problemas con una temática ya discutida anteriormente. Son modelos que ofrecen soluciones a estos problemas, como un conjunto de clases. Si la metodología utilizada para un proyecto es posible extraerla, explicarla y reutilizarla en el futuro, nos encontramos con un patrón de diseño de software. Es decir, las características principales de un patrón son: reutilizabilidad y efectividad.

Existen varias definiciones sobre qué es un patrón, entre ellas tenemos:

- *" Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software. "*
- *" Los patrones de diseño son unas técnicas para resolver problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. "*

Este concepto de patrón no es ninguna novedad, existiendo desde finales de los años 70, aunque su verdadera popularización no fue hasta los años 90 con el libro 'Design Patterns' [34] escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Desarrollan 23 patrones de diseño, convirtiéndose en todo un referente para los desarrolladores de software desde su publicación.

La utilidad de los patrones la encontraremos durante el desarrollo de nuestro proyecto, ya que su utilización nos proporcionará un ahorro de tiempo en el proceso de búsqueda de una solución si existe ya una conocida y con la capacidad de adaptarse a nuestro problema. También a la hora de validar el código fuente y de establecer un lenguaje común entendible para todo el grupo de trabajo que conozca el patrón utilizado.

Existen varias clasificaciones de los patrones atendiendo a según qué característica. Tenemos en primer lugar una clasificación basada en los requisitos de seguridad, rendimiento, despliegue y almacenamiento:

1. **Control de accesos :** el acceso a determinadas partes de la arquitectura software ha de ser controlado rigurosamente.
2. **Concurrencia :** diferentes formas de permitir que los componentes de la aplicación sean concurrentes.
3. **Distribución :** la comunicación entre entidades software es muy diversa y afecta al diseño, la ubicación de los componentes ha de ser optimizable (configurabilidad).
4. **Persistencia :** la supervivencia de los objetos (atributos, estado) entre distintas ejecuciones es algo diseñable. Una mala solución puede dañar la eficiencia del software gravemente.

Otra clasificación se basa en el nivel de abstracción o escalabilidad, obteniendo:

1. **Patrones de arquitectura :** aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.
2. **Patrones de diseño :** aquellos que expresan diferentes esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.
3. **Dialectos :** patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

Y por último, también existe una clasificación de patrones una basada en su naturaleza:

1. **Creacionales :** instanciación de objetos reforzando las restricciones en el tipo y número de estos.
2. **Estructurales :** atienden a la organización de un software entendida como integración de clases de objetos.

3. **Comportamentales :** asignación de responsabilidades y comunicación entre los objetos.

Para describir los patrones creados existen una serie de plantillas o estructuras de forma que se expresen uniformemente y puedan constituir efectivamente un medio de comunicación uniforme entre diseñadores. La plantilla más estándar es la creada por los autores mencionados y consta de los siguientes apartados:

Nombre del patrón: nombre estándar del patrón.

Clasificación del patrón: creacionales, estructurales o de comportamentales.

Intención: ¿Qué problema pretende resolver el patrón?

También conocido como: Otros nombres de uso común para el patrón.

Motivación: Escenario de ejemplo para la aplicación del patrón.

Aplicabilidad: Usos comunes y criterios de aplicabilidad del patrón.

Estructura: Diagramas de clases oportunos para describir las clases que intervienen en el patrón.

Participantes: Enumeración y descripción de las entidades abstractas (y sus roles) que participan en el patrón.

Colaboraciones: Explicación de las interrelaciones que se dan entre los participantes.

Consecuencias: Consecuencias positivas y negativas en el diseño derivadas de la aplicación del patrón.

Implementación: Técnicas o comentarios oportunos de cara a la implementación del patrón.

Código de ejemplo: Código fuente ejemplo de implementación del patrón.

Usos conocidos: Ejemplos de sistemas reales que usan el patrón.

Patrones relacionados: Referencias cruzadas con otros patrones.

3.2. ¿Cuáles son los principios de un diseño basado en patrones?

1. Descubrir el contexto del problema a partir de un modelo de requerimientos previamente elaborado.
2. Extraer los patrones del nivel actual de abstracción (suponemos un diseño iterativo): datos, diseño, arquitectónico, etc.
3. Escoger un esqueleto de implementación que refleje el contexto.
4. Trabajar siempre hacia dentro del contexto (modelar el contexto del sistema no es nuestra responsabilidad).
5. Refinar el prototipo que obtengamos adaptándolo a las características de la plataforma software donde desplegaremos el sistema objetivo.

3.3. ¿Qué son los patrones arquitectónicos?

Los patrones arquitectónicos tienen varias definiciones según varios autores. La definición más esquemática es que son plantillas para arquitecturas de software muy concretas, que especifican la estructura de una aplicación junto a sus propiedades y tienen un impacto en la arquitectura de subsistemas. Podríamos agruparlos en conjuntos de subsistemas con responsabilidades y relaciones entre sí.

El uso de ciertos mecanismos como los estilos que describen una clase general de arquitectura software o partes de la misma y, con más importancia, los patrones arquitectónicos, permite mejorar las características de calidad del software producido, bien sean estas observables o no en tiempo de ejecución del sistema que se pretende desarrollar.

3.4. Características de los patrones arquitectónicos

- La selección de un patrón arquitectónico es el primer paso a la hora de abordar el diseño de la arquitectura del software.
- Su elección debe estar basada, fundamentalmente, en los requisitos no funcionales de la aplicación (o del sistema) objetivo y en las propiedades que ha de poseer.

- Es posible que un proyecto de desarrollo de software necesite de más de un patrón arquitectónico para su estructuración, por lo que podríamos llegar a combinar varios patrones durante la etapa de diseño.
- Nos asegura llegar a una selección correcta la prueba de otros patrones con anterioridad.
- Los patrones imponen ciertas reglas a la arquitectura del sistema, además de abordar problemas de comportamiento del software a determinados sucesos.
- Un patrón arquitectónico abarca aspectos específicos en una arquitectura. Es menor su alcance comparado con un estilo arquitectónico que engloba componentes, relaciones, restricciones, etc.
- Tanto patrones como estilos arquitectónicos se pueden utilizar conjuntamente para dar forma a la estructura completa.

3.5. ¿Qué son los atributos de calidad?

Los atributos de calidad han sido objeto de mucho interés para la industria del software desde hace años, precisamente desde la década de los 70. Existen numerosas definiciones publicadas, cada una cuenta con comunidades de profesionales que la propone como más representativa del concepto para su aplicación al software.

Siguiendo el libro 'Software Architecture in Practice' [22] podemos encontrar las definiciones, clasificaciones y demás datos interesantes y popularizados por el mundo del software sobre los atributos de calidad.

Un atributo de calidad es definido como una propiedad de un sistema que posee la capacidad de ser medida o comprobable, este concepto es utilizado para indicar cómo de bien se cumplen las necesidades de las partes interesadas (o "stakeholders") en el proyecto. Su utilización nos posibilita tener una o varias características en nuestro proyecto. Sin embargo, cumplir con un atributo de calidad producirá un impacto en los restantes atributos, tanto negativa como positivamente.

Para especificar estos atributos de forma común (con el riesgo de no ser totalmente fieles a algunos de ellos), se tienen en cuenta seis tipos de escenarios (pudiéndose omitir algunos de estos):

1. **Fuente del estímulo :** entidad donde se genera el evento.

2. **Estímulo :** se entiende como un evento que requiere de una respuesta del sistema.

3. **Medio :** entorno sobre el que opera el estímulo bajo ciertas condiciones.

4. **Artefacto :** puede tratarse de una colección de sistemas, todo el sistema o alguna pieza del mismo.

5. **Respuesta :** actividad realizada como respuesta al estímulo.

6. **Medida de respuesta :** medición de la respuesta del estímulo, que resulta imprescindible para su validación.

La clasificación más formal existente de los atributos de calidad del software tiene en cuenta los siguientes atributos principales: disponibilidad, interoperabilidad, modificabilidad, rendimiento, seguridad, comprobabilidad y usabilidad. Todas estas características se pueden cubrir mediante el uso de patrones de diseño arquitectónico y están recogidos en el ISO 9126, un estándar internacional para la evaluación de la calidad del software.

3.6. Tipos de atributos de calidad

En el diseño arquitectónico, el uso de patrones de diseño nos van a permitir que nuestro sistema cumpla con los criterios de calidad que necesitemos. Algunos de otros patrones son los siguientes:

3.6.1. Disponibilidad

Proporciona la seguridad de que el sistema estará disponible para su uso legítimo, listo para llevar a cabo su tarea cuando se necesite. Puede asociarse al concepto de confiabilidad (capacidad de evitar fallos que son más frecuentes y más graves de lo aceptable), obteniendo una definición más completa:

"La disponibilidad se refiere a la capacidad de un sistema para enmascarar o reparar fallos de modo que el período acumulado de interrupción del servicio no exceda un valor requerido durante un intervalo de tiempo específico".

3.6.2. Interoperabilidad

La interoperabilidad implica la existencia de al menos dos sistemas para llevar a cabo tareas e intercambio de información a través de interfaces en un determinado contexto. No solo abarca el operar conjuntamente, además la capacidad de poder interpretar correctamente los datos que se intercambian (interoperabilidad semántica).

3.6.3. Modificabilidad

La modificabilidad ofrece la capacidad de permitir cambios en el sistema cuando sean necesarios, ya sea por un cambio de requisitos o por un error no detectado en etapas anteriores y es necesaria su corrección. Se centra también en el costo y riesgo de realizar estos cambios.

Para ello, el ingeniero debe considerar varias cuestiones:

1. ¿Qué consecuencias pueden producirse tras el cambio?
2. ¿Qué cambios deben llevarse a cabo y cuáles no?
3. ¿Quién y cuándo se hará?
4. ¿Qué precio conlleva el cambio?

Aportar modificabilidad a un desarrollo de software ágil puede ofrecer numerosas ventajas.

3.6.4. Rendimiento

El rendimiento aporta una medida de eficiencia en el uso de los recursos del sistema. El sistema debe ser capaz de responder a cualquier evento que se produzca como interrupciones, mensajes o solicitudes de usuarios u otros sistemas, dentro de los requisitos de tiempo establecidos.

3.6.5. Seguridad

Un atributo con la capacidad para proteger la información del sistema a accesos no autorizados y a los que sí lo están. Estos accesos no autorizados se denominan ataques, pudiendo llegar a adoptar la forma de un intento de acceso no permitido al sistema o negar servicios a usuarios legítimos. Alguno de los beneficios que nos proporciona un sistema con seguridad son:

- Comprobar la identidad de las personas que intentan acceder al sistema.
- Garantizar que sólo las personas específicamente autorizadas pueden ver y modificar sus zonas de acceso.

3.6.6. Comprobabilidad

La comprobabilidad ofrece la facilidad con la que un sistema puede mostrar los errores a través de pruebas en diferentes apartados. Es un mecanismo esencial para conseguir sistemas con un buen diseño a prueba de fallos. Intuitivamente, un sistema es comprobable si arregla sus errores con relativa facilidad. Si hay un fallo en un sistema, entonces queremos que falle durante la prueba específica para detectarlo lo más rápido posible.

3.6.7. Usabilidad

La capacidad de un sistema para poder utilizarse fácilmente y aprender de él, se denomina usabilidad. Es uno de los atributos que se ha venido utilizando más intensamente para mejorar la calidad del software pensando en el usuario. Este concepto abarca los siguientes aspectos:

- Facilitar el sistema de aprendizaje.
- Usar un sistema de manera eficiente.
- Minimizar el impacto de los errores.
- Adaptar el sistema a las necesidades del usuario.
- Incrementar la confianza y la satisfacción.

3.6.8. Extensibilidad

La facilidad de adaptar el producto software a los cambios de especificación durante el proceso de desarrollo se suele denominar con el término 'extensible'. La extensibilidad de un sistema de computación es la característica que determina si el sistema puede ser extendido y re-implementado en diversos aspectos.

3.6.9. Acoplamiento

Determina el grado y la forma de independencia que poseen los módulos de software (o funciones, subrutinas, etc.) en un sistema informático. Da una idea de lo dependientes que son las unidades, el grado de que una unidad necesite de otra para funcionar.

3.6.10. Encapsulamiento

El encapsulamiento es un mecanismo que consiste en organizar datos y métodos de una estructura y protegerlos dentro de algún componente o elemento modular, conciliando el modo en que el objeto se implementa con su uso por parte del resto del sistema, es decir, evitando el acceso a datos por cualquier otro medio distinto a los especificados para ese componente. Proporciona ocultamiento.

3.6.11. Reusabilidad

Explota las similitudes entre elementos para evitar la reinvención de soluciones a problemas ya arreglados. A partir de la reusabilidad, un elemento software se podrá aplicar en otros desarrollos diferentes.

3.6.12. Complejidad

El atributo referido a la complejidad es referido al proyecto software que tienen múltiples interconexiones, unido a una estructura altamente complicada. Aumenta la dificultad al grupo de trabajo para llevar el desarrollo sin problemas.

3.6.13. Escalabilidad

Similar al atributo de modificabilidad, pero la escalabilidad se refiere a la posibilidad del software de modificar sus prestaciones conforme se amplía el alcance del producto. En general, también se podría definir como la capacidad del sistema informático de cambiar su tamaño o configuración para adaptarse a las circunstancias cambiantes

3.6.14. Transparencia

La transparencia como atributo de calidad permite que el usuario no perciba los procesos internos que se llevan a cabo con cada operación y es como si no existieran.

3.6.15. Optimización

El atributo de optimización ofrece que el patrón que usemos para el diseño que favorezca esta característica, permita que el sistema funcione de manera más eficiente y a utilizar menos recursos, obteniendo un mayor rendimiento.

3.6.16. Correctitud

La correctitud ofrece la posibilidad de conocer si el sistema satisface las propiedades definida, es decir, se puede demostrar correcto de la forma más conveniente para sus usuarios. Es decir, aporta al sistema mecanismos para encontrar los errores y solucionarlos.

3.6.17. Adaptabilidad

La adaptabilidad ofrece la facilidad del producto software para ser adaptado a diferentes entornos especificados sin afectar a otros componentes del mismo.

3.6.18. Fallos

Puede ocasionar pueden causar daños en los datos y alteraciones en el comportamiento del sistema.

3.6.19. Cohesión

El atributo de cohesión se refiere al grado en que los elementos de un módulo permanecen juntos. Por lo tanto, la cohesión mide la fuerza de la relación entre las piezas de funcionalidad dentro de un módulo dado.

3.7. Tipos de patrones y atributos asociados

A continuación se describen algunos de los patrones arquitectónicos más comúnmente utilizados para cubrir escenarios de modificabilidad e integrabilidad:

3.7.1. Patrón Interceptor

El patrón arquitectónico Interceptor permite incluir diferentes servicios o procesos de forma transparente para que sean activados en el momento en el que ocurra un determinado evento. Se basa en la idea de incluir dinámicamente servicios por parte de las aplicaciones. Por lo tanto, el patrón mejorará el mantenimiento mediante la promoción de la facilidad de los cambios / inclusión de nuevos servicios y los objetos del administrador de llamadas en la estructura del interceptor.

Posee algunos problemas como que los marcos de trabajos, arquitecturas de software, etc. deberían poder anticiparse a las demandas de servicios concretos y que esta integración no afecte a otros componentes. Una posible solución es llevar un registro offline de servicios

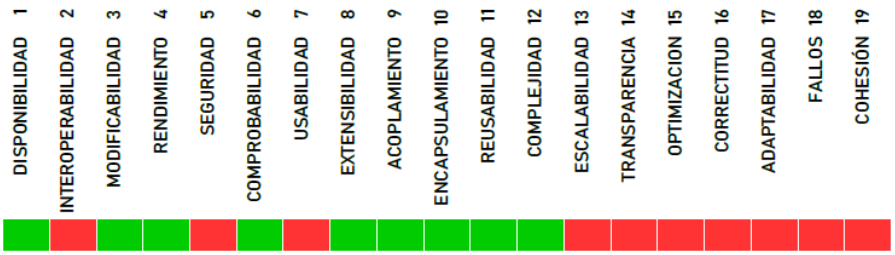

Figura 3.1: Atributos de calidad asociados al patrón Interceptor.

3.7.2. Patrón Broker

El patrón Broker nos ofrece la posibilidad de modelar sistemas distribuidos compuestos por componentes software desacoplados que interactúan mediante invocaciones a servicios remotos. Para ello el patrón coordina la comunicación, redireccionando peticiones y transmitiendo resultados y excepciones. El problema que existe radica al estructurar componentes configurables dinámicamente e independientes de los mecanismos concretos de

comunicación de un sistema distribuido. Para ello podemos introducir un componente broker para un mejor desacoplamiento entre cliente y servidor.

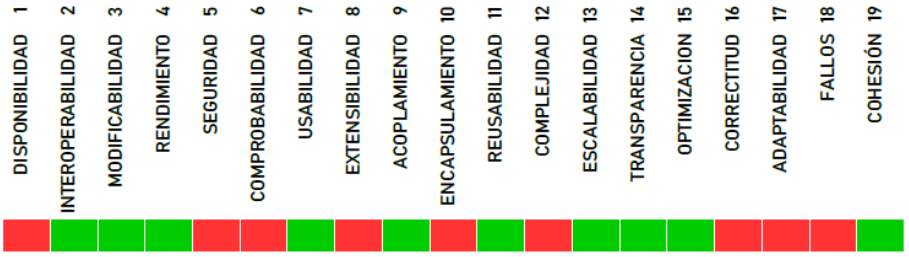

Figura 3.2: Atributos de calidad asociados al patrón Broker.

3.7.3. Patrón Reflection

Nos permite modificar la estructura y comportamiento de sistemas software de forma dinámica. En este patrón tenemos un nivel que proporciona información sobre propiedades del sistema y otro cuya responsabilidad principal consiste en incluir la lógica de la aplicación. Es útil para sistemas que necesiten soporte para realizar cambios propios y proporcionar persistencia de sus entidades. La aplicación en este caso se divide en dos niveles: metanivel para información de ciertas propiedades del sistema que incluyen las reglas para modificar las clases de la aplicación y un nivel-base con las clases de la lógica de la aplicación.

La utilización de este patrón puede llevar a problemas como el modificar el comportamiento de los objetos de una jerarquía dinámicamente, sin afectar los propios objetos en su configuración actual. Ante ello, el software puede ser auto-consciente de su función y comportamiento.

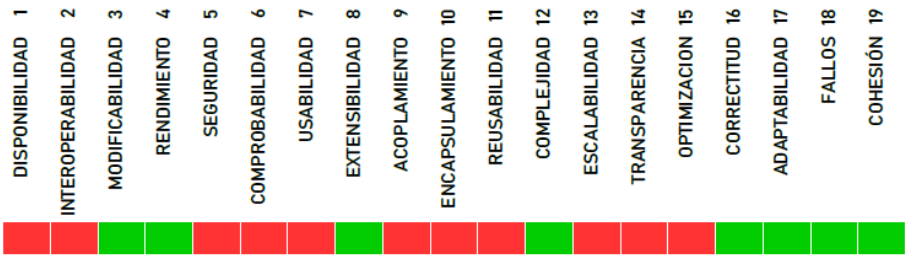

Figura 3.3: Atributos de calidad asociados al patrón Reflection.

3.7.4. Patrón Blackboard

En el diseño de patrones, Blackboard o patrón de pizarra proporciona un marco computacional para el diseño e implementación de sistemas que integran módulos especializados grandes y diversos. Es útil para atacar problemas no deterministas a partir de transformaciones de datos sin formato en estructuras de datos de alto nivel. Su idea general es implementar programas independientes que trabajan de forma cooperativa bajo una estructura de datos, ofreciendo una posible solución parcial.

Un componente blackboard central evalúa el estado de procesamiento y coordina la actividad de los programas especializados.

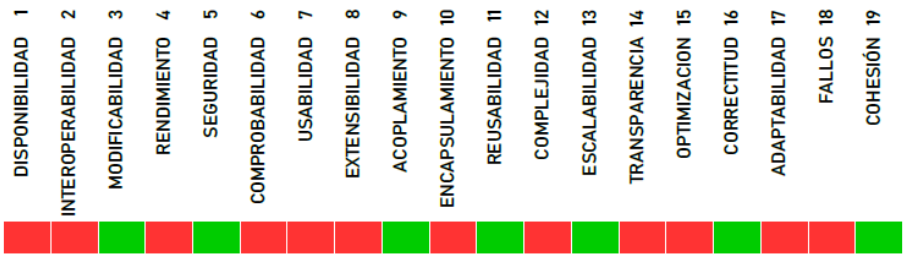

Figura 3.4: Atributos de calidad asociados al patrón Blackboard.

3.7.5. Patrón MVC

Patrón explicado anteriormente.

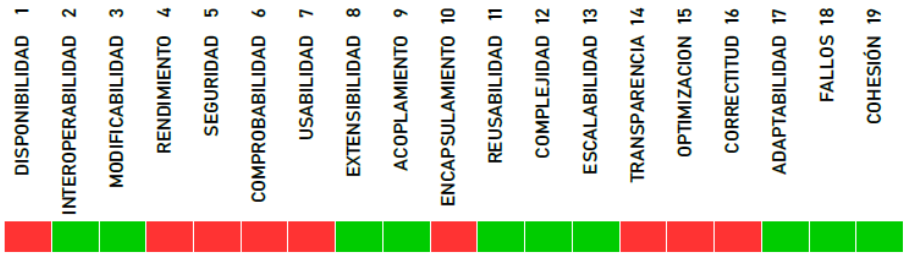

Figura 3.5: Atributos de calidad asociados al patrón MVC.

3.7.6. Patrón Microkernel

Este patrón arquitectónico se aplica a sistema de software con riesgo de sufrir cambios constantes, con capacidad para adaptarse a los requisitos cambiantes del sistema. Separa un núcleo funcional mínimo y partes determinadas del cliente. Implementa servicios semánticamente coherentes, extendidos a servidores internos y externos. Tiene también la función de socket para conectar estas extensiones y coordinar su colaboración.

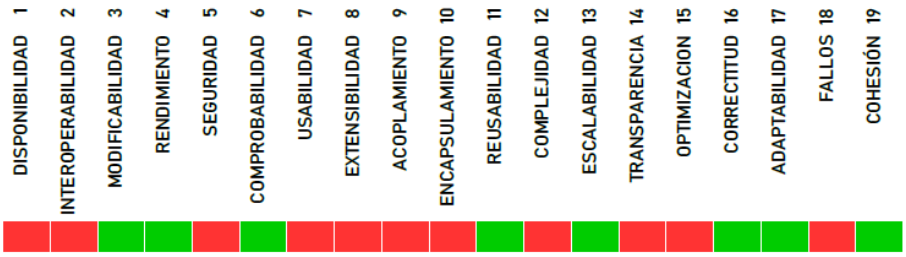

Figura 3.6: Atributos de calidad asociados al patrón Microkernel.

3.7.7. Patrón DCI

El patrón DCI (Datos, Contexto e Interacción) es un patrón de diseño destinado a sistemas complejos con el objetivo de aportar mecanismos simples para la relación entre modelos del usuarios y la implementación y organización de los requisitos funcionales. En términos puros de código, plantea separación de clases en un caso de uso concreto, colocándose en un artefacto distinto llamado contexto.

Cada una de las letras que definen al patrón son las partes que lo componen. D (Datos) tiene relación con el modelado de datos, representan las entidades del negocio. C (Contexto), mapear los roles en objetos por cada caso de uso. I (Interacción), donde se desarrolla el sistema, lo que hace y cómo lo hace.

DCI es complementario a MVC.

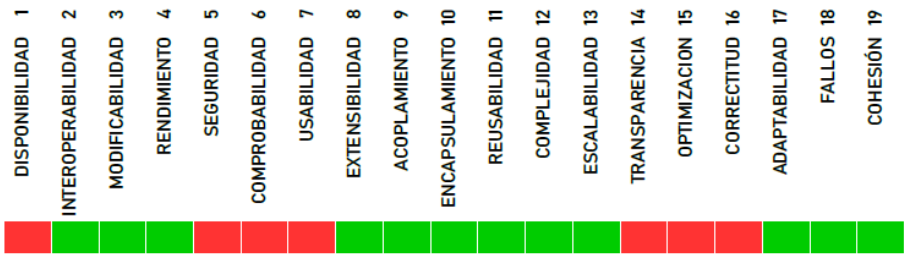

Figura 3.7: Atributos de calidad asociados al patrón DCI.

Capítulo 4

Desarrollo de la metodología ágil e implantación en Scrum

4.1. Introducción

Una vez hemos abarcado todo el flujo de información referido al desarrollo de software, partiendo desde sus inicios, introduciendo las metodologías de desarrollo existentes hasta las más actuales (conocidas como ágiles), los distintos tipos de arquitectura de software, los patrones más utilizados y el debate entre el diseño y el desarrollo ágil, se propone una forma de unificar ambos conceptos y aprovechar las ventajas que nos ofrecen.

Como comentamos anteriormente, la arquitectura de software es un elemento fundamental en la obtención de un producto que satisfaga los requisitos no funcionales, muy ligados a la obtención de los atributos de calidad. Emplear un tiempo en la elección de una arquitectura correcta nos va a proporcionar el diseño y las especificaciones sobre las cuales se construirá el sistema solicitado por el cliente, como las relaciones existentes entre elementos o circunstancias de más bajo nivel. El alcance de la arquitectura puede ser el de una aplicación única, de una familia de aplicaciones, y su uso podría estar orientado a una organización o a una infraestructura como Internet compartida por muchas organizaciones.

La existencia de una fase encargada del diseño de un sistema software, dentro de una metodología ágil, suscita la necesidad de realizar algunas consideraciones. Una arquitectura de desarrollo de software tiene una serie de pasos establecidos, que se van completando a un ritmo determinado y que no tienen por qué tener ninguna base ágil. Por otra parte, si unificamos el diseño y un procedimiento ágil podremos seguir un esquema de implantación clásica de la arquitectura en la metodología o bien podemos adaptar esta fase de diseño dentro una implantación ágil. Para el caso que nos concier-

ne, lo más deseable sería alterar el proceso de diseño para adaptarlo a la metodología ágil que utilicemos como base, ya que este paso será uno más dentro del desarrollo completo del proyecto. Este enfoque permite que la arquitectura de un sistema evolucione con el tiempo, a la vez que respalda las necesidades de los usuarios. Esto evitaría la falta de eficiencia del proceso ágil, que producirían los retrasos en los plazos de entrega derivados del rediseño inherente a una metodología de este tipo respecto del mantenimiento y posterior evolución de un sistema.

Algunos datos interesantes sobre la aplicación de arquitecturas de software en desarrollos ágiles:

- En '2008 Agile Practices and Principles' se obtuvo que la mayoría de los encuestados habían informado que estaban obteniendo un valor positivo a cambio de los esfuerzos iniciales de visualización de la arquitectura y de permitir que los detalles del diseño se tuvieran en cuenta a lo largo del desarrollo del proyecto.

- En '2009 Agile Project Initiation Survey', el 86 de los encuestados señaló que en su proyecto ágil más reciente tuvieron que realizar algún tipo de diseño de arquitectura /diseño inicial.

- En '2008 Modeling and Documentation Practices on IT Projects Survey' se encontró que el enfoque más popular para el modelado de una arquitectura software era crear diagramas de alto nivel. Esa encuesta también encontró que los equipos ágiles, ayudados por una definición arquitectónica previa, tenían más probabilidades de modelar correctamente el sistema objetivo que los equipos tradicionales.

4.2. Propuestas de diseño ágil

El diseño arquitectónico sobre un enfoque ágil no es un tema lo suficientemente estudiado a día de hoy como para que pueda existir una respuesta predeterminada a cuestiones como: ¿dónde se debe ubicar el encargado del diseño en las fases de la metodología?, ¿cuándo debemos tener en cuenta el diseño?, ¿es necesario elaborar todo el diseño de la arquitectura o sólo parcialmente? Es decir, para desarrollar el resto durante las fases posteriores.

4.2.1. Propuesta 1

Existen empresas que cubren este asunto como la empresa Scaled Agile, que propone el framework 'SAFE' (Scaled Agile Framework) como una base de conocimiento de patrones de éxito comprobados, para las personas y sistemas que crean software. Permiten tanto diseño emergente como una serie de iniciativas arquitectónicas que mejoran varios parámetros. La idea está respaldada por los siguientes principios:

El diseño emerge como una colaboración. Para evitar que un primer diseño inicial sea frágil ante cambios posteriores, consumiendo un valioso tiempo, se propone un diseño emergente impulsado mediante la colaboración.

Cuanto más grande es el sistema, más largo es el camino. Las propuestas arquitectónicas se desarrollan de forma incremental mediante la creación de tareas en un espacio de tiempo determinado.

Construir la arquitectura más simple que lleve al sistema objetivo. Toman algunas consideraciones tales como:

- Utilizar un lenguaje simple e informal para describir el sistema.
- Hacer que el modelo de la solución sea lo más parecido posible al modelo del dominio del problema.
- Refactorizar el software producido continuamente.
- Asegurase que las interfaces de los objetos y componentes que hayamos definido sirvan para expresar claramente su funcionamiento.
- Seguir buenos principios de diseño contrastados.

En caso de duda, codificarlo o protegerlo. Ante posibles soluciones de diseño, los equipos ágiles producen código que sirve para crear pequeños prototipos para probarlos y evitando en lo posible la refactorización (una técnica de la Ingeniería de Software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo).

Ellos lo construyen, ellos lo prueban. Los encargados del diseño son los responsables de probar la arquitectura creada, testeando su capacidad para cumplir los requisitos. Para hacer esto, los equipos también deben construir la infraestructura de prueba y automatizar las pruebas siempre que sea posible, para permitir la realización continua de pruebas a nivel del sistema.

No hay monopolio en la innovación. Las 'ideas felices' pueden venir de cualquier miembro del equipo, favoreciendo la colaboración de todos los equipos.

Implementar el flujo arquitectónico. Las funciones e iniciativas arquitectónicas siguen un patrón de flujo de trabajo común de exploración, refinamiento, análisis, priorización e implementación.

Un posible primer acercamiento a cómo introduciríamos el diseño arquitectónico dentro de una metodología de desarrollo ágil como Scrum podría ser definirlo como un primer 'sprint inicial' o 'sprint cero'. Esta idea la desarrollaremos posteriormente.

4.2.2. Propuesta 2

Otra empresa que propone un acercamiento es 'Scott Ambler + Associates'. Ofrece servicios de capacitación y consultoría para la implantación ágil en una empresa. A partir de un artículo propone una serie de ideas y profundiza en temas relacionados:

En primer lugar presenta algunas ideas generales como:

- Valorar en igual medida el trabajo del encargado del diseño arquitectónico en el proyecto ágil al igual que el resto del equipo. La humildad es un factor influyente en el éxito. En resumidas cuentas, el rol del arquitecto es válido para la mayoría de los proyectos, simplemente no debe ser un rol que alguien realice sobre un pedestal.
- Se debe tener especial hincapié en integrar a los encargados del diseño dentro del desarrollo cotidiano del equipo sobre el proyecto. Llaman arquitecturas de torre de marfil a las creadas a partir del aislamiento del creador del diseño. Esto lleva problemas a largo plazo, ya que dichas arquitecturas son funcionales teóricamente, pero en la práctica el encargado del diseño no es el que implementa lo propuesto, con lo cual se pueden ocasionar problemas de interpretación y también sobrevalorar las posibilidades reales del equipo para llevarlo a cabo.
- La existencia de casos en los que un equipo de desarrollo no necesita de una arquitectura software del sistema objetivo, como tal, debido a la experiencia de los miembros del equipo que lo están trabajando sólo necesita realizar consultas entre ellos. Sin embargo, la razón por la que, en este caso, es aconsejable realizar un modelado arquitectónico es para abordar el riesgo de que los miembros del equipo no trabajen para lograr llegar a una visión común de lo que tienen que hacer, ya sea

por no compartir la misma ubicación o porque se hacen responsables de tareas de tamaño considerablemente más grandes de lo normal.

Aborda también el problema de la responsabilidad del rol del arquitecto. Al contrario que otras propuestas existentes, en ésta se aboga por no cargar el peso del diseño en una única persona. En equipos de trabajo es preferible abordar la arquitectura a partir de todos los miembros, ya que la arquitectura es un elemento demasiado importante. La idea es que cuando el equipo completo desarrolle una arquitectura, a menudo las personas están mucho más dispuestas a replantearse su enfoque porque es un problema de equipo y no un problema personal. Por otro lado, no es la mejor solución para equipos de gran escala y puede provocar discrepancias.

Con respecto a la base para crear el diseño arquitectónico, es evidente que se ha de obtener del análisis de los requisitos. Estos requisitos provienen de los interesados en el proyecto, no de los desarrolladores. Cuando se trabaje en los aspectos técnicos de la arquitectura, deberíamos basarnos en los requisitos técnicos, en las limitaciones y, posiblemente, en los posibles cambios. De manera similar ocurre cuando se trabaja en aspectos comerciales de la arquitectura, identificando potencialmente subsistemas de software o componentes comerciales.

De igual manera, todo diseño se debe modelar y es lo que propone el modelo propuesto por la empresa 'Scott Ambler + Associates'. El objetivo de modelar es llegar a un entendimiento común entre todos sobre cómo construir el sistema. Todos los equipos necesitan de un modelado y el proceso que proponen consiste en crear uno/varios diagramas que presenten una visión general.

Otra visión dentro del diseño ágil es tener en cuenta las limitaciones de la empresa. Los equipos disciplinados construyen sistemas cuya arquitectura emerge dentro del entorno organizacional en el que están trabajando y bajo las limitaciones de la infraestructura existente. Además, producen soluciones potencialmente utilizables que funcionan dentro del ecosistema de su organización.

4.3. Principios a cumplir por la metodología

Estos son algunos de los principios propuestos dentro de un desarrollo ágil para favorecer la integración del diseño dentro del entorno de trabajo:

4.3.1. Principios básicos

4.3.1.1. Un solo arquitecto para definir la arquitectura

Para grupos de trabajo grandes lo deseable es un solo miembro de diseñador. Debe ser alguien experimentado en el trabajo y capaz de adaptarse a las posibilidades y limitaciones de su entorno de trabajo. El arquitecto puede apoyarse en otros miembros del grupo para responder cuestiones para tomas de decisiones de menor importancia. En casos de proyectos con un grupo más reducido, sería posible una diseño de la arquitectura compartido por todos.

4.3.1.2. El progreso lo ofrece el software

Si bien el diseño arquitectónico es totalmente opuesto al tipo de mentalidad ágil, nos encontramos en una metodología ágil y por tanto debemos adaptar este procedimiento a estos principios. Por tanto, aunque una documentación completa es muy importante, el trabajo conseguido debe tener prioridad por encima del resto. Las especificaciones de la arquitectura, los documentos del diseño, los procesos de aprobación, etc., pueden ser importantes, pero solo cuando nos acercan a nuestro objetivo de trabajar con el software

4.3.1.3. El arquitecto asume sus responsabilidades

Si nos encontramos ante un caso en el que la responsabilidad de diseño recae en una persona, ésta ha de ser responsable de los resultados obtenidos. El rol del arquitecto no puede delegarse en otras personas.

4.3.1.4. Resolver únicamente problemas conocidos

Un arquitecto debe ocuparse de solucionar problemas que surjan durante el proceso de desarrollo sin adelantarse a otros futuros que puedan ocurrir. La arquitectura a corto plazo es lo relevante.

4.3.1.5. Usar SAAM

A partir de SAAM (Software Architecture Analysis Method) podemos evaluar una arquitectura de sistema. Es útil para analizar las arquitecturas candidatas y comunicar las ideas entre todos los miembros. Fue el primer método documentado de análisis de arquitectura de software, y se desarrolló a mediados de la década de 1990 para analizar un sistema de modificabilidad, pero es útil para probar cualquier aspecto no funcional.

4.3.1.6. Dedicarse a las decisiones arquitectónicas importantes

El arquitecto debe centrarse en las decisiones relevantes: huir de decisiones de menor impacto, discusiones irrelevantes con otros miembros y evitar en cualquier caso situaciones que provoquen un retraso.

4.3.2. Principios en el ámbito organizativo

4.3.2.1. Promover equipos autoorganizados

Una colaboración creativa implica que alguien que realmente utilizará el producto final conducirá a una arquitectura que se adapte a las necesidades del equipo, así como a los requisitos de la empresa. Un contrato arquitectónico transmitido desde lo alto a menudo no lo hará.

4.3.2.2. Compartir las decisiones conflictivas

Ante decisiones de diseño que presenten algún tipo de conflicto, es preferible compartirla con el resto de miembros del equipo de desarrollo.

4.3.2.3. Grupos de arquitectura verticales

Una buena práctica de desarrollo podría ser implicar a varios miembros de distintos módulos en determinadas tareas de la arquitectura formando grupos de trabajo. Esto llevaría a conocer mejor la arquitectura a todo el equipo.

4.3.2.4. Usar un lenguaje de alto nivel

Utilizar un lenguaje altamente profesional, mediante el conocimiento común de las técnicas y herramientas utilizadas por el equipo de profesionales.

4.3.2.5. La arquitectura como un servicio

La evolución de la arquitectura conforme el desarrollo del software es un requisito más de calidad. Plantear el proceso como un servicio ayuda a su evolución.

4.3.3. Principios en el ámbito personal

4.3.3.1. Motivación

La motivación es un aspecto fundamental para seguir unas pautas de desarrollo regidas por una estructura. Es esencial mantener un equipo motivado para hacer un buen trabajo.

4.3.3.2. Mejora personal

La profesión, como muchas otras, implica un aprendizaje continuo durante el resto de la vida laboral. Los campos en los que nos basamos tienen la capacidad de cambiar y mejorar con el tiempo y nuestra labor es adaptarnos. Para ello, debemos conocer el dominio del proyecto y poseer algunas habilidades para comunicar, aprender y mejorar.

4.3.3.3. Confianza en tus compañeros

El encargado de la arquitectura y el equipo encargado de desarrollar tendrán una confianza mutua, asegurando que ambas partes llevarán a cabo lo mejor posible.

4.3.3.4. Uso de patterns

El uso de patrones es fundamental para comunicar a todo el equipo de forma simplificada y ágil, por lo que es esencial su conocimiento.

4.3.4. Principios de calidad

4.3.4.1. Definición formal de los requisitos de calidad

Los requisitos de calidad son la base fundamental para el desarrollo de la arquitectura. Estos deben probar que la arquitectura es valida y cada uno de ellos ofrecer la posibilidad de ser testeable. Es decir, debe ofrecer una medida que valore cuanto cumple el requisito de calidad en cuestión. La documentación de los mismos debe ser formal.

4.3.4.2. Simplificar

Para evitar en futuras fases de desarrollo el diseño e implementación de funciones totalmente operativas que no serán útiles, debemos partir de un diseño lo más simple y claro posible. A partir de ello, obtener un producto funcional y con la calidad necesaria será más sencillo.

4.3.4.3. Buscar la excelencia

Una arquitectura ágil debe encontrar el equilibrio adecuado para el equipo, el software, el entorno y la empresa. El mejor diseño arquitectónico va a permitir que los cambios que suelen producirse en desarrollos ágiles estén contemplados.

4.3.4.4. Testear los requisitos de calidad

Definir baterías de test, que validen los requerimientos de calidad. Por ejemplo:

- Tiempo de respuesta promedio
- Número de usuarios concurrentes, etc.

4.3.5. Principios de validación y control

4.3.5.1. Utilizar la taxonomía de los riesgos operativos del SEI

Un informe desarrollado por el Software Engineering Institute (SEI) de la Universidad Carnegie Mellon presenta una categorización que trata de identificar los riesgos de software en proyectos informáticos. Se organiza en 4 clases (gestión de proyectos, ingeniería del producto, entorno de desarrollo y restricciones del programa), 22 elementos y 111 atributos. En estos últimos se construyen cuestionarios o listas de comprobación de elementos de riesgo para la identificación de los mismos

4.3.5.2. Uso de prototipos y tracer bullets

Usar para la validación tanto prototipos como "tracer bullets"(traducido como balas trazadoras). Por un lado el uso de prototipos nos proporciona respuestas para preguntas específicas (abaratando los costes) y un tracer bullet se basa en la creación de código en el mismo entorno y restricciones del proyecto con consecuencias minimizadas, obteniendo el funcionamiento del sistema a una tarea.

4.4. La arquitectura en Scrum

Anteriormente dedicamos un apartado orientado a explicar Scrum y los conceptos fundamentales que lo completan. Por ello, podemos ver como Scrum es un tipo de metodología o gestión de desarrollo de software ágil basado en un trabajo incremental. Resumiendo sus características más importantes a modo de recordatorio, Scrum ofrece:

- Resultados en un tiempo reducido.
- Adaptación a proyectos con requisitos cambiantes.
- Solución a desarrollos conflictivos.

La competitividad del mercado de desarrollo de software y la necesidad de los clientes de reducir el "time to market" obligan a las organizaciones de desarrollo de software a ser agresivas en sus calendarios de entrega. Esto ha hecho que hayan surgido metodologías de desarrollo de software ágiles tales como esta.

Scrum se ejecuta en ciclos cortos de una duración determinada (2,3 o incluso 4 semanas) con resultados completos, proporcionando un incremento en el desarrollo general. Los roles existentes hasta la fecha en la metodología son el 'product owner', 'scrum master' y 'stakeholders'. Bajo estas directrices, no existe nadie encargado de la arquitectura por lo que es importante tener definido al final de este proyecto el rol encargado de diseñarla, las tareas que deberá llevar a cabo y la ubicación de este proceso dentro de Scrum.

Deberiamos de poder responder a las siguientes preguntas una vez que la metodología esté totalmente definida: ¿Quién/Quienes serán los encargados de desarrollar la arquitectura? ¿En que fases se ubicará la metodología? ¿Cuál será la base de la arquitectura ágil?

4.4.1. Flujo de trabajo original de Scrum

Scrum se basa en 'sprints', periodos de tiempo planificados desde el comienzo y con un resultado esperado al final de cada uno. El flujo de trabajo en Scrum (sin añadir ningún proceso encargado de la arquitectura) es el siguiente:

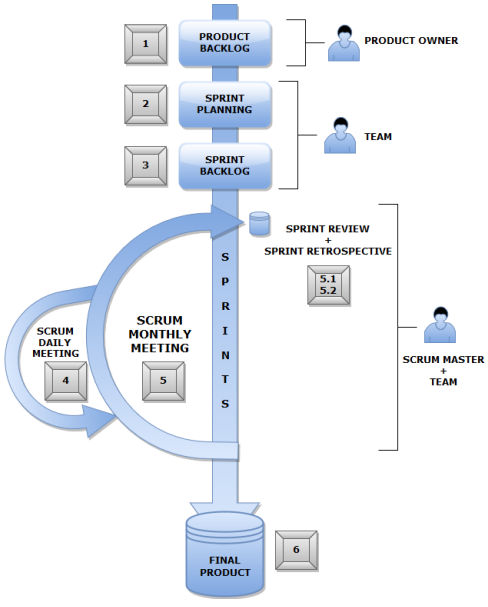

Figura 4.1: Esquema del ciclo de desarrollo original de Scrum. Basado en la figura de Alejandro Frechina, https://winred.com/management/metodologia-scrum-que-es/gmx-niv116-con24594.htm

Tenemos un ciclo de desarrollos con un número de fases establecidas: todo lo relacionado con requisitos y planificación del sprint en las tres primeras fases y el resto reuniones y análisis de los resultados obtenidos. Para cada fase existe un rol encargado del mismo, que puede ser el equipo, el cliente o una especie de líder equipo que actúa como una protección entre el equipo y cualquier influencia que les distraiga. Esta iteración de pasos se repetirá hasta conseguir el producto pedido.

4.4.1.1. Product Backlog

Todo desarrollo de software bajo una metodología Scrum comienza con una reunión inicial con el cliente. En esta fase el equipo de desarrollo pregunta las dudas que surjan y se genera una lista de requisitos propuesta por el cliente interesado (bajo una duración de unas 4 horas como máximo). En esta lista trata de valorar los objetivos, priorizando su valor y el equipo evalúa su coste (la lista representa la visión y expectativas del cliente respecto a los objetivos y entregas del producto o proyecto). Por tanto, para cada iteración se asociará los requisitos que podrán ser entregados al cliente. La lista también tiene que considerar los riesgos del proyecto e incluir los requisitos o tareas necesarios para mitigarlos.

Este documento se denomina 'product backlog', que contiene los requisitos mencionados junto a descripciones genéricas de funcionalidades solicitadas ordenadas por prioridad. Solo puede ser modificado por el cliente.

4.4.1.2. Sprint Planning

La primera fase de un sprint (recordemos que es iterativo) debe estar orientada a la planificación del sprint en cuestión determinando el trabajo a realizar para desarrollar los requisitos asociados a este sprint (en un máximo de 4 horas).

4.4.1.3. Sprint Backlog

El tramo del sprint dedicado a abarcar (mediante una reunión) en profundidad el conjunto de requisitos a desarrollar se denomina 'Sprint Backlog'. Estos requisitos se dividirán por tareas, cada tarea asociada a un tiempo máximo establecido y para aquellas que sea imposible en ese tramo de tiempo, serán divididas en tareas de menor recorrido. Estas tareas no se asignan automáticamente, son autoasignadas por miembros del equipo.

Sirve como compromiso con el cliente de las funcionalidades que tendrá el proyecto al final del sprint.

4.4.1.4. Scrum Daily Meeting

Durante el transcurso de un sprint, que puede durar entorno a un mes, cada día se realiza una breve reunión de sincronización (con una duración entorno a los 15 minutos) para compartir los avances realizados, los problemas encontrados y aportar soluciones o variantes a otros compañeros. Las preguntas a responder al finalizar la reunión deben ser: qué avances he llevado a cabo, los problemas encontrados, tareas futuras y posibles problemas que puedo encontrar.

4.4.1.5. Scrum Monthly Meeting

Tras finalizar un sprint, transcurre una reunión global que incluye dos pasos

4.4.1.6. Sprint Review

En esta primera fase (de unas 4 horas), el equipo presenta el progreso realizado al cliente, como un incremento o avance del producto anterior y funcional. Tras los resultados mostrados, el cliente tiene la posibilidad de replantear los requisitos solicitados o no.

4.4.1.7. Sprint Retrospective

En esta segunda fase (de unas 4 horas), el equipo hace autocrítica del trabajo realizado hasta el momento con el objetivo de mejorar de manera continua su productividad y la calidad el producto en cuestión. Se deben de tener en cuenta los siguientes factores:

- Conocimiento adquirido en base a la experiencia.
- Elementos que han funcionado.
- Elementos que no han funcionado.
- Qué problemas han surgido.

- Qué debería probar en el siguiente sprint.

4.4.1.8. Final Project

Incremento final del producto con todas las funcionalidades implementadas y listo para ser entregado al cliente.

4.4.2. Flujo de trabajo con arquitectura en Scrum

El flujo de trabajo de Scrum a día de hoy es más que conocido y, como está establecido, descarta por completo cualquier planificación o utilización del diseño de una arquitectura de software previo al inicio del proceso de desarrollo. La idea que presentamos aquí consiste en tratar de proponer distintas soluciones para incluir un rol de arquitecto y, consecuentemente, su trabajo: la definición de una arquitectura software que sirva de base inicial al desarrollo, dentro las actividades de una metodología ágil. Dado que los métodos ágiles "funcionan" mediante la realización de "sprints" en unos plazos establecidos, hemos de tratar de incluir este diseño tratando de afectar lo menos posible el flujo de trabajos del método ágil original.

Por consiguiente, nos proponemos lograr una integración "sin costuras" de una arquitectura de software dentro del marco de trabajo Scrum. Dado que dicha integración es aún un tema de estudio poco explorado en ambientes académicos y profesionales, existe poca documentación o artículos relacionados para conseguir llevarlo a cabo, por consiguiente, nuestra propuesta puede considerarse como un modelo de gobernanza de la gestión ágil de proyectos y que se aplica a un método útil para el desarrollo de sistemas software complejos, como es el citado Scrum.

Para llevar a cabo nuestra propuesta, propondremos varias soluciones para distintos casos y nos quedaremos con una en concreto para profundizar más en ella.

4.4.2.1. Rol de arquitecto externo al equipo Scrum

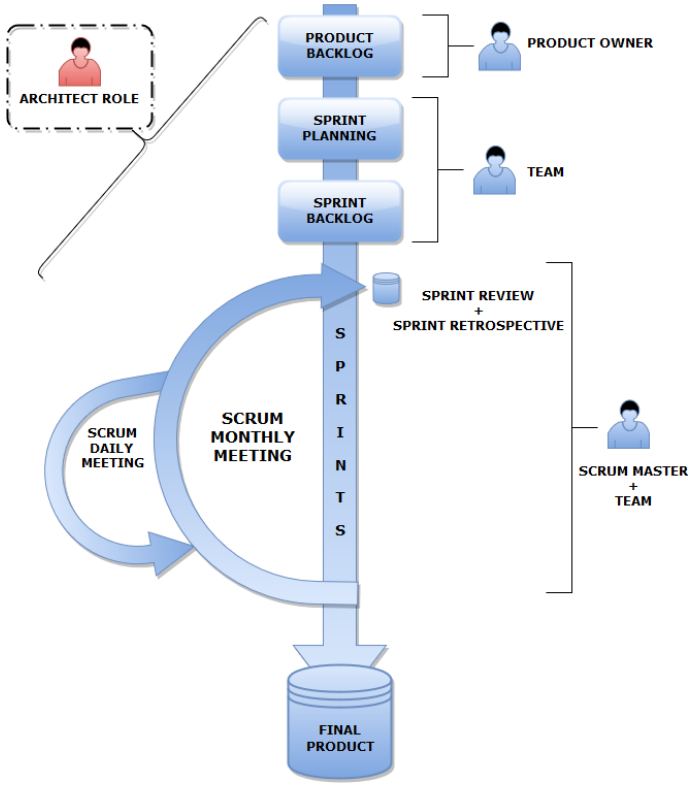

Figura 4.2: Posible esquema del ciclo de desarrollo tomando como rol de arquitecto un miembro externo al equipo.

La primera opción es el uso de un rol (individual/grupal) de arquitecto externo al equipo principal de Scrum. Este rol podría ocuparlo una compañía dedicada a diseño arquitectónico o una sola persona. La intención es aportar soporte al equipo para diseñar la aplicación siguiendo una serie de pautas de diseño e influir además sobre el cliente. No es recomendable para proyectos donde la arquitectura de software es fundamental en el proceso de desarrollo, ya que la principal aportación del arquitecto externo consistirá en la toma de decisiones concretas, que podrían ser en cierto modo contradictorias con el diseño previo, en proyectos orientados a la arquitectura.

4.4.2.2. Rol de arquitecto dentro del equipo

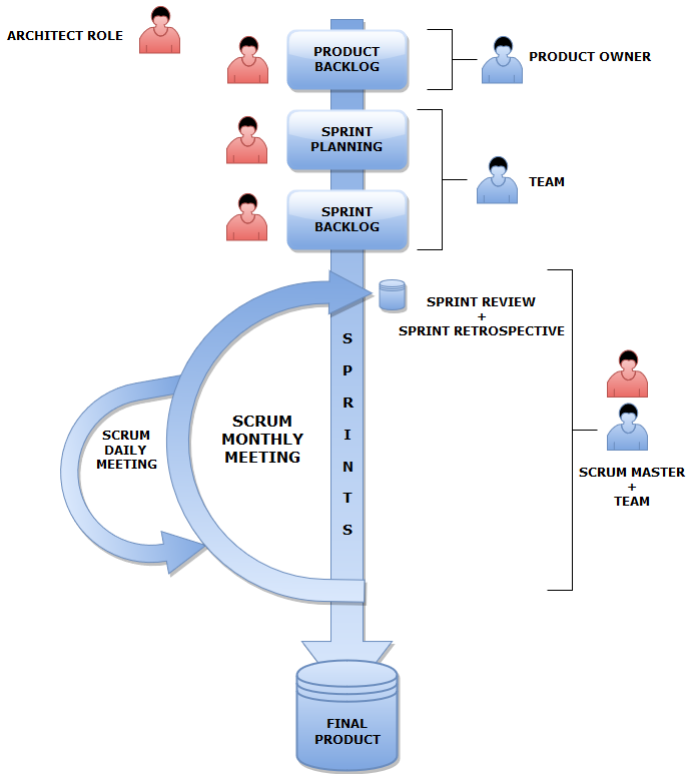

Figura 4.3: Posible esquema del ciclo de desarrollo tomando como rol de arquitecto un miembro/varios miembros del equipo.

La segunda opción es el uso de un rol (individual/grupal) dentro del equipo principal de Scrum. Esta idea provoca que el nuevo rol debe influir en todas las etapas de Scrum: 'Product Backlog' podría servir ahora para determinar los requisitos arquitectónicos, 'Sprint Planning' y 'Sprint Backlog' para planificar las tareas asociadas a determinados componentes arquitectónicos y velar por el desarrollo del sprint. El rol puede ser ocupado por varios miembros, pero deben tener los conocimientos necesarios para dirigir un proyecto de desarrollo basado en una arquitectura de software.

4.4.2.3. Rol de arquitecto como dueño del producto

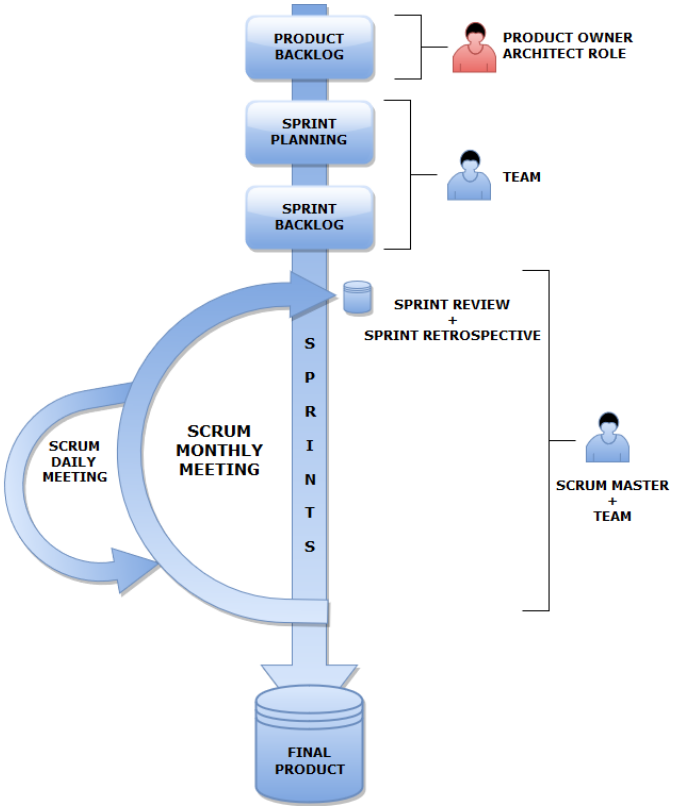

Figura 4.4: Posible esquema del ciclo de desarrollo tomando como rol de arquitecto al dueño del producto.

La tercera opción es el uso de un rol (individual/grupal) como dueño del producto. Esta idea es específica para un caso en concreto, donde el resultado final sea un producto requerido dentro de la compañía, por tanto, el rol lo ha de ocupar un arquitecto de software, que tomará las decisiones oportunas para priorizar tareas y definir las características y los atributos de calidad relacionados con la arquitectura software seleccionada para desarrollar el proyecto.

4.4.2.4. Rol de arquitecto como recurso externo

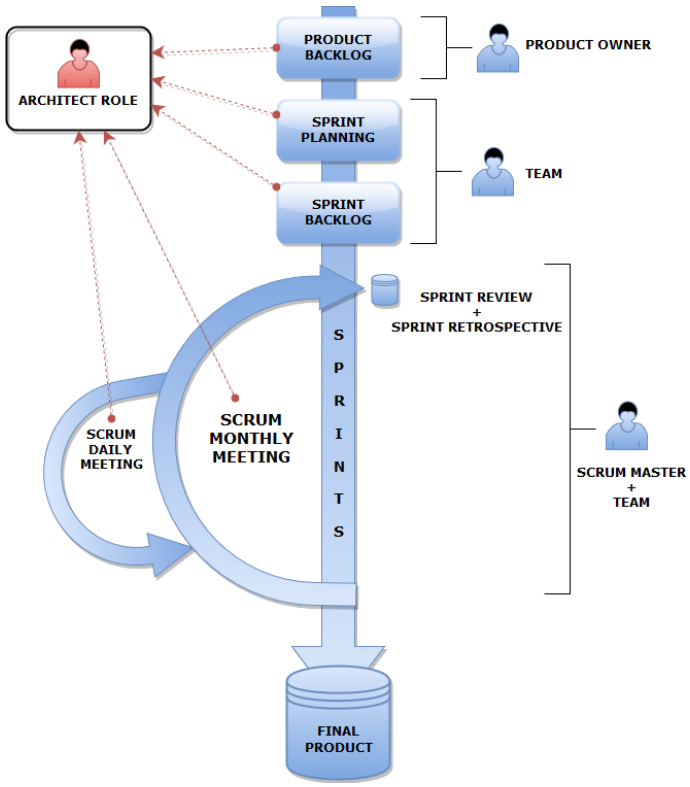

Figura 4.5: Posible esquema del ciclo de desarrollo tomando como rol de arquitecto a un recurso externo.

La cuarta opción es el uso de un rol (individual/grupal) como un recurso externo. De entre todas las opciones, esta sería la que representaría un menor involucramiento del arquitecto software con el proceso de desarrollo Scrum. Consecuentemente, en este caso, el arquitecto funcionaría como un asesor externo al equipo al que los miembros pueden acudir en cualquiera de las fases del sprint para resolver problemas o pedir ideas.
Se considera una opción óptima si queremos evitar los problemas de retrasos que puedan producirse en el desarrollo derivados de la implicación de un arquitecto en las tareas del día a día, que es una de las principales preocupaciones que motivan la idea del desarrollo ágil, en su versión más conservadora.

4.4.2.5. Sprint 0 y primera versión

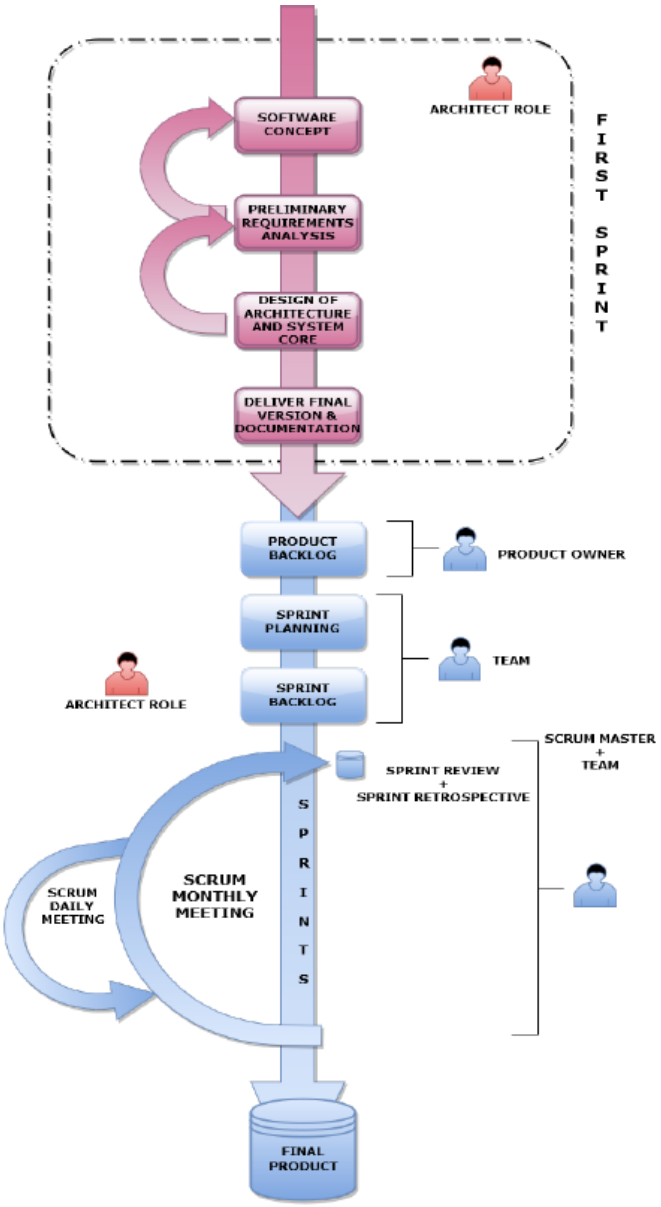

Figura 4.6: Posible esquema del ciclo de desarrollo tomando un sprint previo. Basado en la figura de la referencia [12]

Ninguna de las opciones anteriores nos asegura alcanzar los atributos de calidad deseados para nuestro producto , que se han de propiciar debido a la utilización de una arquitectura determinada. Esta quinta alternativa implica

ir un paso más.

En las anteriores propuestas añadíamos un rol de arquitectura, de forma externa o interna, al proceso de desarrollo que propone Scrum, involucrándose dicho rol en el ciclo de desarrollo directamente, sin llegar a realizar unos pasos previos propios de un sprint de Scrum. Para remediarlo, podemos aprovechar la existencia de un sprint extra que no está reconocido en las guías oficiales de Scrum pero que es conocido comúnmente como 'Sprint 0'. La introducción de un sprint previo es la solución más reconocida entre los usuarios del marco de trabajo Scrum para rellenar ese vacío que poseen metodologías ágiles respecto de todo lo relacionado con la preparación para un determinado proyecto, como son las siguientes cuestiones: ¿qué diseño y arquitectura vamos a usar? ¿en qué momento adaptamos los equipos para trabajar con el diseño arquitectónico?

La inclusión de un 'sprint 0' tiene tanto detractores como defensores. Para unos porque añadir un sprint de este tipo rompe con la mecánica ágil del resto del proceso de desarrollo y para otros porque que consideran que siempre es necesario trabajar sobre un base diseñada con antelación al resto del desarrollo del proyecto.

Entre muchas de las tareas que podemos asignar a este sprint 0, tendremos:

- Determinar el tamaño de los sprints posteriores.
- El número y tipo de pruebas a realizar.
- Qué herramientas de gestión del proyecto utilizaremos.
- Definir estándares de codificación y documentación del código.
- Definir la documentación.
- Realizar las instalaciones pertinentes.
- Definir la arquitectura base del proyecto.

Se recomiendan una serie de medidas a aplicar en este sprint :

- Al contrario que un sprint tradicional mensual, es recomendable cumplir con las tareas previstas en este 'sprint 0' dentro de un plazo de una semana aproximadamente.
- El objetivo no es crear un resultado incrementalmente, sino completar la planificación del proyecto y el conjunto de elementos estándar necesario para determinar el rendimiento del proyecto.

- Terminar todo lo necesario para estar listos para el primer sprint.

En el diagrama de desarrollo propuesto para esta quinta idea, trataremos de aportar una breve idea que extenderemos después sobre como añadir el diseño de la arquitectura en una metodología ágil como Scrum mediante la idea del 'Sprint 0'. No usaremos el término dado para este paso ya que es entendido como un sprint adicional para tareas previas, entre ellas el diseño. Por ello, denominaremos como 'Sprint 1' (first sprint') las tareas que se centran exclusivamente en la definición de la arquitectura.

En el 'Sprint 1' podemos encontrar cuatro pasos bien diferenciados, todos ellos tomados del ciclo de desarrollo propuesto por Bass en 'Software Architecture in Practice', que apostaba por un modelo evolutivo del ciclo de desarrollo del software donde se permite obtener retroalimentación de los usuarios y del propio cliente a través de la realización de varias iteraciones, además de añadir funcionalidad incrementalmente al producto en cada una de ellas.

Extraemos del diagrama la parte del nuevo sprint:

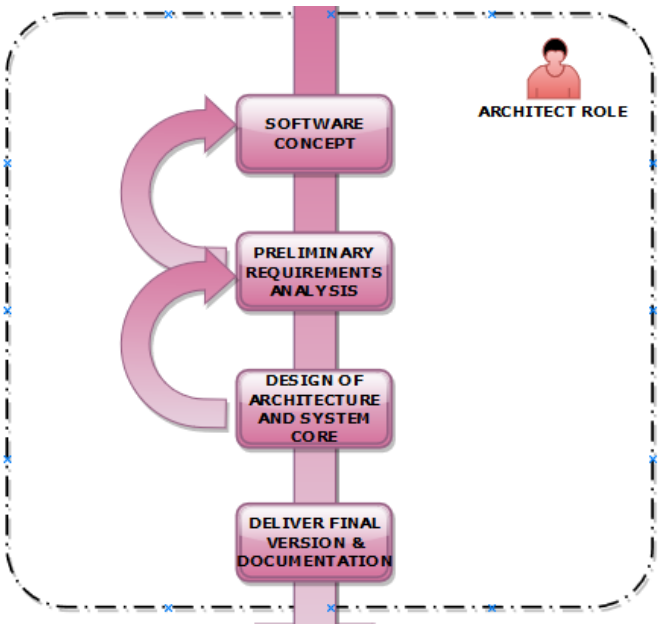

Figura 4.7: Sprint 0.

Tenemos estas 4 fases mencionadas que abarcan:

Concepto del Software : donde definimos el objetivo del software a crear, ya sea del sistema, programación de componentes o de una aplicación.

Análisis Preliminar de Requerimientos : el análisis de requisitos abarca aquellas tareas que determinan las necesidades o condiciones que debe cumplir un proyecto nuevo teniendo en cuenta los requisitos del cliente. En nuestro caso serán exclusivamente de diseño.

Diseño de la Arquitectura y Núcleo del Sistema : todo el proceso relacionado con el diseño de la arquitectura.

Entrega de la Versión Final y de la Documentación : versión final de la arquitectura junto a la documentación necesaria para continuar las directrices impuestas.

En esta primera versión podemos comprobar como las tres primeras fases no son lineales, pudiendo volver a la fase anterior si el concepto del software no es lo suficiente claro o definir con más exactitud los requisitos arquitectónicos dentro de la fase de diseño. El ciclo podría ser el siguiente:

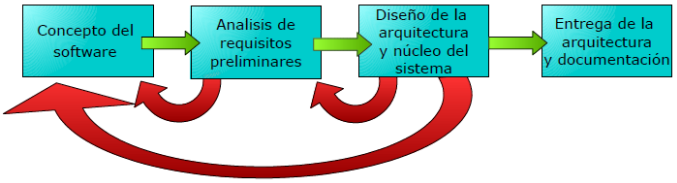

Figura 4.8: Ciclo del sprint.

En todo el 'Sprint 1' actuaría un grupo especializado en diseño de arquitecturas de software y éste, a su vez, puede actuar en sprints posteriores para modificaciones o para guiar al equipo en el seguimiento de unas pautas arquitectónicas.

Sin embargo, debemos profundizar más en la idea para adaptar de una mejor forma una arquitectura software dentro de la metodología ágil. Necesitamos poder elegir un módulo central a partir del cual podamos dividir en pequeños módulos que repartiremos en el sprint, seleccionaremos atributos de calidad acordes a un desarrollo ágil y, por tanto, también escogeremos los patrones arquitectónicos, cohesionaremos el 'Sprint 1' con los restantes y determinaremos las actividades de los roles encargados del diseño en todo el desarrollo, entre otras tareas.

4.4.3. First Sprint

Partiendo de la base de usar como referencia el 'Sprint 0', nuestra metodología compartirá varias de las características relacionadas con esta idea.

La idea fundamental a tener en cuenta a la hora de añadir un sprint al ciclo de desarrollo consistirá en integrar el diseño de la arquitectura de la forma menos intrusiva posible, es decir, sin alterar los principios "ágiles" que ha de poseer una metodología de este tipo. Aparte de ello, nos va a permitir poder organizar el proyecto y avanzar desde el minuto uno por el camino correcto. Esto será posible obteniendo una previsión general en esta etapa de todos los requisitos, el coste necesario, las características de calidad de la arquitectura elegida, el alcance o incluso una posible estrategia a seguir.

El añadir un paso extra para el diseño arquitectónico no implica condicionar todo el desarrollo a las decisiones tomadas. Como proponen los fundamentos ágiles, el objetivo es tener una base sobre la que trabajar, tomando una dirección técnica con todos los riesgos y beneficios previstos sobre una documentación detallada, pero no certera. Conforme el desarrollo progrese, deberemos adaptar la arquitectura con continuos refinamientos, si son necesarios, pero siempre desde una base sólida.

Una alternativa a este enfoque liviano para el modelado de arquitectura inicial es intentar definir completamente su arquitectura antes de que comience la implementación. Este concepto se conoce como "Big modeling up front" (BMUF). La consecuencia de su uso es que implica un consenso general sobre el enfoque antes de poder avanzar. Esto resulta en lentitud para el desarrollo provocando una arquitectura de torre de marfil que en la mayoría de los casos resulta frágil en la práctica, una arquitectura que es excesiva para lo que realmente se necesita y que ralentiza el trabajo. BMUF pertenece a una etapa antigua de desarrollo en cascada (sobre las décadas de los 70 y 80). La realidad es que el desarrollo de la arquitectura es muy difícil, un esfuerzo necesario para conseguir el éxito y la calidad en el producto pero que no se obtiene desde el principio.

Uno de los problemas que surgen es la elección del equipo o individuo dedicado a la arquitectura. En nuestra propuesta el rol de arquitecto lo tomarían todos los miembros del equipo o en todo caso un subgrupo del equipo. El propósito sería el involucrar lo máximo posible a todos los individuos en las decisiones tomadas en el diseño de la arquitectura y que éstas sean conocidas en reuniones diarias y mensuales de cada sprint. Esto aumenta la comprensión y la aceptación de la arquitectura por parte de todos porque trabajaron juntos como un equipo para ello.

¿Cuál es el inconveniente?

Para equipos de un tamaño compacto se podría llevar a cabo nuestra propuesta, pero si hablamos de entornos de trabajo a gran escala sería imposible aplicar esta metodología sin complicar en exceso el proceso de diseño en Scrum. En este caso, una posible solución sería adjudicar este rol a una única persona o a un grupo muy reducido, responsabilizándoles de todas las tareas y asegurando que el desarrollo prosigue adaptándose a las directrices

impuestas. El propietario único de la arquitectura es diferente al papel tradicional de arquitecto. En el pasado, el arquitecto solía ser el principal creador de la arquitectura y sería una de las pocas personas que trabajaron en ella. Desarrollaban la arquitectura y luego la presentaban o, más concretamente, la imponían sobre el equipo de desarrollo. Un propietario de arquitectura debe colaborar con el equipo para desarrollar y evolucionar la arquitectura. Aunque es la persona con la autoridad decisoria final cuando se trata de la arquitectura, esas decisiones deben tomarse en colaboración con el equipo. Estos deben ser desarrolladores con experiencia en las tecnologías con las que su organización está trabajando y que tienen la capacidad de trabajar en picos de arquitectura para explorar nuevas estrategias.

En cualquier caso, este rol lo tendrá una persona que será respetada por el resto del equipo. Esta persona ha de poseer la capacidad de trabajar intensivamente en determinados momentos críticos para explorar nuevas estrategias. Deberá poseer las habilidades necesarias para comunicar la arquitectura a los desarrolladores y a otras partes interesadas del proyecto. También deben tener una buena comprensión del dominio comercial.

Un posible diagrama que represente el flujo de trabajo podría ser el siguiente:

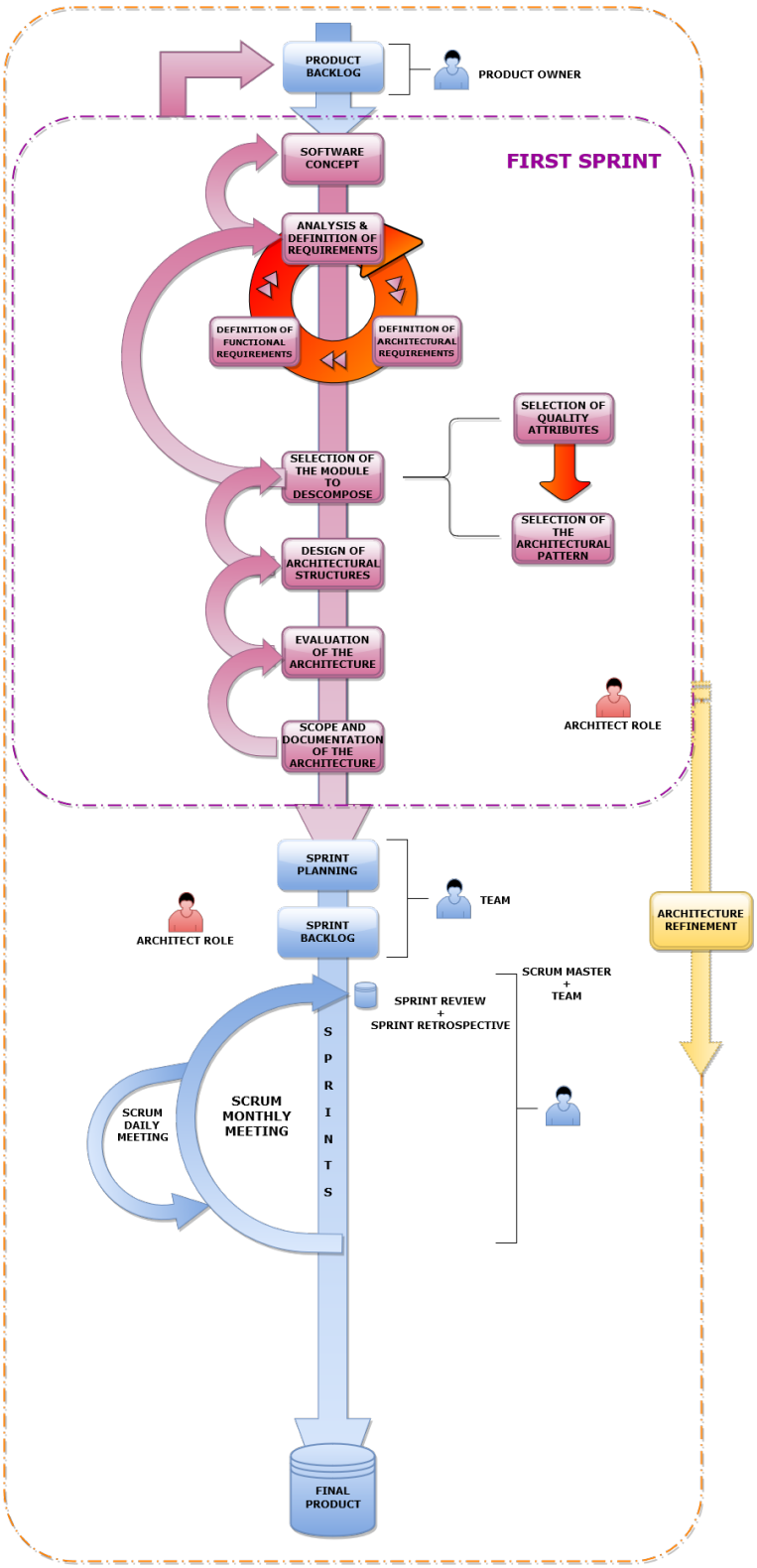

Figura 4.9: Esquema del ciclo de desarrollo propuesto en una metodología ágil con diseño previo de la arquitectura.

Para acompañar a la explicación, se usará un ejemplo práctico publicado por William G. Wood en un artículo de 2007 exponiendo un ejemplo práctico sobre ADD [36].

4.4.3.1. Concepto del software

Siguiendo el diagrama del ciclo propuesto, partimos de la primera fase de Scrum, 'Product Backlog'. Como hemos expuesto, en esta primera reunión con el cliente, el equipo de desarrollo debe resolver todas las dudas posibles y elaborar una lista con los objetivos planteados y su organización.

Una vez obtenido toda la información posible del cliente que nos encarga el proyecto, comenzaríamos el sprint dedicado al diseño y, por tanto, iniciamos una primera etapa de concepto de software. Para ello deberíamos analizar diferentes factores que afectan a la elección de una arquitectura software para su inclusión dentro de un entorno ágil y plantear una planificación base para el desarrollo de la arquitectura. Esta se irá desarrollando conforme avance el proyecto y se implante, con tal de mejorar el proceso y encontrar la aplicación perfecta para el uso de la arquitectura. Entre las tareas podemos encontrar :

- Tamaño del equipo, donde lo recomendable sería a baja escala.
- Retroalimentación constante, con la posibilidad de volver a etapas superadas si surgen dificultades.
- La participación de todos los miembros debe ser continua.
- Establecer unas fechas límite pero con la posibilidad de ser flexibles.
- Analizar los tipos de arquitecturas posibles y métodos de calidad de software.
- Precisar la documentación que hará falta.

A partir del 'Product Backlog' habremos obtenido los primeros requisitos, objetivos y riesgos del producto, además de la idea principal del mismo. Por ello, en esta etapa tenemos un primer vistazo del proyecto. En nuestro caso nos encontramos con un típico sistema cliente-servidor que debe cumplir con requisitos básicos como la disponibilidad y la tolerancia a fallos. En un entorno ágil es obligatorio que podamos satisfacer otros requisitos como la escalabilidad, modificabilidad o la adaptabilidad.

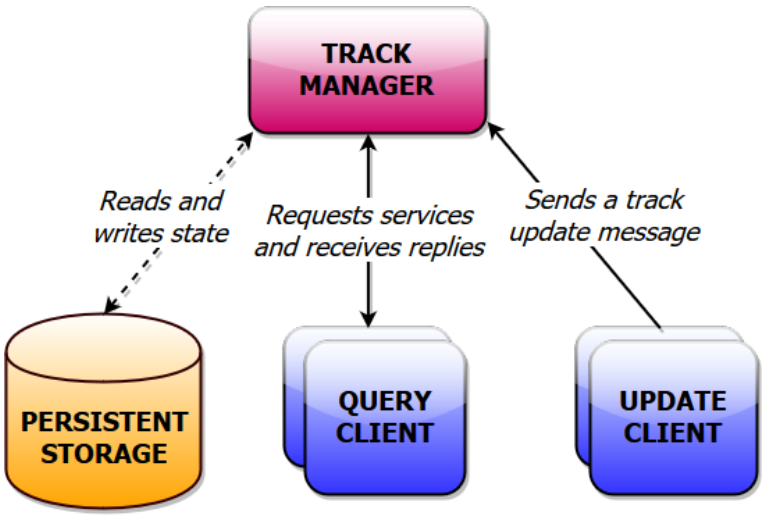

Figura 4.10: Esquema de funcionamiento del ejemplo de ADD. Basado en la figura de la referencia [36]

4.4.3.2. Análisis y definición de requerimientos

La forma de trabajar será siguiendo la propuesta en el libro de Bass denominada como Attribute-Driven Design (ADD) que permite cumplir con los requisitos de calidad solicitados por el cliente de forma recursiva, es decir, descomponiendo el proyecto en módulos sobre los que aplicaremos patrones arquitectónicos. Tenemos por tanto un mecanismo de trabajo bajo una estrategia de mejora continua de la calidad, llamado 'Ciclo de Deming' que consta de tres pasos en nuestro caso:

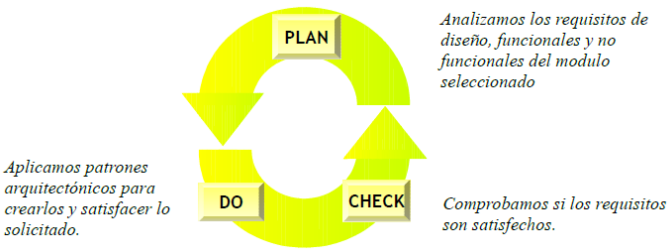

Figura 4.11: Esquema del ciclo de Deming.

Como define Bass, "es un proceso recursivo de descomposición donde, en cada etapa, se eligen tácticas y patrones arquitectónicos para satisfacer un conjunto de escenarios de calidad y luego la funcionalidad se

asigna para crear instancias de los tipos de módulos proporcionados por el patrón." A partir de la división en diferentes submódulos del módulo principal, podríamos asignar estas secciones a diferentes grupos del equipo (al igual que las tareas en cada sprint), obteniendo retroalimentación en cada ciclo.

Las siguientes 2 etapas, que comprenden la definición de los parámetros necesarios para elaborar la metodología, se desarrollan en un continuo bucle asegurándonos que todos los requerimientos están bien formalizados y priorizados.

4.4.3.3. Definición de requerimientos funcionales

En primer lugar determinamos los requisitos funcionales del sistema software. Estos requisitos son declaraciones de los servicios que proveerá, es decir, la reacción del sistema a entradas particulares. Una función es descrita como un conjunto de entradas, comportamientos y salidas. Los requisitos funcionales pueden ser: cálculos, detalles técnicos, manipulación de datos y otras funcionalidades específicas que se supone, un sistema debe cumplir.

Para representar estas funciones podemos utilizar un grafo de dependencias. Lo explicaremos con un sencillo ejemplo. Tenemos cinco funciones de un sistema software simple. Cada vértice se define como una función y están conectados por aristas con una dirección declarada por el tipo de dependencia (saliente o entrante).

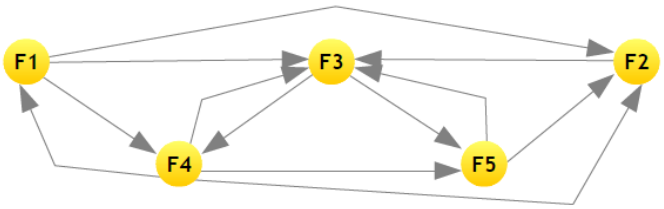

Figura 4.12: Ejemplo de un grafo de dependencias.

Nos fijamos en el proceso del que más dependen el resto, en este caso 'F3' y después de los procesos de los que depende, 'F4' y 'F5'. Al ser 'F5' el proceso con menos dependencias, fijamos este como el primero en prioridad y segundo el ya mencionado. Una vez abarcados, podremos centrarnos en los procesos F1 y F2, tomando como orden el proceso con menos dependencias. Finalmente se desarrollaría el proceso central, F3.

PROCESOS	DEPENDE DE	DEPENDE PARA	PRIORIDAD
F1	3	1	4
F2	1	3	3
F3	2	4	5
F4	4	2	2
F5	2	2	1

Figura 4.13: Tabla de dependencias.

En nuestro ejemplo, el Track Manager proporciona un servicio de seguimiento para dos tipos de clientes:

- **Clientes de actualización:** Los clientes envian actualizaciones de forma periodica al track manager, tolerando este algunas perdidas ocasionales. Los clientes realizan una actualización cada segundo, y el track manager puede recuperarse de dos señales perdidas. Si pierde más de 2 señales, el operador ayuda al track manager en la recuperación.

- **Clientes de consulta:** Su tiempo de operación es esporádico y recibe una respuesta exacta por cada consulta. Los clientes de consulta varian según el tamaño del fragmento que solicitan (ya sea pequeño o grande). El tiempo de respuesta a las consultas debe ser menos del doble del tiempo de respuesta normal para una consulta en particular.

  Usando el grafo y tabla expuestos, una representación simple de su funcionamiento y las prioridades seria el siguiente:

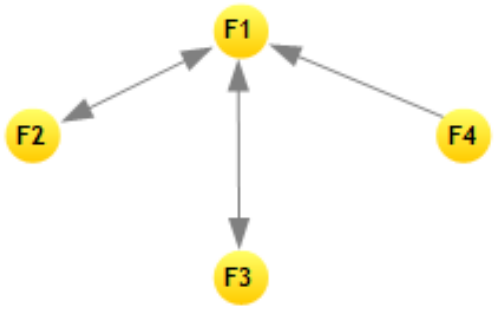

Figura 4.14: Grafo de dependencias del ejemplo.

Siendo F1 asociado al track manager y los restantes, las funciones disponibles.

PROCESOS	DEPENDE DE	DEPENDE PARA	PRIORIDAD
F1	2	3	4
F2	1	1	2 | 3
F3	1	1	2 | 3
F4	1	0	1

Figura 4.15: Tabla de dependencias del ejemplo.

4.4.3.4. Definición de requisitos arquitectónicos

El otro tema a tener en cuenta es la captura de los requisitos arquitectónicos. Un requisito como tal define una condición que el sistema debe cumplir, ya sea impuesto por el cliente o por algún documento externo. Mediante un análisis del contexto podemos extraer consideraciones críticas para el funcionamiento del sistema. La finalidad es conseguir una especificación de la arquitectura, útil para guiarnos en la elaboración de su estructura, que evite ser compleja y que garantice el éxito a ojos de los interesados. Crear la lista de requisitos arquitectónicos conlleva:

- Ayuda a la toma de decisiones a partir de los beneficios y limitaciones de cada alternativa.
- Crea nuevas oportunidades de diseño.
- Describe posibles cambios en la arquitectura seleccionada.

En este punto no necesita una especificación arquitectónica detallada, de hecho, la creación de dicha especificación al comienzo de un proyecto de desarrollo de software es un riesgo muy grande. En cambio, los detalles se identifican en una base de just-in-time (JIT) durante las iteraciones a través del modelado de iteración inicial al comienzo de cada iteración o modelando el asalto a lo largo de la iteración. El resultado final es que la arquitectura emerge con el tiempo en incrementos a partir del refinamiento, más rápido al principio debido a la mayor necesidad de establecer las bases de un proyecto, pero que sigue evolucionando a lo largo del tiempo para reflejar la mayor comprensión y conocimiento del equipo de desarrollo. Esto sigue el modelo de práctica en pequeños incrementos y reduce el riesgo técnico de su proyecto: siempre tiene una base sólida y comprobada para trabajar. En otras palabras, quieres pensar en el futuro pero esperas para actuar.

Para nuestro ejemplo se requieren tres restricciones de diseño:

1. **Restricciones de capacidad:** tenemos 100 clientes de actualización y 25 clientes de consulta. Por tanto, los procesadores tendrán la mitad de su capacidad de procesador adicional y capacidad de memoria en el momento de la entrega y la red local una capacidad adicional del 50.

2. **Servicio de almacenamiento continuo:** el servicio mantendrá una copia del estado actual que el track manager va verificando cada minuto. Si todas las replicas del track manager fallan, se producirá un reinicio desde un punto de control.

3. **Dos replicas:** para cumplir con los requisitos de disponibilidad y confiabilidad, se deben mantener dos réplicas operando en circunstancias normales.

Existen otros requisitos ajenos al funcionamiento principal que se deben cumplir debido a la naturaleza ágil del desarrollo como son:

1. **Facilidad para modificaciones posteriores:** una posible solución será concentrar todo el texto de la aplicación en un archivo de propiedades separado del código que pueda ser fácilmente cambiado y que no requiera de recompilar el sistema

2. **Escalabilidad.**

3. **Reusabilidad.**

4.4.3.5. Selección del módulo a descomponer

Para manejar sistemas de gran escala, ADD propone la descomposición en tareas menos significativas. Por tanto, una vez identificados cada uno de sus componentes se procede a seleccionar uno de ellos para aplicar el método y obtener el diseño del mismo. Posteriormente se evalúa si el diseño resultante es válido para el próximo módulo o si debemos modificar el diseño establecido.

Al tomar como primer paso en este proceso el módulo central, que se consideraría como el sistema en sí, todos los requisitos identificados pasarían a ser globales. Una vez superada esa etapa, las próximas iteraciones dividirán el sistema en distintas partes, las cuáles se asignarán a los miembros del equipo utilizando para ello algún mecanismo de prioridad. Podríamos seguir la siguiente clasificación, siguiendo un orden que favorezca la condición ágil del desarrollo del proyecto, priorizando la organización y elementos de riesgo frente a temas de negocio.

1. **Organización :** todas las condiciones relacionadas con la organización como primera prioridad con el fin de tener más en cuenta la disposición del equipo en ese momento:

   a) La capacidad de desarrollo del módulo frente a la habilidad del equipo.

   b) La disponibilidad de personal para abordar el componente.

   c) Favorecer el aprendizaje a la elección del módulo.

2. **Dificultades y riesgos :** podemos priorizar siguiendo condiciones de complejidad o necesidad:

   a) Módulos cuyos requisitos implican una dificultad mayor o menor.

   b) Riesgos asociados al módulo.

3. **Conocimiento actual :** a través de la experiencia obtenida en módulos anteriores, podemos tener en cuenta algunos factores:

   a) Problemas y facilidades surgidos con módulos similares.

   b) Número de dependencias (al ser mayor, puede ser prioridad respecto al resto).

   c) El nivel adquirido por un equipo facilita la elaboración de un módulo en concreto.

4. **Negocio :** por último, todos los factores relacionados con criterios comerciales:

   a) Su impacto en el tiempo para ser comercializado.

   b) El impacto en los recursos existentes.

   c) El método para conseguir su implementación: construido por el equipo, comprado, licenciado, etc.

   d) Su viabilidad.

Para nuestro caso, se parte de un módulo central definido como el sistema en su conjunto, tomando algunas de estas decisiones:

1. El diseño utiliza un modelo cliente-servidor clásico, brindando el servidor (track manager) servicios a los clientes tanto de actualización como consulta.

2. El Track Manager se ha dividido en dos partes, tomando dos estrategias: una donde ambos elementos operan en un procesador para satisfacer los requisitos de rendimiento y otra donde operan separados.

3. Se ha dispuesto de un mecanismo de comunicación asíncrono para clientes de actualización y síncrono para clientes de consulta.
4. Se almacenan los datos de estado de A y B en el almacenamiento permanente.
5. Se contrata a un experto para tolerancia a fallos.
6. Se organizan los equipos para la siguiente descomposición siguiendo la clasificación anterior.

Esta primera iteración de ADD nos lleva a una descomposición del sistema en los siguientes elementos:

Nº	Elementos	Prioridad
1	Track Manager	4
2	Clientes de consulta	2
3	Clientes de actualización	2
4	Almacenamiento persistente	3
5	Track Manager A	5
6	Track Manager B	5
7	Comunicación sincrónica	3
8	Comunicación asíncrona	3
9	Servicio de nombres	3
10	Servicio de registro	3
11	Servicio de tolerancia a fallos	1

Figura 4.16: Tabla de elementos descompuestos. Basado en la tabla de la referencia [36]

En este caso, priorizamos en primer lugar el servicio dedicado a la tolerancia a fallos como elemento de riesgo. Estos elementos se repartirán entre los equipos por orden de prioridad, dedicando el trabajo exclusivamente al elemento con prioridad exclusiva y otros que compartan prioridad, con varios equipos en paralelo.

4.4.3.6. Selección de atributos de calidad

En la primera de las dos fases que componen la selección de un módulo, determinamos los atributos de calidad o directrices arquitectónicas. Estos son características que se consideran deseables para la arquitectura prevista, siempre en función del contexto en el que nos encontremos (en este caso, un

desarrollo ágil). Para elaborar una lista de atributos de calidad, partimos de directrices arquitectónicas que representarán requerimientos del sistema con exactitud, todos ellos asociados a un atributo en concreto.

La selección de atributos de calidad en un desarrollo ágil para la arquitectura implicará que la elección se oriente hacia terrenos de modificabilidad e integrabilidad. Algunos de los atributos existentes para el diseño pueden ser:

Disponibilidad	Interoperabilidad	Modificabilidad
Rendimiento	Seguridad	Comprobabilidad
Usabilidad	Extensibilidad	Acoplamiento
Encapsulamiento	Reusabilidad	Complejidad
Escalabilidad	Transparencia	Optimización
Correctitud	Adaptabilidad	Fallos
	Cohesión	

Figura 4.17: Tabla de atributos de calidad.

Una vez elaborada una lista de directrices con sus atributos de calidad asociados, podemos llevar a cabo una segunda clasificación en relación a su impacto e importancia. Con esto permitiríamos poder distribuir las tareas entre los miembros del equipo de mayor a menor relevancia, pudiendo organizar futuras revisiones, analizando si las directrices más importantes están completas. Esta clasificación podemos implementarla mediante una sencilla valoración mediante letras, asignando la prioridad como 'Alta' (A), 'Media' (M) y 'Baja' (B) a cada atributo.

Las reuniones en el equipo para coordinar el trabajo podrían provocar que la lista de directrices y sus atributos asociados sea modificada, ya sea añadiendo nuevos candidatos o eliminándolos. Esto también influye en la prioridad, pudiendo cambiar si durante el desarrollo se determina que su relevancia no es tal y como se indicó en su momento. Son cambios propios de una metodología ágil.

A partir de los elementos generados a partir de la descomposición del módulo, cada equipo trabajará con el elemento adjudicado determinando los atributos o escenarios de calidad. Uno de estos elementos es la comunicación asíncrona:

La comunicación asíncrona son aquellas donde la conexión que se establece entre cliente y servidor no es producida en tiempo real, permitiendo la transferencia de datos no síncrona, es decir, los participantes no están conectados en el mismo espacio de tiempo. Por tanto, el cliente podría realizar varias peticiones al servidor sin tener que esperar por la respuesta de

la primera petición. Sus características principales son:

- Es independiente al tiempo.
- Es independiente al lugar.
- La comunicación es individual o grupal.

Su implementación conlleva aumentar la interactividad, la usabilidad e incluso el rendimiento. Teniendo en cuenta las características del elemento, se asume que atributos como la disponibilidad, interoperabilidad, rendimiento o la seguridad son esenciales para un funcionamiento correcto. Para este y el resto de elementos, debemos considerar la modificabilidad o la escalabilidad para este contexto. Una vez listados todos los atributos, clasificar como prioridad los que definen el funcionamiento de la comunicación asíncrona será fundamental, pero tambien los asociados a la idea ágil.

4.4.3.7. Selección del patrón arquitectónico

La selección de atributos de calidad nos da una idea de la estrategia a seguir para un módulo en concreto. Un patrón arquitectónico engloba una serie de tácticas para lograr una calidad especifica, pero cada patrón influye además en otros atributos. Por tanto, debemos llegar a un punto intermedio donde unifiquemos el uso de varios patrones que satisfagan los requisitos establecidos para el módulo y asegurando que los efectos sobre otros atributos se minimicen.

El objetivo en esta fase es discutir la elección de un patrón arquitectónico que cubra las necesidades. Al estar en un marco ágil, esta elección está condicionada por lo que debemos seleccionar un patrón o un conjunto de ellos que favorezcan atributos como la modificabilidad, es decir, principios propios del desarrollo ágil.

En la sección 3.7 elaboramos una lista de patrones ampliamente conocidos, definiendo su uso y los atributos de calidad a los que afecta. Cada grupo del equipo debe elaborar un análisis que determine los patrones arquitectónicos favorables para el módulo elegido teniendo en cuenta el marco ágil. Siendo la metodología de trabajo 'ADD', podemos utilizar varios de los pasos que la definen para adaptarlos a esta metodología ágil. Para esta fase seguiremos 5 etapas secuenciales:

1. Elaborar una lista consensuada entre los miembros con patrones arquitectónicos que aborden las directrices determinadas (atributos de calidad definidos). Cada patrón aportará ventajas para determinados

atributos y desventajas para otros. Se deben listar para realizar una criba posterior.

2. Mediante una tabla, seleccionamos los patrones arquitectónicos más apropiados para cada directriz. Esta clasificación determinará que patrones son los más favorables, posibles combinaciones entre sí, etc. Una vez tengamos los patrones definidos, debemos asegurarnos que la selección favorece las propiedades de un desarrollo ágil. Si el patrón final no posee como atributos de calidad la adaptación, sencillez o la reusabilidad, en etapas posteriores del desarrollo la arquitectura podría convertirse en un cuello de botella para seguir un entorno de desarrollo ágil agradable.

   Es necesario recordar que se desarrolla el sistema de forma ágil, por tanto no se necesita obtener una arquitectura correcta desde el primer día. Es posible que los patrones seleccionados en esta fase no sean los definitivos.

3. Mediante relaciones, elaborar un nuevo patrón a partir de los patrones elegidos. Para ello determinamos como se relacionan los elementos que los componen, descartando el resto. Llegaremos a nuevos elementos como resultado de esta combinación.

4. Crear instancias de los módulos y asignar la funcionalidad desde los casos de uso siendo representado utilizando múltiples vistas.

   Para este punto es necesario modelar: crear uno o más diagramas que presenten una visión general del "paisaje" del módulo. Al igual que un mapa de ruta muestra la organización de una ciudad, el uso de diagramas resumen la organización de su sistema. Los diagramas de navegación son la instanciación de las vistas arquitectónicas del sistema. Muestran cómo organizar y sistematizar las secciones y gracias a ello observamos a la perfección la estructura jerárquica. Debemos tener en cuenta que ningún conjunto de vistas arquitectónicas es adecuado para cada proyecto, si no que la naturaleza del proyecto ayudará a definir los tipos de vistas que debería considerar crear.

   Para crear un diagrama de navegación, el principal impulsor de los esfuerzos de modelado debe ser asumir la simplicidad. La práctica indica que debe existir un esfuerzo por identificar el enfoque de arquitectura más simple posible: cuanto más complicada sea la arquitectura, mayor será la posibilidad de que no sea comprendida por los desarrolladores y llegar a errores. Además, los modelos arquitectónicos deben contener el nivel de información correcto, que muestre cómo varios aspectos del sistema funcionan juntos, pero no los detalles concretos (contenido ágil).

5. Definir interfaces para elementos instanciados.
6. Determinar mediante continuas reuniones si existen inconsistencias en el patrón, si existen controladores que no fueron considerados, etc.

El libro 'Modifiability Tactics' de Felix Bachmann establece una conexión de decisiones de diseño para mejorar uno de los atributos de calidad que es importante dentro del desarrollo de software ágil como es la modificabilidad. Para ello elabora la siguiente tabla:

Pattern	Modifiability									
	Increase Cohesion		Reduce coupling					Defer Binding Time		
	Maintain Semantic Coherence	Abstract Common Services	Use Encapsulation	Use a Wrapper	Restrict Comm. Paths	Use an Intermediary	Raise the Abstraction Level	Use Runtime Registration	Use Start-Up Time Binding	Use Runtime Binding
Layers	x	x	x		x	x	x			
Pipe-and-Filter	x		x		x	x			x	
Blackboard	x	x			x	x	x	x		x
Broker	x	x	x		x	x	x	x		
Model-View-Controller	x		x			x				x
Presentation-Abstraction-Control	x		x			x	x			
Microkernel	x	x	x		x	x				
Reflection	x		x							

Figura 4.18: Tabla de patrones para el atributo de modificabilidad.

Observando la tabla, podemos comprobar que los patrones Broker y Blackboard son los que más técnicas dentro de modificabilidad cumplen. Podría ser un análisis útil para la elección de estos patrones dentro del proyecto de software en un entorno ágil.

4.4.3.8. Diseño de estructuras arquitectónicas

Una vez hemos planificado la arquitectura mediante ADD, el equipo debe ser capaz de elaborar el diseño planteado. Llegados a este punto, deberíamos tener completamente descompuesto el sistema objetivo desde una perspectiva de los atributos de calidad. El modulo central y los distintos submódulos analizados desde esta metodología con el patrón arquitectónico establecido y la documentación necesaria para la implementación.

En esta fase, el rol encargado de controlar todo este primer sprint dedicado al diseño debería comprobar si las tareas asignadas están completas

y llevar a cabo una retroalimentación entre los equipos implicados con los problemas encontrados, soluciones propuestas, etc. Los objetivos para esta fase deben ser:

1. Preparar los equipos necesarios para el diseño con las herramientas previstas.

2. Automatizar todos los procesos posibles.

3. Cumplir con los requisitos conseguidos del paso anterior en la implementación.

Siendo una arquitectura de desarrollo centrada en la calidad gracias a ADD y dentro de un proceso de desarrollo ágil continuo, debemos considerar que existan varias alternativas y mantener esas alternativas "abiertas" siempre que sigan siendo viables. A medida que la comprensión del equipo de los requisitos avance, evolucionarán las visiones arquitectónicas.

4.4.3.9. Evaluación de la arquitectura

A partir de una arquitectura base implementada, debemos evaluarla mediante una serie de medidas que se aplicarán para mejorar el diseño si es necesario, volviendo a la fase anterior. Una evaluación formal de arquitectura de software debe ser un estándar parte de nuestra metodología de arquitectura de software en entornos ágiles. La evaluación de la arquitectura de software es una forma rentable de mitigar los riesgos sustanciales asociados con este artefacto de gran importancia. La capacidad de los atributos de calidad de un sistema de software depende mucho más sobre la arquitectura del software que sobre cuestiones relacionadas con el código tales como elección de idioma, algoritmos, estructuras de datos, etc. Se requiere que la mayoría de los sistemas de software sean modificables y tengan un buen rendimiento, además de cualquier otra propiedad que beneficie el desarrollo ágil.

Entre los principios definidos para la metodología en el punto 5.3, uno de ellos era el uso de 'Software architecture analysis method' (SAAM) como principio básico para la evaluación del diseño resultante. Existen otros mecanismos de evaluación de arquitecturas como los presentes en esta tabla [37]:

MÉTODOS	SAAM	ATAM	CBAM	ALMA	FAAM
Atributos de calidad evaluados	Modificabilidad	Intercambio entre atributo	Costos, beneficios e imprevistos	Modificabilidad	Interoperabilidad y extensibilidad
Descripción del proceso	Racional	Óptimo	Razonable	Razonable	Flujo exhaustivo
Métricas y herramientas	En función del escenario	Identificación de las áreas de alta dificultad. Abierto para cualquier explicación arquitectónica	Tiempo y coste	Estimación del impacto y predicción de la modificabilidad.	Tablas y diagramas

Figura 4.19: Tabla comparativa de mecanismos de evaluación de arquitecturas.

SAAM presenta otras características como:

- Sólo algunas actividades son presentadas como grupales.
- Se identifican atributos de calidad mediante la generación de escenarios.
- Persigue la evaluación de la arquitectura por medio de un atributo.
- Se especifica un rol de revisor o crítico, siendo adjudicado a todos los especialistas.

Sin profundizar en exceso, se propone el uso de SAAM debido a su existencia como método para analizar un sistema en base a su modificabilidad, altamente importante en entornos ágiles. Las principales actividades involucradas en SAAM se enumeran a continuación:

1. Desarrollar escenarios de calidad, compuestos por cualidades, usos y usuarios esperados. Cada elemento descompuesto del módulo central (este incluido) podria caracterizarse como un escenario donde estarían involucrados los miembros de cada escenario en cuestión.
2. Clasificación de los escenarios en directos (sin problemas para recorrerse) e indirectos (requieren modificaciones).
3. Evaluación de los escenarios que presenten problemas mediante una medición de la dificultad para los cambios necesarios, su coste y una descripción que explique estas modificaciones.

4. Comprobar la interacción de escenarios (elementos) a modificar con otros, que pueden llevar a conclusiones como baja cohesión o una alta complejidad estructural.

5. Evaluación del módulo central, con un peso de relevancia para cada escenario y sus interacciones.

Para el caso presentado, si volvemos a la tabla anterior donde se descomponía el módulo central, podemos encontrar lo siguiente:

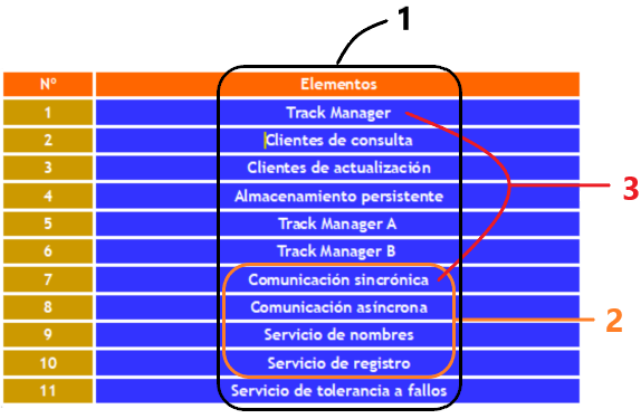

Figura 4.20: Tabla de elementos a evaluar. Basado en la figura de la referencia [36]

1. Los elementos descompuestos del módulo son tratados como escenarios de calidad, de los cuáles definiriamos con más detalle.

2. Dividiriamos los escenarios en directos e indirectos, en función de lo conseguido hasta el momento en los pasos anteriores de definición de atributos y patrones arquitectónicos y su implementación. Los escenarios indirectos, al igual que la medición para valorar su relevancia, se clasificarian a partir de la dificultad para ser modificado.

3. Comprobamos en aquellos escenarios determinados como indirectos sus dependencias. En este caso, podría existir algún problema sobre las comunicaciones síncronas y asíncronas, afectando al track manager.

Por último, se realizaría una valoración en global del módulo central.

4.4.3.10. Alcance y documentación de la arquitectura

Con una arquitectura diseñada y evaluada, necesitamos una documentación útil para poder manejar la suficiente información una vez el proyecto se encuentre en pleno desarrollo dentro de una metodología ágil, ya sea XP o Scrum.

En cualquier entorno de trabajo común, una arquitectura se debe describir con el suficiente detalle sobre un formulario de fácil acceso para todas las partes interesadas, por tanto toda documentación debe tener un uso previsto y audiencia en mente, y ser producido de una manera que sirva para ambos. Esto puede llevar a documentar el diseño de una manera exageradamente extensa, elaborando documentos inútiles u obsoletos.

Por otra parte, como ya se comentó, la lógica imperante en el uso de metodologías ágiles es la de prescindir de la documentación y trabajar sobre la marcha sin textos que aten a las tareas. Este pensamiento sobre proyectos pequeños puede ser viable, pero no sobre grandes producciones. Incluir una fase inicial dedicada a la arquitectura implica elaborar una documentación lo suficientemente completa como para evitar problemas típicos de desconocimiento o descontrol en el proceso. Esto es, crear una documentación ágil.

Un artículo que apareció en la IEEE Software (2009) de Bran Selic, titulado "Agile Documentation, Anyone?", ofrece una visión muy particular sobre este tema:

Frecuentemente escucho a los desarrolladores decir que no les gusta documentar, que no lo encuentran útil, pero... ¿No era el objetivo principal de documentar el ayudar a otros? ¿Cómo es posible una visión tan distorsionada de la documentación? Incluso escucho a los desarrolladores enorgullecerse del "no documentes". Punto de vista que refleja el tan elogiado Manifiesto Ágil, que proclama: "software que funcione, por encima de documentación exhaustiva", como si fueran necesariamente excluyentes entre sí.

El propósito de la documentación es enseñar a quienes no están familiarizados con un sistema cómo este se estructura, funciona y los motivos que llevaron a decidirse por ese diseño. Los principales usuarios de la documentación de diseño son los futuros responsables del mantenimiento del sistema. La única alternativa a no tener documentación de diseño es explorar directamente el sistema, buscar un camino a través de una selva sin mapa ni brújula. Tarea difícil, incluso cuando los autores del diseño están disponibles para hacer de guías. También ineficaz, ya que estos autores tienen que explicar el diseño una y otra vez. Así, mientras que documentar tiene un coste, la inversión, si se hace correctamente, vale la pena.

Pero, sin embargo, en la práctica, la tendencia parece que va en la di-

rección opuesta. La razón que se argumenta es la dificultad de mantener la documentación sincronizada con algo tan dinámico como es el software, y que el coste de documentar se podría utilizar en generar más código ("software que funcione, por encima de documentación exhaustiva"). Por último, hay quienes argumentan que el código es la única documentación, que es el único que contiene la realidad.

La industria del software está repleta de historias sobre viejo código heredado que nadie se atreve a tocar, porque nadie lo entiende. La semántica de los lenguajes de programación está muy lejos de la semántica del dominio de aplicación. La amplitud de esta brecha es la razón por la que todavía se necesitan programadores.

La preferencia del Manifiesto Ágil por el código por encima de la documentación, sólo se opone a una categoría específica de documentación de diseño: los documentos voluminosos y detallados que describen implementaciones, previas a comprender el problema o tener experiencia con la solución. Los documentos de este tipo se vuelven rápidamente obsoletos, y gran parte del esfuerzo invertido en la producción y actualización de ellos se pierde. Como señaló Helmuth von Moltke: "Ningún plan sobrevive al primer contacto con el enemigo."

Sin embargo, la ineficacia de los primeros documentos de diseño, no es argumento para su eliminación. En el desarrollo ágil se llega a la comprensión mediante la experimentación con el sistema software, sin duda una de las maneras más efectivas de aprender. Sin embargo, esta manera de aprender sólo está disponible para los diseñadores que crearon el diseño, y no para aquellos que les seguirán y que no han podido participar directamente en la creación del diseño.

Documentar el software en el código es imprudente, redunda en la pesadilla de intentar mantener la duplicación de información coherente. Necesitamos documentos de diseño a un mayor nivel de abstracción, sin detalles tecnológicos innecesarios y estrechamente asociados a conceptos del dominio y los requisitos. Estos documentos no sólo ayudan a ver el bosque que se oculta en el mar de código, también a documentar los principios básicos del diseño y a servir como el principal medio de comunicación entre los actores del sistema. Con la excepción de los sistemas relativamente pequeños, llevar el diseño de alto nivel en el código (como recomienda, por ejemplo, Extreme Programming) es imprudente.

Debiera haber documentación de diseño ágil, que mantenga y enlace el diseño y la codificación. Y el esfuerzo realizado en documentar debiera ser proporcional al tamaño del diseño.

Los métodos ágiles concuerdan fuertemente en un punto central: "Si la información no es necesaria, no la documente". Como cualquier otro docu-

mento, la documentación que guarde el progreso debe tener un uso y estar dirigido a las personas previstas. Es decir, debemos escribir para el lector y tener en cuenta que no es un proceso fijo. Algunas ideas para su creación pueden ser:

- Partir de un documento base que ayude a explicar la arquitectura propuesta usando todos los esquemas y mecanismos necesarios.
- Especificar las vistas arquitectónicas a producir según los recursos disponibles.
- Priorizar la finalización de todas las secciones restantes. Una vez completadas, será el momento de refinar.

Para el sistema ejemplo propuesto, si descartamos en este momento la documentación posterior al primer sprint que dependerá de la metodología ágil utilizada (Scrum, XP, LSD, etc), todo la información recogida será en exclusiva de la arquitectura. En este caso, cada paso será documentado siguiendo la visión ágil propuesta de informar únicamente de lo necesario. Debemos tener en cuenta los siguientes puntos:

- Todo lo relacionado con el concepto del software se encuentra dentro del sprint, pero no es necesario incluirlo como información. Se añadirá en todo caso en una documentación posterior que englobe el producto.
- Requisitos funcionales y arquitectónicos mediante el uso de diagramas y esquemas que resuman los puntos principales, sin demasiada extensión.
- Breves resumenes de las etapas elaboradas en ADD, tanto del módulo central como de los componentes descompuestos. Para aligerar el proceso, es recomendable documentar cualquier alternativa u opción descartada durante el proceso.
- Detalles sobre la evaluación a partir de SAAM realizada sobre la arquitectura.
- Actualizaciones sobre la documentación si se produce algún tipo de modificación o refinamiento.

4.4.3.11. Refinamiento

El refinamiento como tal implica perfeccionar. No es una etapa posterior a las anteriores, existe durante el desarrollo del proyecto en cualquier momento. El refinamiento arquitectónico ayudará a proporcionar un grado de estabilidad en el transcurso de las iteraciones que se lleven a cabo. El refinamiento arquitectónico es uno de los factores clave para escalar ágilmente con éxito ya que va a permitir describir y mantener el diseño arquitectónico y consigo incluir nuevas características al producto.

En la figura 5.9 podemos comprobar como esta fase se incluye por igual tanto en el sprint propuesto como en Scrum o la metodología ágil que usemos. El desarrollo provocará posiblemente cambios y mejoras en el diseño inicial por circunstancias lógicas como problemas de incompatibilidad o falta de detalle en ciertos elementos. De mano del encargado de actuar como rol de arquitecto, deberá organizar estas etapas de refinamiento en las reuniones diarias y mensuales.

4.4.4. Consideraciones

En sprints consecutivos, el rol de arquitecto de software asegurará que el equipo de Scrum entienda el enfoque y los retos arquitectónicos más importantes en cada sprint. Algunas de las responsabilidades que pueden tener son:

- En reuniones posteriores de cada sprint, el dueño del producto, el encargado del equipo y el arquitecto/arquitectos, priorizan los requisitos a implementar.
- El rol de arquitecto llevará a un consenso las decisiones tomadas en cada entrega de un sprint y la arquitectura, además de facilitar las dudas que surjan, dar orientación y ofrecer coordinación junto al encargado para adaptarse a la arquitectura propuesta en el first sprint.
- Refinamiento continuo del product backlog.

Capítulo 5

Conclusiones finales

Como se ha comentado, el diseño y el desarrollo ágil son dos conceptos que deben ir de la mano, complementandose. La tendencia actual nos lleva a desarrollos que evitan la complejidad en su proceso, con técnicas ágiles que simplifican el trabajo pero que provocan dificultades para alcanzar las cotas de calidad planeadas. El objetivo de integrar el diseño marca un camino a seguir.

Por tanto, a partir de este proyecto se pretende continuar con la ardua tarea de mejorar en todos los aspectos el desarrollo de software, utilizando para ello elementos de arquitectura de software (como los patrones) . Se ha planteado la necesidad de definir un modelo de gobernanza que permita introducir el proceso encargado de la arquitectura de software dentro de una metodología ágil. Para ello, se han tratado los siguientes temas:

- La historia del desarrollo de software hasta nuestros dias, un camino de numerosos años con continuos avances y mejoras.
- Una introducción a conceptos claves del desarrollo ágil, los principios en los que se sustenta, sus ventajas e inconvenientes.
- Un breve repaso por las metodologías ágiles más utilizadas: AUP, XP, Kanban, FDD, LSD, OpenUP y DSDM.
- Con más profundidad la metodología ágil denominada como Scrum, explicando su composición, eventos, etc.
- Todo lo relacionado con las arquitecturas de software, mencionando tales como MVC o la programación por capas.
- Los patrones arquitectónicos y los atributos de calidad asociados.
- Una reflexión sobre la tendencia actual y la necesidad de trabajar en metodologías ágiles con el diseño.

- Y por último una propuesta metodológica para incluir el diseño en una metodología ágil (por ejemplo, Scrum) que consta de:
  - Propuestas existentes.
  - Una exposición de principios que abarquen todos los aspectos sobre como trabajar.
  - Un resumen con los pasos originales de Scrum
  - Propuestas que incluyan un paso previo de diseño y un rol de arquitecto asociado.
  - Una propuesta final denominada 'First Sprint' definida por una serie de pasos.
  - Consideraciones finales tras la etapa de diseño.

5.1. Trabajos futuros

La propuesta metodológica ofrecida en este proyecto podría ser un punto de partida para un desarrollo más extenso. El propósito fundamental era adaptar el diseño sobre una metodología ágil clásica como Scrum, sin embargo, pueden existir numerosas modificaciones en los pasos explicados. Detalles como el tiempo para cada etapa, el orden de los pasos a seguir o la viabilidad del mismo solo es posible refinarlos a partir de su puesta en funcionamiento.

Si bien consolida algunos puntos claves que debe poseer cualquier tipo de etapa dedicada al diseño, puede ser susceptible de mejora a partir de la experiencia adquirida mediante la prueba de la misma. Esto podría ser posible:

- Aplicándolo a casos concretos, desde proyectos de menor a mayor escala para comprobar aspectos como el número de componentes que deben conformar el rol de arquitecto.
- Tests de satisfacción tanto a los miembros del equipo que usen esta metodología como al propio cliente para alcanzar sus pretensiones.
- Realizando comparativas entre la aplicación de la metodología propuesta o no a un proyecto en concreto. Solo así comprobaríamos si los beneficios son tales como la teoría indica.

En resumen, como cualquier otra idea teórica, su aplicación en un proyecto es la forma real de comprobar las posibilidades del método y los refinamientos que sean necesarios.

Glosario

- **TDD:** Desarrollo guiado por pruebas de software, o Test-driven development (TDD) es una práctica de ingeniería de software que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés). En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito
- **Praxis:** Praxis significa acción. Implica emprender una filosofía que difiera de la pura especulación, o de la contemplación.
- **Burn Out:** El síndrome de desgaste profesiona, es un padecimiento que a grandes rasgos consistiría en la presencia de una respuesta prolongada de estrés en el organismo ante los factores estresantes emocionales e interpersonales que se presentan en el trabajo, que incluye fatiga crónica, ineficacia y negación de lo ocurrido.
- **Post-it:** Post-it o pósit es una marca registrada de 3M Company que identifica unas pequeñas hojas de papel autoadhesivo de varias dimensiones, formas y colores, aunque predominan los colores brillantes. Vienen en paquetes de varias hojas pegadas entre sí.
- **Stakeholders:** Una parte interesada o interesados hace referencia a una persona, organización o empresa que tiene interés en una empresa u organización dada. Tal y como fue definida en su primer uso en un memorando interno del Stanford Research Institute, un interesado es un miembro de los grupos sin cuyo apoyo la organización cesaría de existir.
- **Regla 80/20:** Trabajar las funcionalidades principales, completando la anterior antes de pasar a la siguiente.

Bibliografía

[1] "Los 8 inconvenientes de los métodos ágiles".
https://www.obs-edu.com/es/blog-project-management/
metodologias-agiles/los-8-inconvenientes-de-los-metodos-agiles-los-que-deberas-enfre

[2] "Cómo hacer rentable un proyecto". Raúl Alonso, 01/02/18
https://elpais.com/economia/2018/01/25/actualidad/
1516872244_202557.html

[3] "Principios del desarrollo ágil". Joaquín Alviz, 27/10/16
https://www.renacen.com/blog/principios-del-desarrollo-agil-metodologias-agiles/

[4] "Principios del Manifiesto Ágil"
http://agilemanifesto.org/iso/es/principles.html

[5] "Desarrollo rápido de aplicaciones"
http://metodologiarad.weebly.com/

[6] "The New New Product Development Game". Hirotaka TakeuchiI and
kujiro Nonaka, Enero 1986.
https://hbr.org/1986/01/the-new-new-product-development-game

[7] "Metodologias Ágiles existentes". Ervin Flores, 2018
http://ingenieriadesoftware.mex.tl/

[8] "¿Cuáles son los métodos ágiles más utilizados?"
https://www.obs-edu.com/es/blog-project-management/
agile-project-management/cuales-son-los-metodos-agiles-mas-utilizados

[9] "Lean Software Development (LSD): Los siete principios". Ángel M.
Rayo, 27/09/16
https://www.bit.es/knowledge-center/
lean-software-development-lsd-los-siete-principios/

[10] "La guía de Scrum". Ken Schwaber, Jeff Sutherland, Julio 2013
http://www.scrumguides.org/docs/scrumguide/v1/
scrum-guide-es.pdf

[11] "¿Qué es la arquitectura de software?"
https://msdn.microsoft.com/es-es/hh144976.aspx

[12] "El Papel de la Arquitectura de Software en Scrum". Edwin Rafael Mago, Germán Harvey Alférez
https://sg.com.mx/revista/30/el-papel-la-arquitectura-software-scrumx

[13] "Agile Architecture: Strategies for Scaling Agile Development". Scott W. Ambler, 2018
http://agilemodeling.com/essays/agileArchitecture.htm

[14] "An Agile Approach to Software Architecture". Gene Gotimer, 28/06/17
https://www.agileconnection.com/article/agile-approach-software-architecture

[15] "Capturar Requerimientos Arquitectónicos". Antonio Acosta Murillo, 16/06/12
https://es.scribd.com/document/97302200/Capturar-Requerimientos-Arquitectonicos

[16] "El rol de la arquitectura de software en las metodologías ágiles". Quirón, 27/12/08
http://www.epidataconsulting.com/tikiwiki/tiki-read_article.php?articleId=28

[17] "Agile architecture - Where does an architect fit in a Scrum sprint?". Bill Ross, 20/11/15
https://www.equinox.co.nz/blog/agile-architecture-where-architect-fit-scrum-

[18] "ScrumClinic: Sprint 0 e Inception". Jerónimo Palacios Vela, 05/10/16
https://jeronimopalacios.com/2016/10/scrumclinic-sprint-0-e-inception/

[19] "El sprint 0 de Scrum y el diseño inicial". Antonio Martel, 21/06/15
https://www.antoniomartel.com/2015/06/el-sprint-0-de-scrum-y-el-diseno-inicial.html

[20] "Code as Design: Three Essays". Jack W. Reeves, 2005
https://www.developerdotstar.com/mag/articles/PDF/DevDotStar_Reeves_CodeAsDesign.pdf

Artículos y libros utilizados :

[21] Mehdi Mekni, Mounika G, Sandeep C, Gayathri B. Software Architecture Methodology in Agile Environments. 2017, Information Technology

and Software Engineering
https://www.omicsonline.org/open-access/software-architecture-methodology-in-agile-environments-2165-7866-1000195.pdf

[22] Len Bass, Paul Clements, Rick Kazman. Software Architecture in Practice (SEI Series in Software Engineering) 2nd Edition. Addison-Wesley Professional, 2003. ISBN: 9780321815736.

[23] Rob Wojcik, Felix Bachmann, Len Bass, Paul Clements, Paulo Merson, Robert Nord, Bill Wood. Attribute-Driven Design (ADD), Version 2.0. 2006, Software Architecture Technology Initiative
https://resources.sei.cmu.edu/asset_files/TechnicalReport/2006_005_001_14795.pdf

[24] Felix Bachmann, Len Bass, Robert Nord. Modifiability Tactics. 2007, Software Architecture Technology Initiative
https://resources.sei.cmu.edu/asset_files/TechnicalReport/2007_005_001_14858.pdf

[25] Rick Kazman, Len Bass, Gregory Abowd, Mike Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. Department of Computer Science University of Waterloo, Software Engineering Institute Carnegie Mellon University, Texas Instruments Inc.
https://resources.sei.cmu.edu/asset_files/WhitePaper/2007_019_001_29297.pdf

[26] Mirta E. Navarro, Marcelo P. Moren, Juan Aranda, Lorena Parra, José R. Rueda, Juan Cruz Pantano. Selección de Metodologías Ágiles e Integración de Arquitecturas de Software en el Desarrollo de Sistemas de Información. Departamento de Informática - F.C.E.F. y N. - U.N.S.J.
http://sedici.unlp.edu.ar/bitstream/handle/10915/62179/Documento_completo.pdf-PDFA.pdf?sequence=1

[27] M.I. Capel. Desarrollo Utilizando Patrones Software. 2018, ETS Ingenierías Informática y Telecommunicación, Universidad de Granada.

[28] Manuel I. Capel, Anna C. Grimán and Eladio Garví. Calidad Ágil: Patrones de Diseño en un contexto de Desarrollo Dirigido por Pruebas. Departamento de Lenguajes y Sistemas Informáticos (ETS Ingenierías Informática y Telecommunicación, Universidad de Granada), Departamento de Procesos y Sistemas (Universidad Simón Bolívar, Caracas, Venezuela).

[29] Stephan Bode, Matthias Riebisch. Impact Evaluation for Quality-Oriented Architectural Decisions regarding Evolvability. Ilmenau University of Technology.

[30] Neil B. Harrison, Paris Avgeriou. Leveraging Architecture Patterns to Satisfy Quality Attributes. Department of Mathematics and Computing Science, University of Groningen.

[31] Jean Dahl. The Making of an Agile Leader. O'Reilly Media, Inc., 2018. ISBN: 9781492027980.

[32] Moreira, Mario E. The Agile Enterprise Building and Running Agile Organizations. Apress 2017 .ISBN 9781484223918.

[33] Sara van de Geer. Empirical process theory and applications. 2006, Handout WS.
`http://www.stat.math.ethz.ch/~geer/empirical-processes.pdf`

[34] Erich Gamma, Richard Helm. Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley professional computing series). Prentice Hall, 1995. ISBN: 9780201633610.

[35] Ruth Malan, Dana Bredemeyer. Less is more with minimalist architecture. IT Professional, 2002.
`https://ieeexplore.ieee.org/document/1041178/`

[36] William G. Wood. A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0 .Software Architecture Technology Initiative , Febrero 2007.
`http://www.dtic.mil/dtic/tr/fulltext/u2/a468604.pdf`

[37] A. Meiappane, Dr. V. Prasanna Venkatesan, N. Roshini, S.Nivedha, R. Maheswari. Evaluation of Software Architecture by Weight Metric for an Internet Banking System. International Journal of Scientific and Engineering Research, Volume 4, Issue 4. Abril 2013 .
`https://www.ijser.org/researchpaper/Evaluation-of-Software-Architecture-for-an-Internet-Banking-System.pdf`