UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA INDUSTRIAL
ESPECIALIDAD EN ORGANIZACIÓN INDUSTRIAL

PROYECTO FINAL DE CARRERA

IMPLANTACIÓN DE ARQUITECURA DE DESARROLLO ÁGIL DEL SOFTWARE

AUTOR: DAVID CAÑADILLAS MEDINA
TUTOR: MARIA TERESA VICENTE DIEZ

Noviembre de 2010

Agradecimientos

A mis padres, por la confianza que siempre han depositado en mí.

A mi hermano, que sin su apoyo nunca habría sido posible la realización de muchos de mis objetivos.

A Mayte, Jose María, Diego, Gonzalo, Sergio y todos aquellos que en la Universidad siempre me han apoyado y han puesto su granito de arena.

A todos, muchas gracias

The best way to predict the future is to invent it.

Alan Kay

Índice general

Lista de Figuras

Lista de Tablas

Capítulo 1

INTRODUCCIÓN AL PROYECTO

1.1. Introducción

Desarrollar un producto, implementar una solución, desplegar una arquitectura, realizar un servicio... Cada una de estas actividades es un proyecto en sí y por tanto una consecución de fases y tareas en un tiempo determinado. Pero las necesidades de usuarios, clientes y sectores de negocio se han vuelto más exigentes a lo largo del tiempo, tanto en calidad del resultado final como en tiempo de ejecución del proyecto.

Las teorías sobre gestión de proyectos se han preocupado continuamente sobre la mejor forma de planificar, ejecutar y controlar la consecución de proyectos determinados para cumplir su objetivo de una forma fiable y satisfactoria. Estas teorías se han visto mejoradas actualmente con el desarrollo de las tecnologías de la información y del conocimiento de "filosofías de producción" en entornos donde el período de entrega es crítico.

La consecución de tareas de un proyecto se solían definir de manera secuencial, en la que una parte final de esa secuencia realimenta al inicio, mejorando por tanto la consecución de las siguientes fases. Dicho de otra forma, durante la ejecución y seguimiento del proyecto se van a modificar tareas de planificación en función de los resultados obtenidos de tareas anteriores.

Pero toda esta planificación y ejecución de los proyectos suelen estar rígidamente definidas y la capacidad de maniobra en la redefinición de tareas y fases es bastante limitada. Es por esta razón por la que las teorías de producción y gestión de proyectos se han optimizado al máximo y se han obsesionado con la perfecta definición de procedimientos que permitan minimizar los errores de ejecución y previsión.

En el ámbito de la producción y fabricación se aplicaron teorías y "filosofías" que permitieron solucionar todo tipo de problemas en la ejecución de los proyectos en general y en líneas de producción en particular, naciendo definiciones de procedimientos que evolucionaban según la experiencia y el sector sobre el que se aplicaba. Se conocen así términos como *Just In Time*, *Lean Manufacturing*, *SMED*, *Six Sigma*, *5S*, *Kanban*, *CONWIP*, etc. que han sido de ejemplo para todo tipo de proyectos, y extrapolándose a todos los ámbitos y sectores.

El desarrollo de las tecnologías de la información y las herramientas que hacen posible la optimización de proyectos de envergadura ha sido esencial, y se ha llegado a un nivel de precisión tan avanzado que los procesos han mejorado de forma exponencial. Se ha llegado así a tal nivel de calidad en los procesos de una gran mayoría de proyectos que el resultado es por norma general de elevada calidad y fiabilidad, siempre y cuando se haya conseguido aplicar todas estas técnicas y herramientas de forma correcta.

Pero cuando la complejidad y extensión de un proyecto aumentan considerablemente, las herramientas informáticas utilizadas, tales como tipos de ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), HCM (Human Capital Management), Sistemas Operativos, herramientas de cálculo complejo, etc. También se complican de manera muy notable. A veces incluso se complica más el uso de las herramientas que del desarrollo del propio proyecto en sí.

Por esta razón, además de otras muchas, el desarrollo de este tipo de aplicaciones y del software en general se ha convertido en la ejecución de proyectos a veces más complejos que el fin para el que son desarrollados. El desarrollo del software es una tarea crítica para muchos procesos en los que se utiliza.

Cualquier desarrollo de software también es un proyecto, y por tanto aplicable por todas las técnicas y filosofías de la gestión de proyectos nacidas de otros campos y entornos distintos. Durante mucho tiempo se pensó que para gestionar un proyecto de software de envergadura y complejidad elevadas sólo había que utilizar las metodologías y filosofías basadas en los grandes proyectos de ingeniería, llegando así a la calidad, fiabilidad, rendimiento y optimización que se había logrado en la gestión de aquel tipo de proyectos.

La realidad ha resultado ser bien distinta y todos los expertos en la ingeniería del software han aprendido a manejar todos los proyectos de desarrollo con metodologías "superficialmente" basadas en la gestión de proyectos clásica, pero profundamente modificadas en su aplicación de ejecución para conseguir su objetivo.

¿Pero por qué no se pueden aplicar las técnicas utilizadas en otros proyectos y que funcionan tan bien en la gran mayoría de sectores de ingeniería? Muchas son las razones, pero básicamente se puede reducir a la eterna pregunta del desarrollo software: ¿Es arte o ingeniería?

En este proyecto no se entrará en esta discusión ni se pretenderá dar una respuesta universal, pero sí es esta pregunta la clave de cómo cambian las cosas en un proyecto de "ingeniería" de software, entendido éste como el desarrollo de un software específico a través de una consecución de procesos.

Al igual que gran parte de los desarrolladores de software consideran su programación como una obra de arte en la que el artista es único y decide el devenir de su obra (el pro-

ducto final de software), otros tantos ingenieros de software consideran su programación y la del equipo de desarrollo como parte de un proceso medible, adaptable y basado en unos principios, estándares y métricas bien definidos, y por lo tanto gestionable mediante un proceso de ingeniería.

Si bien ya no se va a entrar en la discusión, sí se va a reconocer la validez de las dos posiciones. Efectivamente, la particularidad de todos los proyectos de desarrollo de software radica en la gestión de un proyecto, donde la codificación del programador/ingeniero es única, con identidad y difícilmente reproducible, al igual que cualquier obra de arte, pero también requiere métricas, planificación, gestión de un equipo de desarrollo, gestión de riesgos y criterios de desarrollo que lleven a la consecución de los objetivos del software diseñado, al igual que cualquier proyecto de ingeniería.

Estas particularidades van a generar una forma distinta de enfocar los proyectos de software, y se va a identificar una metodología de trabajo distinta a la del típico proyecto de ingeniería, de forma más dinámica, extremadamente flexible y que puede generar un caos del proyecto, y provocar la imposibilidad de terminar el desarrollo a tiempo, o incluso en algunos casos no poder seguir adelante. Esto es así por las características de la *realimentación continua* dentro de un proyecto de desarrollo de un producto software de envergadura.

Para explicar esto se muestra la diferencia entre las etapas típicas de un proyecto "clásico" de ingeniería y un proyecto de software actual:

- Los proyectos de ingeniería mediante procesos perfectamente definidos, como puede ser una línea de producción de automóviles, una obra de ingeniería civil o el desarrollo de una infraestructura de antenas, por ejemplo, se pueden reducir básicamente a la consecución secuencial de:

    1. Planificación
    2. Ejecución

3. Obtención de resultados
4. Análisis de resultados
5. Modificación de la planificación en tiempo real en la evolución del proyecto, volviendo a realimentar el proceso.

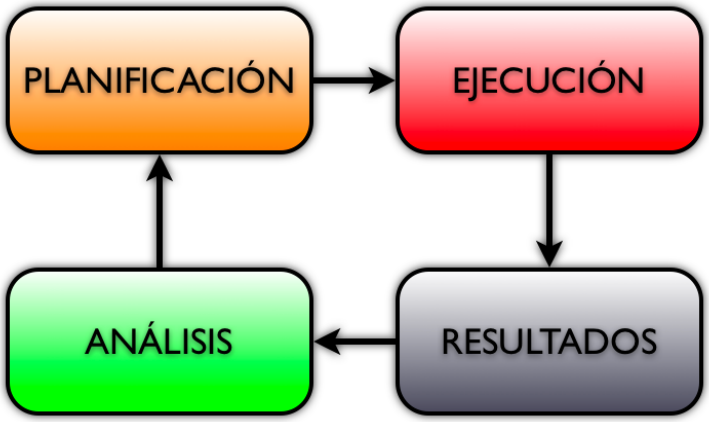

Figura 1.1: Realimentación de etapas en la gestión de un proyecto de ingeniería

Es decir, definida la planificación, se ejecuta tal y como se planifica, se obtienen unos resultados esperados de la ejecución, que se analizan, y reproducen modificaciones en la planificación general, volviendo al proceso de ejecución del proyecto.

- En los proyectos de software se encuentran las mismas etapas. Sin embargo, estas se desarrollan de forma que la realimentación es mucho más compleja y está presente en cada una de ellas. Es decir, en el momento que se escribe la primera línea de código la planificación puede verse ya modificada. O de la misma forma, en el momento que se obtiene un resultado el código puede cambiar o incluso volver a planificar el proyecto.

  Normalmente esto es así por la forma de trabajar de los desarrolladores en un proyecto de software, donde muchas veces el software se diseña mientras se codifica.

  En esta diferencia es donde se comprueba el posible razonamiento del argumento del software como obra de arte, donde el artista diseña y planifica su obra conforme la modela, en todas y cada una de las etapas.

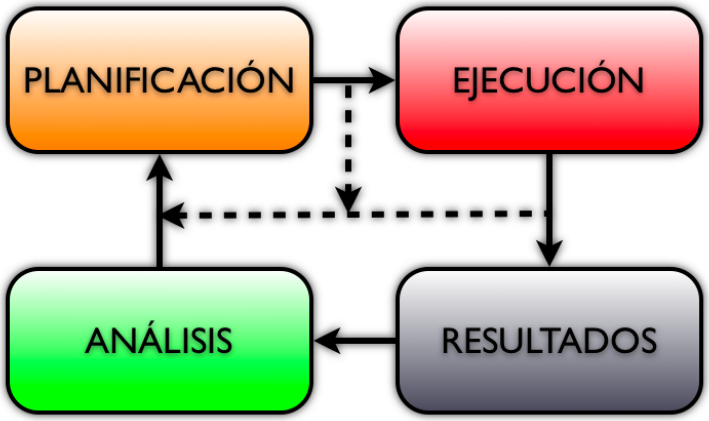

Figura 1.2: Realimentación de etapas en la gestión de un proyecto de software

De forma resumida se puede decir entonces que un proyecto de ingeniería de software no deja de ser similar a un proyecto típico de ingeniería, pero cargado de peculiaridades que harán necesarias modificar las filosofías de gestión de proyectos y de organización del desarrollo. Todo ello para obtener el mismo resultado óptimo, fiable y de calidad que se han conseguido en los procesos de ingeniería actuales.

1.1.1. Desarrollo de ingeniería del software.

Partiendo de la particularidad de estos proyectos de desarrollo, la forma de planificar, ejecutar, controlar y finalizar va a ser un reto en cada nuevo proyecto que se lleve a cabo. Y desde finales de los años 90, donde la ingeniería del software vivió uno de sus mejores momentos impulsada por la "era internet", las metodologías que se utilizan en los proyectos de software tienen sus propias aplicaciones.

Así, de técnicas usadas en proyectos típicos de ingeniería nacieron otras más específicas para el mundo del software, y conocidas la mayoría como Metodologías Ágiles de desarrollo, las cuales están enfocadas a un desarrollo más dinámico en esa realimentación continua de los proyectos de desarrollo de software. Se trata de dar una mayor "agilidad" al proyecto para no entrar en un problema de realimentación infinita y poder salir de ese

bucle sin salida.

Existen numerosas técnicas para gestionar proyectos de desarrollo de software, algunas más específicas y otras más generales, pero en definitiva todas van dirigidas a un desarrollo ágil. Y cómo funcionan muchas de estas metodologías suele ser de forma iterativa, es decir, de forma que se dividen los procesos de desarrollo en pequeñas fases temporales - iteraciones - que irán evolucionando en función de la fase anterior - iteración anterior -.

Para entenderlo de manera más tangible se pensaría en el desarrollo de un herramienta "colaborativa", es decir, una herramienta software que conecte e integre servicios o procesos de distinta naturaleza, pero que todos cumplen una misión común cuando interactúan entre ellos. Esta herramienta necesitará del desarrollo de una arquitectura de software (diseño de plataforma), unas tecnologías de conexión de servicios, un desarrollo de interfaces... Se podría simplificar al desarrollo de una capa de interfaz y una capa arquitectónica.

Lo que ocurre con el desarrollo de la plataforma es que irá evolucionando según se desarrolle la parte del interfaz y viceversa. Es decir, no se puede desarrollar uno y luego el otro, sino que tendrán que ir desarrollándose de forma paralela, pero dependientes entre sí. Esto es así si no se quieren limitar entre ellos. Si se desarrolla el portal de usuario de la herramienta *colaborativa*, la funcionalidad de uso que ofrece dependerá de la plataforma que se desarrolla en paralelo. Pero además, la plataforma tendrá que ser desarrollada en función a las características que vaya a ofrecer el portal de usuario.

Evidentemente, esto es diferente a la fabricación de un puente en un proyecto de ingeniería civil, en el que primero se desarrolla la plataforma (terreno, cimientos, pilares, materiales...) y posteriormente se edifica el resto sobre la plataforma ya diseñada, sin poder cambiarla una vez desarrollada e instalada. No es el caso del software, que necesita técnicas más iterativas y no secuenciales a la hora de su desarrollo.

Todo este desarrollo software se ha vuelto mucho más complejo desde la evolución de Internet, las tecnologías web y las necesidades de las empresas y usuarios de soluciones mucho más complejas.

Al fin y al cabo, de lo que se trata en la gestión de un proyecto de desarrollo de software es llegar a un compromiso con la calidad y el cumplimiento de los objetivos marcados, al igual que cualquier otro proyecto de cualquier otro campo. La calidad, de hecho, es uno de los pilares básicos en la gestión de cualquier proyecto actual, marcado sobre todo por el incremento de exigencia y competitividad de los últimos tiempos.

1.1.2. Optimizar el desarrollo software

Una vez vista y entendida la gran diferencia a la hora de gestionar un proyecto de desarrollo de software, el problema es aplicar las técnicas necesarias en la gestión del proyecto y utilizar unas plataformas de desarrollo adecuadas conforme a la particularidad del proyecto a desarrollar.

El objetivo a la hora de la gestión del proyecto de desarrollo es optimizar lo mejor posible todo el proceso. Esto es, realizar las tareas necesarias con la mayor calidad posible en un período de tiempo establecido, lo cual produce un resultado que cumpla con las especificaciones, las necesidades determinadas de un principio, y con un producto final de calidad excelente.

Pero esto no es fácil y ya se ha visto que las técnicas aplicadas a otro tipo de proyectos no es idealmente aplicable. De aquí nacieron las metodologías ágiles y la Programación Extrema (XP por sus siglas en inglés), que tienen en cuenta la particularidad del desarrollo software y proponen medidas y herramientas específicas para la gestión y desarrollo de un proyecto de ingeniería del software.

En este sentido existen numerosas filosofías para gestionar el proyecto de forma “ágil”

y aplicar las técnicas para desarrollar en fases iterativas durante el proyecto. Y todas ellas, al fin y al cabo, están enfocadas a obtener procesos de calidad que produzca el mejor resultado posible. Las filosofías que tienen actualmente más éxito en la Metodología Ágil son algunas como *Scrum* o *Lean Software Manufacturing*, mezcladas con técnicas específicas de trabajo como la *Programación Extrema*.

Además, en muchos casos la mezcla de filosofías de gestión de proyectos software con gestión típica de proyectos de ingeniería logran un muy buen resultado, como puede ser la mezcla entre una metodología ágil como *Scrum* y técnicas de producción tipo *pull*[1], como son las técnicas *Kanban*.

Estas aplicaciones metodológicas siempre dependen de factores como el tipo de desarrollo, las plataformas utilizadas, el entorno de trabajo, recursos disponibles, el tipo de aplicación final a desarrollar, las necesidades del usuario final, y el objetivo marcado. Y también dependerá en gran parte de la apuesta personal del gestor del proyecto por una filosofía controlable y adecuada para poder gestionarla por su parte. En la actualidad, una figura muy importante en los proyectos de software es quien se encarga de adoptar la metodología de desarrollo y realizar su seguimiento. Esta persona encargada de la metodología puede ser dedicada (es su única función), o puede poseer otro roles, dependiendo de las particularidades del proyecto.

El paso más importante para poder llevar a cabo este tipo de gestión es hacerlo realidad mediante una arquitectura de desarrollo adaptada, que permita desarrollar de la forma más ágil y de calidad posible. En este caso no hay una forma definida y estándar de hacerlo. Cada proyecto de desarrollo gestionado mediante metodologías ágiles está determinado por el uso de unas herramientas y plataformas seleccionadas, pero en muchos casos la gestión del propio proyecto no ha determinado esta selección, y es este paso el que aún no se ha enfocado de forma adecuada y en el que existe actualmente un desarrollo

[1]El cliente final determina cuándo se debe producir una unidad a través de su necesidad de consumo, que es la que “tira"del proceso

por hacer.

Optimizar el desarrollo software consiste entonces no sólo en la aplicación de unas metodologías ágiles adecuadas, sino también en saber adaptar la arquitectura de desarrollo a la propia gestión del proyecto y a las necesidades intrínsecas del producto o servicio a desarrollar, enfocando los procesos hacia la calidad de ejecución, que suele ser la búsqueda de la máxima calidad.

1.1.3. Plataformas y arquitecturas de desarrollo

La importancia de las plataformas de desarrollo en un proyecto de ingeniería de software se demuestra no sólo en las características del producto o servicio a desarrollar, sino también en el desarrollo adecuado del propio proyecto para cumplir los objetivos y obtener la calidad necesaria. Las plataformas de desarrollo ya no son decisivas en las características ofrecidas intrínsecas al objeto del proyecto, sino también al propio proyecto en sí y su entorno de desarrollo.

Durante mucho tiempo la selección de plataformas de desarrollo se tenía en cuenta para las funcionalidades básicas que iba a obtener el usuario final. Características como rendimiento, funcionalidad general, flexibilidad de uso, compatibilidad de versiones y otras relacionadas con el propio producto en sí eran las que finalmente decidían el uso de una plataforma u otra de desarrollo. No se tenía en cuenta el propio entorno de desarrollo a lo largo de la duración del proyecto. Esto ha provocado en ocasiones el retraso continuo y repercusiones en la misma calidad del producto, y hoy día, en un mundo software tan competitivo, estos factores pueden determinar el fracaso o éxito del producto final.

La elección de la plataforma actualmente es vital para el desarrollo de un proyecto de ingeniería software. Además de los factores tecnológicos , que efectivamente son muy importantes, hay que tener en cuenta los propios factores ligados al entorno del proyecto:

- Flexibilidad para el desarrollo ágil

- Integración con los distintos equipos del proyecto
- Capacidad dinámica de actualizaciones
- Fácil acceso a los recursos asignados
- Minimización de configuración durante el desarrollo
- Curvas de aprendizaje optimizadas
- Minimización de la complejidad en las dependencias
- Entornos agradables de trabajo
- Documentación accesible y clara para el desarrollo

Estos y otros muchos factores ligados más la gestión del proyecto que al desarrollo de la tecnología a producir, son factores que se están empezando a tener en cuenta en la elección de la plataforma.

Además, y muy ligado a dicha plataforma, es crítico diseñar una arquitectura que permita sacar el mayor partido de su potencial. Es decir, la propia arquitectura de desarrollo es tan importante como la arquitectura del producto o servicio a desarrollar. Actualmente los diseños de arquitecturas de desarrollo, donde los repositorios de código se orquestan perfectamente con el entorno de los programadores, ingenieros de software y sus herramientas, son los óptimos para gestionar ágilmente un proyecto de software.

Plataformas y arquitecturas de desarrollo tienen que ser altamente dependientes unas de la otras a la hora de gestionar el proyecto. Siempre que un desarrollo de software se traduce en calidad y satisfacción del usuario final, la orquestación de los procesos de desarrollo debe mantener una sincronización perfecta, y para ello la propia arquitectura debe ir muy ligada a las plataformas seleccionadas para el desarrollo y el producto. Es un hecho que durante el tiempo ha mejorado y se ha tenido cada vez más en cuenta.

Aún así, todavía no es algo muy común que durante la planificación de un proyecto se ponga mucho énfasis en la arquitectura de desarrollo del equipo de trabajo, ya que normalmente es algo que va tomando forma según se programa y según las necesidades que se encuentran. Además, en muchas ocasiones esta arquitectura viene determinada o limitada por el uso obligatorio de herramientas según la política de la compañía encargada del desarrollo.

Precisamente con el emergente desarrollo de las metodologías ágiles han ido apareciendo multitud de herramientas que se incorporan a la arquitectura de desarrollo, con el fin de adaptar a la programación ese desarrollo ágil e iterativo exigido por la metodología. Y estas herramientas, se van incorporando al proyecto en función de la necesidad en el momento. Pero esta incorporación sobre la marcha puede provocar problemas y retrasos inesperados en la consecución del proyecto.

Por tanto, la tendencia de las nuevas metodologías y prácticas de desarrollo software se basan en una búsqueda de la calidad en el proceso de programación, la cuál no será óptima sin una búsqueda de unas prácticas adecuadas e iterativas de mejora continua durante la ejecución y desarrollo del proyecto. En este punto la *Integración Continua* [2] en el desarrollo y la realización de pruebas desde el primer momento serán esenciales para un correcto desarrollo.

1.2. Objetivos del proyecto

La finalidad general de este proyecto consiste en lograr definir y construir una arquitectura de desarrollo completamente adaptada a las metodologías ágiles de ingeniería del software, y que permita un desarrollo de código y plataformas enfocadas a la calidad de todos los procesos del proyecto, las cuales repercuten finalmente en el producto o aplicación software a desarrollar.

[2] Concepto definido en el capítulo 2.3

Para poder cumplir el objetivo final es necesario marcar unos objetivos más específicos y que son obligados en el proyecto:

- Encontrar las herramientas necesarias, adecuadas y adaptables para la implantación de la arquitectura
- Diseñar la arquitectura y metodología que definan las líneas de trabajo adecuadas
- Adaptar el desarrollo y el trabajo del equipo a la arquitectura diseñada
- Implantar la arquitectura de calidad que permita obtener unos resultados satisfactorios mediante métricas

Estos objetivos tendrán que adaptarse además a la metodología de trabajo del caso de desarrollo software que se aplicará en un caso de estudio. En este caso, el desarrollo de un ERP específico será el producto software a desarrollar, y el cual está basado en la ejecución de un proyecto mediante Metodología Scrum. Para este fin y tipo de proyecto es esencial precisamente el uso de una arquitectura de calidad del software y desarrollo adaptado a sus necesidades, donde el proyecto que conlleva será determinante.

Este proyecto de implementación de la arquitectura estará también adaptado a este tipo de metodolgías ágiles, y por tanto se emplearán métodos combinados de metodologías ágiles y técnicas clásicas de gestión de proyectos para realizarlo.

Para entender este método de trabajo la cuestión esencial se basa en seguir un método iterativo, pero el cuál se dividirá en fases típicas de proyecto, las cuales determinarán los hitos con objetivos determinados para el caso. Así la metodología iterativa será llevada a cabo mediante Scrum, la cual se definirá en el capítulo 2.1, y las fases contendrán las tareas determinadas en la metodología iterativa.

1.3. Fases y metodología del proyecto

Aunque las fases del proyecto se detallarán más adelante, se verá una pequeña introducción sobre fases y metodología del proyecto que marcan muy claramente los objetivos.

Las fases contendrán tareas basadas en parte sobre la metodología *Scrum*, y para ello se dividirán las fases de trabajo en *sprints*[3], los cuales tienen una duración de dos semanas y contienen unas tareas determinadas a realizar durante ese tiempo. Las tareas a realizar por tanto, dependerán tanto de la fase en la que se esté como del objetivo final del proyecto.

En el capítulo 4, se detallarán las fases del proyecto de implementación y despliegue de la arquitectura en un entorno de desarrollo ágil de software. Aún así, para entender claramente los objetivos marcados se resumen las fases a continuación:

1. **Estudio o diseño de la arquitectura a implementar**
   Una parte muy importante es cómo utilizar adecuadamente las herramientas de la arquitectura para conseguir el resultado perfecto en la posterior consecución de un proyecto de desarrollo. Para este cometido es esencial realizar el diseño adecuado de la arquitectura en función de las herramientas analizadas para el desarrollo en el entorno ágil.

2. **Planificación en el despliegue y configuraciones**
   Una vez diseñada la arquitectura, es necesario desplegar las herramientas de forma correcta y configurarlas, de forma que el equipo de desarrollo y las personas implicadas puedan adaptarse sin problema al desarrollo del proyecto software. Por tanto, habrá que planificar de forma adecuada todas las instalaciones, pruebas, y configuraciones necesarias que se hagan posteriormente.

3. **Implementación de las herramientas**
   Planificada la ejecución de configuraciones e instalaciones habrá que ejecutar. Se deben instalar de la forma planificada todas la herramientas necesarias según el

[3]Todos estos términos vendrán definidos claramente en el capítulo sobre Metodologías Ágiles 2.1

diseño de la arquitectura. En este punto las variables que se tuvieran en cuenta y que afectan al proceso deben mantenerse lo mejor controladas posible. Cuando la implementación esté lista habrá que realizar el despliegue sobre la arquitectura para empezar a funcionar y comprobar el correcto funcionamiento.

4. **Despliegue final**
   Realizada la implementación de la arquitectura hay que desplegar el desarrollo sobre ella y empezar a hacerla funcionar. Conectar las plataformas de desarrollo y calidad de forma adecuada es esencial para que el funcionamiento sea correcto y con la máxima calidad posible de programación.

Estas fases anteriores son inherentes al proceso de ejecución del proyecto, pero además existe otra fase muy importante, pero que no entra en una secuencia tan evidente como las fases anteriores. Además, debido a la adaptabilidad al proceso de desarrollo iterativo, las fases anteriormente mencionadas van a realimentarse entre sí, gracias a la metodología de trabajo. La fase que se añade, por tanto, y que es igualmente importante es:

5. **Seguimiento y control**
   Esta fase es más bien un proceso de control sobre toda la implementación y despliegue del proyecto, es decir, en la fase de ejecución (si se distingue entre las típicas fases de planificación y ejecución de un proyecto cualquiera). Es muy importante tener en cuenta durante todo el desarrollo del proyecto que las cosas funcionan y se miden de forma correcta. Más adelante se detallará toda la parte de seguimiento del proyecto.

Teniendo en cuenta los objetivos, mediante estas fases del proyecto se definirá una arquitectura adaptada a un proyecto de desarrollo de software mediante metodología ágil. Así se podrá determinar una forma de construir un entorno de desarrollo y programación adecuado para este tipo de proyectos.

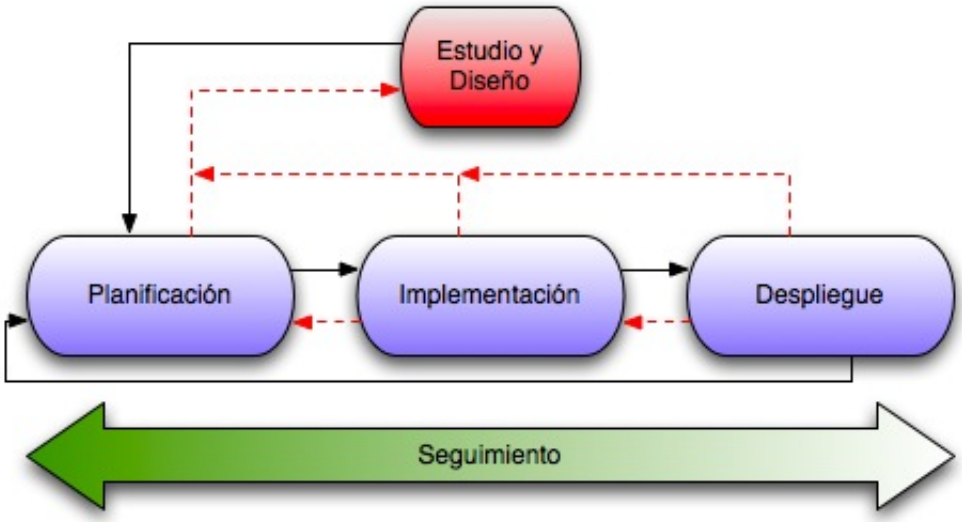

Figura 1.3: Fases del proyecto realimentadas

Gráficamente se puede ver cómo estas fases seguirán una secuencia que se realimenta, empezando desde el diseño y acabando por el despliegue, pero que en todo caso se puede volver a recurrir a tareas de fases anteriores para mejorar el proceso. Es esta la razón donde mezclar la clásica gestión secuencial y la iterativa dará buen resultado.

En la figura 1.3 destacan las flechas continuas negras, que son las que definen la secuencia típica, y las flechas rojas intermitentes, que indican la realimentación a cualquier fase del proyecto. Y en las tres fases azules el seguimiento se produce durante todo el proceso.

En definitiva, en el proyecto se distinguen muy bien las fases, que son la parte secuencial clásica de las tareas, las cuales seguirán un proceso iterativo y que producirán realimentación a otras fases, pero durante la ejecución de fases posteriores.

Capítulo 2

ESTADO DEL ARTE

2.1. Metodologías Ágiles

En la introducción se ha hablado de metodologías ágiles, haciendo referencia a métodos iterativos en la gestión de proyectos de software. Pero las implicaciones y definiciones de estas metodologías van mucho más allá y es necesario un conocimiento específico de este tipo de metodologías.

Normalmente cuando se habla de *agilidad* en la ingeniería del software y en la gestión de sus proyectos surgen preguntas:

- ¿Qué es la agilidad?
- ¿Por qué metodologías ágiles?
- ¿Qué aporta la agilidad?
- ¿Cómo llevar a cabo un proyecto de manera ágil?
- ¿Es la solución a mis problemas de proyectos software?
- ...

Estas y otras muchas preguntas que vienen a la cabeza cuando se habla de metodologías ágiles tienen sus respuestas. Pero lo mejor es definir de manera ordenada este tipo

de metodologías y entonces encontrar respuestas y maneras de aplicarlas a los casos de desarrollo.

Aún así, la primera pregunta tiene una respuesta muy clara, y existen definiciones específicas. Una de ellas y que aclara a la perfección el término *agilidad* se encuentra en la definición [12]:

> "Agilidad es la habilidad tanto de crear cambios como de responder a ellos obteniendo beneficios en un entorno empresarial turbulento".

Se puede decir entonces que la agilidad en un entorno de ingeniería de software proporciona una manera más efectiva de actuar respecto a los continuos cambios existentes en las tecnologías de la información y el software en general. Es evidente que en un entorno empresarial en el que en ocasiones su tecnología evoluciona en períodos muy cortos de tiempo (a veces incluso semanas), necesita de metodologías de producción capaces de adaptarse a un entorno tan cambiante.

Cuando se habla de crear y responder a los cambios en un mismo contexto de producción o generación de un producto, es inevitable pensar en acciones realimentadas, es decir, iterativas. Todo el desarrollo empleado en producir está en progreso y es algo continuamente cambiante.

Pero hay que decir que no siempre las técnicas iterativas proporcionan agilidad a los proyectos de software, sino que muchas veces pueden jugar un papel en contra. Existe el peligro de que en ciertos proyectos con determinadas características ese bucle iterativo no encuentre salida, y entre en una espiral de desarrollo que no tenga fin. Esto suele ocurrir en proyectos de envergadura donde los recursos y variables son muy poco controlables.

Es por tanto de importancia determinar las características del proyecto para evaluar la forma y posibilidad, tanto de adaptar el desarrollo a una metodología ágil como de

adaptar la metodología ágil al entorno de desarrollo. Estas características vendrán determinadas por variables del proyecto, de las cuales algunas serán más flexibles y otras más rígidas.

Para aplicar una metodología ágil lo primero que se debe hacer es analizar las características del proyecto y encontrar el verdadero sentido de la aplicación de dicha metodología, comprobando que efectivamente cumpla con el objetivo y haga más dinámica enormemente la consecución del proyecto. Así, las variables que se deben analizar están dentro de los siguientes entornos:

- El equipo de desarrollo
- Gestión del proyecto
- El cliente final
- Procesos y herramientas
- Contrato

Estos cinco entornos son las categorías de las variables que definen las características de un proyecto, las cuales serán distintas dependiendo si son favorables para un entorno ágil o no. Estas características englobadas en los entornos anteriores se pueden ver de la forma representada en la tabla 2.1.

Así se puede ver que un proyecto se considerará "ágil" cuando se encuentren las características mencionadas. Lo que implica es que todo proyecto donde la colaboración, retroalimentación, accesibilidad a la información, visibilidad e implicación del equipo sean decisivos, la aplicación de una metodología ágil es una mayor garantía de éxito, ya que sus métodos van dirigidos a este tipo de proyectos.

De todas formas, no todos los proyectos son idealmente ágiles, ya que por definición de la tarea a desarrollar, por requisitos o rigidez establecida por el usuario final (o cliente)

Tabla 2.1: Características de un proyecto

Entorno de proyecto		**Características del proyecto**	
Categoría	**Variable**	**Ágil**	**No ágil**
Equipo de Desarrollo	**Estilo de comunicación**	Colaboración regular	Sólo lo necesario
	Localización	Centralizado	Distribuido
	Tamaño	<50 Personas	>50 Personas
	Aprendizaje continuo	Activo	Olvidado
Gestión del Proyecto	**Cultura de gestión**	Dinámica	Ejecución y control
	Participación	Obligatoria	Rechazada
	Planificación	Continua	Al inicio
	Retroalimentación	Varios mecanismos	Inexistente
El Cliente	**Implicación**	A través del proyecto	Durante fase de análisis
	Disponibilidad	Fácilmente accesible	Difícil
Procesos y herramientas	**Input del equipo**	Tienen la última palabra	Usan lo indicado
	Cantidad	Las necesarias	Más de lo necesario
	Adaptabilidad	Puede cambiar	Tal vez no cambie
El Contrato	**Fechas y requisitos**	Flexible	Rígido
	Coste	Según recursos	Fijo

marca la imposibilidad de adaptar cierta agilidad al proyecto.

Precisamente en los proyectos de software se encuentran en muchas ocasiones estas características. Y en el caso de no ser así, normalmente se intenta modificar las características del proyecto, de forma que se adapte a una metodología lo más ágil posible. Por ejemplo, en equipos de desarrollo de software donde la comunicación entre los miembros es escasa por su tipología, se puede intentar establecer un entorno lo más colaborador posible para *agilizar* el proyecto.

Además, la experiencia indica que precisamente las características de un proyecto de software suelen ser más de tipo "no ágil", debido a que normalmente se encuentra con:

1. Desarrolladores descentralizados por normas departamentales
2. Gestión de proyectos jerarquizada, dividida en el tiempo y planificada al inicio
3. Cliente con disponibilidad muy limitada y escasa implicación
4. Rigidez de procedimientos y herramientas por parte de la empresa y el cliente
5. Contratos muy rígidos, incluso en los tipos *Time & Material* (basados en recursos).

Estas y otras características suelen aparecer con mucha frecuencia en proyectos de software, y muy especialmente en proyectos de desarrollo de software a medida, donde el cliente, a pesar de estar en continua comunicación y seguimiento del producto final, “rigidiza” enormemente la consecución del proyecto.

Pero en los casos reales precisamente de lo que tratan las metodologías ágiles no es de intentar cambiar factores incontrolables y de difícil cambio, sino de adaptar esas característica típicas de entornos rígidos a un desarrollo de proyecto donde la respuesta a los cambios sea lo más rápida posible. Se puede entender de forma que existen proyectos donde sus características intrínsecas no permiten tratarlo como un proyecto ágil por sí

mismo, pero sí determinar una metodología que promueva la rapidez de acción ante cambios inesperados, induciendo así cierta *agilidad* a su propia rigidez.

En definitiva, la metodología ágil como procesamiento de métodos de gestión y desarrollo no es más que la forma de adaptar las variables de entorno del proyecto de desarrollo (definidas en la anterior tabla 2.1) a las características de un proyecto ágil. Para ello la mayoría de metodologías ágiles tienen prácticas en común:

- Tests del desarrollo
- Integración Continua
- Scrum [1] diario.

Estas prácticas engloban tanto el propio desarrollo del software como la gestión del proyecto. Así, las metodologías ágiles están centradas en la parte de desarrollo software o en la gestión de los procesos del proyecto, según la distinta metodología aplicada. Se pueden mezclar incluso la aplicación de distintas metodologías con el fin de agilizar todo el proceso, desde la gestión hasta la programación de código del software.

Pero todas las metodologías ágiles están basadas en los doce principios marcados por el denominado *Manifiesto Ágil*[2]:

1. La principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor
2. Los requisitos cambiantes serán bienvenidos, incluso incluso incorporados tarde al desarrollo. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente

[1] Referido a reuniones diarias de la metodología Scrum, la cual se describe más adelante en este capítulo
[2] En inglés *Agile Manifesto for Software Development* [1]

3. Entregar con frecuencia software que funcione, en periodos de un par de semanas a un par de meses, con preferencia de periodos de tiempo cortos

4. Las personas de negocios y los desarrolladores deben trabajar juntos diariamente a través del proyecto.

5. Elaborar proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realizar su trabajo

6. La forma más eficiente y efectiva de comunicar información al equipo de desarrollo y entre e propio equipo es mediante la conversación cara a cara

7. El software que funciona es la principal métrica de progreso

8. Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida.

9. La atención continua a la excelencia técnica enaltece la agilidad

10. La simplicidad como arte de maximizar la cantidad de trabajo no acabado, es esencial

11. Las mejores arquitecturas, requisitos y diseños emergen de equipos que se autoorganizan

12. En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia

Estos doce principios marcan las líneas de trabajo hacia las que se dirigen las metodologías, tanto desde el punto de vista del desarrollo como desde de la gestión. Así, por ejemplo, se encuentran distintas metodologías ágiles aplicadas en distintos ámbitos del proyecto: gestión y ejecución. Como ejemplos se pueden citar:

- Adaptative Software Development (ASD)
- Feature Driven Development (FDD)

- Extreme Programming (XP)
- Scrum
- Agile Unified Process
- Dinamic Systems Development Method (DSDM)
- Essential Unified Process (EssUP)
- Lean Software Development
- The Crystal Methodologies

La mayoría de estas metodologías de nombre inglés provienen de Estados Unidos y que la experiencia durante años de grandes y pequeñas compañías de software ha ayudado a desarrollar. Sus procedimientos y teorías están elaborados a partir de la propia experiencia en la aplicación de los métodos clásicos de gestión de proyectos en determinados proyectos de software.

Además del punto de vista metodológico existen ciertas prácticas dirigidas a la programación ágil, en las cuales no se va a centrar este estudio, pero que detallan a más bajo nivel técnicas de desarrollo de software de manera ágil. Pero todas las metodologías y todas las prácticas tienen en común un objetivo principal, y es responder eficaz y rápidamente a los cambios obteniendo resultados positivos de ello.

Cada una de estas metodologías ágiles tiene sus peculiaridades y se aplican en función de las características del proyecto. Una de las más conocidas y extendidas en los últimos años es la metodología *Scrum*.

Metodología Scrum

Aunque por norma general las metodologías ágiles suelen asociarse a métodos iterativos de desarrollo, la realidad es que no todas las metodologías ágiles se centran en las

iteraciones como eje del proyecto. Sin embargo, la metodología ágil Scrum toma el desarrollo en iteraciones como la herramienta principal de trabajo para lograr un producto final de alta calidad y bien ejecutado. Scrum

La metodología Scrum se emplea con el fin de obtener un producto de software de alta calidad de programación con un coste lo más reducido posible, logrando una optimización del desarrollo que maximiza el retorno de la inversión del producto de software. Así, el método iterativo en Scrum intenta tener una aplicación funcional desde el momento inicial, iterando sobre ese producto en versión muy prematura e ir añadiendo funcionalidad en las iteraciones durante el desarrollo del proyecto.

Esta metodología está siendo ampliamente aplicada en el sector de las Tecnologías de la Información, debido en gran parte a su aplicación desde el punto de vista metodológico de la gestión y no de los procedimientos de desarrollo software. Es una de aquellas metodologías ágiles que se centran en la gestión de proyectos. Por sus características es también una metodología muy flexible para emplear técnicas de desarrollo ágil muy variadas.

Scrum es por tanto un método de *Desarrollo Iterativo e Incremental* [3] más enfocado a los valores de la gestión de proyectos y prácticas que en los requerimientos e implementación. Una de las razones por las que combina tan bien con otras técnicas o metodologías.

Básicamente, la metodología Scrum se basa en tres pilares [19]: transparencia, donde la comunicación y la visibilidad son esenciales; la inspección, dirigida a una calidad continua del proceso; y la adaptación, la cual se basa en esa agilidad y flexibilidad respecto a los cambios de un desarrollo ágil.

En la metodología Scrum, el cliente final, coincidiendo en numerosas ocasiones con el usuario, tiene un gran peso en el desarrollo del proyecto. Esta es una de las razones

[3] IID, *Iterative and Incremental Development*, característica principal de las metodologías ágiles y parte esencial en muchas de ellas

por las que el empleo de esta metodología es muy atractiva para proyectos a medida, que "engancha" al cliente en el proyecto y le convierte en un recurso más del desarrollo del propio proyecto de software. De esta forma uno de los mayores escollos de toda gestión de proyectos de software, que suele ser la no muy engrasada relación entre el cliente y proveedor, se convierte en una ventaja al eliminar las barreras que suelen existir: mala comunicación, falta de información, insuficiente captura de requisitos, etc.

¿Cómo se ejecuta entonces un proyecto en una metodología Scrum? Pues lo importante en esta metodología es sobre todo el seguimiento continuo del trabajo, bajo un método bien definido mediante roles, productos funcionales, y unas reuniones determinadas. Todo tiene que encajar en estas definiciones y tiene que seguir una rigidez metodológica. Entonces, en este desarrollo iterativo, cada iteración, denominada *Sprint*, tendrá asignadas unas funcionalidades a desarrollar que establecerán el seguimiento de la ejecución del proyecto.

Artefactos Scrum

Se encuentran los siguientes documentos del registro de esta metodología:

- *Product Backlog*: Lista de tareas y requerimientos a nivel funcional (alto nivel)
- *Sprint Backlog*: Lista de tareas y requerimientos de la iteración (bajo nivel)
- *Burn Down Chart*: Gráfico de seguimiento del trabajo pendiente.

De todas las funcionalidades (*features*) definidas en el *Backlog* de producto se escogerán un grupo de ellas, de forma que sean realizables en las iteraciones determinadas, formando así el *Sprint Backlog*. Éste tendrá que estar lo suficientemente bien definido para terminar de desarrollar las tareas y funcionalidades establecidas en el *Sprint Backlog*.

Todos los sprints o iteraciones deben durar no menos de una semana y no más de cuatro, para así no estar trabajando en las mismas funcionalidades más de un determi-

nado tiempo que puede llevar a la monotonía e improductividad. Se podría asimilar a la división del proyecto en múltiples hitos que proporcionan una carga de trabajo constante y productiva.

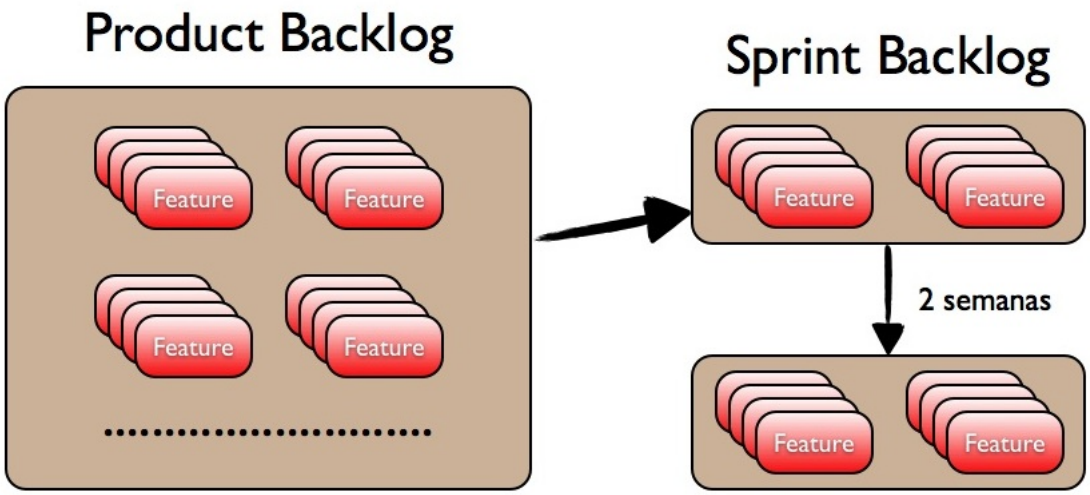

Figura 2.1: Backlog y Sprint Backlog

Roles Scrum

Los roles de la metodología Scrum se dividen básicamente en dos, denominados mediante la analogía de *Cerdos y Gallinas*. La función de cada rol se entiende perfectamente mediante la tira de cómic de la referencia 2.2.

Figura 2.2: El Cerdo y la Gallina. [2]

Es decir, el rol "cerdo" es aquel cuyos personajes están completamente involucrados

en el proyecto internamente (comprometidos), y las "gallinas" son los que su implicación es más externa pero necesaria a la vez.

Los roles "cerdo" en Scrum son:

- El equipo. El equipo de desarrollo es quien se encarga de "fabricar" el producto y conseguir cumplir con los objetivos marcados. Los equipos de desarrollo en esta metodología deben ser pequeños, de no más de 7 ó 9 personas.
- *Scrum Master*. Es una de las figuras más importantes en la metodología Scrum, la cual se encarga de llevar a cabo la metodología con los medios necesarios, llevar el seguimiento de la metodología, y facilitar a todos los actores del proceso su trabajo para cumplir los objetivos en todas las iteraciones.
- *Product Owner*. Es el encargado de representar al cliente del producto software final y su cometido se centra en asegurarse de que los requisitos necesarios para el desarrollo son los adecuados. En esta línea se encarga de que el equipo Scrum realice el trabajo pedido por el cliente y también marca las prioridades requeridas.

Normalmente el *Product Owner* suele involucrarse dentro del equipo Scrum y tener una actuación activa, por lo que se considera dentro del rol "cerdo". En el caso de un *product owner* con poca implicación se puede considerar entonces dentro del rol "gallina".

Los actores pertenecientes al rol "gallina" son:

- Clientes y Proveedores, denominados *Stakeholders*. Son los beneficiados del resultado final proyecto y quienes inician también el proceso. Pertenecen al rol gallina porque su implicación se limita al seguimiento e implicación en las revisiones de las iteraciones o *sprints*.
- Los gestores o *Managers*. Se encargan de proporcionar lo necesario en el entorno de desarrollo para llevar a cabo el trabajo.

Todos y cada uno de estos personajes juegan un papel importante en el proceso metodológico de Scrum y es esencial que cada uno asuma su papel y cumpla los objetivos marcados.

Reuniones

En la metodología Scrum el seguimiento y la comunicación de las distintas partes pertenecientes a cada rol es esencial como parte de un proceso ágil de desarrollo. Para llevar a cabo el trabajo de desarrollo software en cada iteración se deben tener unas reuniones que marcan la ejecución y seguimiento del proyecto a la vez. Las reuniones típicas de Scrum son las siguientes:

- Reunión de Planificación de Versiones (*Release Planning Meeting*). En esta reunión se establecen los objetivos y el plan a seguir y se comunica al equipo y a los participantes. Suele ser una reunión inicial donde se marcan las pautas de trabajo. Esta reunión es opcional y no suele ser típica en todos los procesos Scrum.

- Planificación de Sprint (*Sprint Planning Meeting*). Al principio de cada iteración o *sprint* se planifican las tareas y detalles a desarrollar para la funcionalidad determinada.

- Revisión de Sprint (*Sprint Review*). Al final de cada *sprint* se reúnen todos los pertenecientes al rol *cerdo*, e idealmente los *stakeholders*, para revisar que el trabajo durante la iteración se ha cumplido. Así comunicar problemas, incidencias o inquietudes son esenciales para retroalimentar el proceso de desarrollo. Estas reuniones suelen tener una duración de cuatro horas, dependiendo de la duración del *sprint*, pero nunca debe ser más del 5 % del total, y tienen características de grupos de trabajo.

- Restrospectiva de Sprint (*Sprint Retrospective*). Tras la reunión de revisión de *sprint* y antes de la reunión del siguiente *sprint* se hace una restrospectiva para que el

Scrum Master revise que el proceso ágil cumple su cometido y obtiene información para optimizarlo y hacerlo más eficaz durante el desarrollo de la metodología.

- Reunión diaria Scrum (*Scrum Daily Meeting*). Todos los días durante cada iteración el equipo se reúne durante 15 minutos para revisar su trabajo continuo, exponer problemáticas, y afrontar nuevas ideas que ayuden a su trabajo diario. El *Scrum Master* suele estar presente en estas reuniones, recibiendo información del proceso y ayudando al equipo en cuanto a metodología se refiere.

Básicamente estos son todos los elementos de la metodología Scrum, donde el proceso iterativo y su funcionamiento contempla en la figura 2.3, con sus respectivas planificaciones.

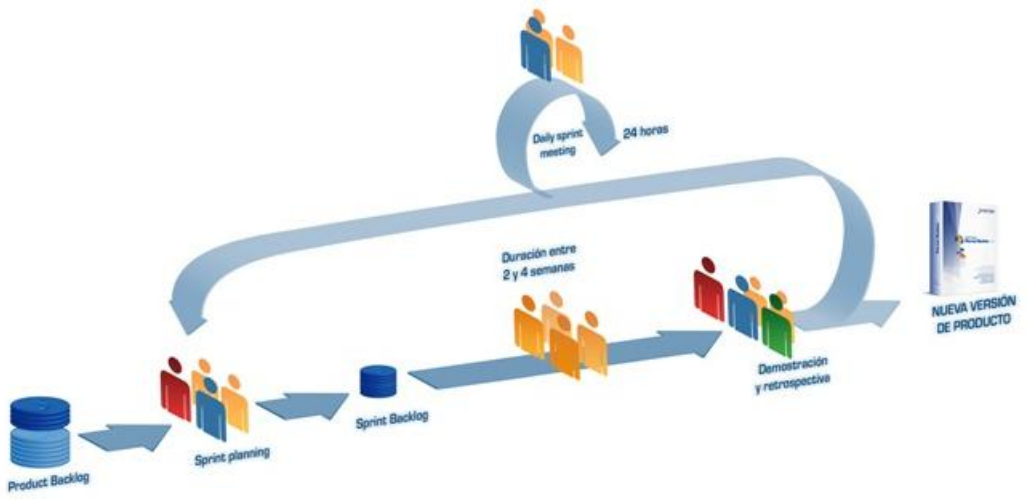

Figura 2.3: El proceso iterativo Scrum. Fuente: http://www.softeng.es

Se comprueba efectivamente que el proceso Scrum es una metodología completamente dirigida a la gestión del proyecto y que es muy flexible para trabajar junto a otras metodologías de trabajo. Es muy común trabajar con metodología Scrum aplicada al *Extreme Programming* (XP), otra metodología distinta, pero centrada en herramientas de trabajo y desarrollo, y no tanto en la gestión.

2.2. Calidad del desarrollo software

En todos los manuales y libros de programación siempre se presta atención a aspectos que marcan al buen programador de software. Siempre se menciona que el buen programador no es siempre el que consigue abstraer el software al menor número de líneas de código o aquel que consigue programar de forma que sólo el propio programador puede entender.

Durante décadas se ha difundido que un buen código de software es aquel que está bien documentado, estructurado y cumple su función de la forma más óptima posible, tanto en cuanto a rendimiento como en funcionalidad. Es decir, la métrica de un buen código no son el número de líneas, sino diversos factores a tener en cuenta y no tan triviales de obtener.

En todo el tiempo de evolución de la computación y el software se han ido añadiendo técnicas, lenguajes y herramientas que consiguen sacar el máximo rendimiento del desarrollo software con el fin de cumplir unos objetivos para los que se desarrolla o programa. Y durante todo ese tiempo cada vez ha ido tomando más peso lo que se considera como calidad del desarrollo, del código y del software en general. El usuario final es cada vez más exigente y busca software de calidad.

Para ello, al igual que en la fabricación industrial, la calidad es una parte esencial en el producto, desde la tarea más básica hasta la más compleja. Así, las técnicas de "Aseguramiento de la Calidad" (*Quality Assurance* ó QA) se han adaptado también al mundo del software y hoy día casi toda empresa dedicada al desarrollo de software tiene un departamento propio de QA, dedicado exclusivamente a la calidad del desarrollo del producto software y que afecta a todo el proceso de desarrollo, y no sólo al producto final.

Actualmente los objetivos en un proyecto de desarrollo de software es ofrecer un producto donde la calidad sea una característica intrínseca al producto. Desde este punto de

vista la calidad no debería repercutir enormemente al coste, ni a ningún otro parámetro en el desarrollo de un proyecto software. La calidad de desarrollo de un producto software debe cubrir:

1. Funcionalidad. El aumento de funcionalidad del producto no debe afectar a la calidad de ejecución.

2. Rendimiento. Mayor calidad no debe sacrificar el rendimiento de la aplicación, entendido este como relación de la rapidez de ejecución con la potencia de procesamiento.

3. Estabilidad del programa. Un software de calidad no debe inducir comportamientos inesperados o "cuelgues" del sistema.

4. Robustez. Un programa de calidad es un programa robusto, por lo que exigir potencia de uso no debe significar pérdida de rendimiento.

Estas características son necesidades típicas de los usuarios de software, las cuales sin un aseguramiento de la calidad adecuado son difíciles de alcanzar sin sacrificarse mutuamente.

Pero lo que diferencia el mundo del software del mundo de la fabricación "física" es precisamente que el concepto de calidad va más dirigida a su aseguramiento que a su control. Es decir, se trata de mejorar continuamente el desarrollo y de evitar todos los defectos posibles, más que detectar los defectos y actuar en consecuencia, que es de lo que se encarga el control de calidad. Aunque también en la fabricación industrial se busca un aseguramiento de la calidad, el control de calidad también es igualmente importante.

Para lograr un aseguramiento de la calidad es esencial tener como objetivo el lema *más vale prevenir que curar*. Para ellos hay que entender qué es un producto de calidad y qué se consideran fallos o errores. Aquí, el punto de vista del usuario y del fabricante pueden ser distintos, pero hay que tener en cuenta los dos.

Como se observa en la tabla 2.2, aspectos como la facilidad de manejo, instalación, documentación o el rendimiento, son percepciones directas por parte del usuario final

Tabla 2.2: Puntos de vista de la calidad del software.

Punto de Vista	Atributo
Fabricante	**Diseño**
	Tamaño
	Cambios
	Complejidad
Cliente o usuario final	**"Usabilidad"**
	Manejo
	Portabilidad
	Rendiemiento
	Instalación
	Documentación

donde la calidad juega un papel decisivo, mientras que desde el punto de vista de quien desarrolla el software los aspectos más controlables y relacionados directamente son otros: diseño, tamaño, complejidad, etc.

El aseguramiento de la calidad trata de evitar fallos y maximizar la optimización desde ambos puntos de vista. Es muy importante que quien desarrolla el software no pierda el foco de que el producto debe cumplir con las necesidades del usuario y satisfacer sus necesidades al máximo nivel. Para esto es necesario que la calidad sea máxima en todas sus fases de desarrollo del proyecto.

En el proceso de desarrollo de una aplicación de usuario se parten de unas necesidades del cliente o usuario con el siguiente desarrollo de programación, el cual debe obtener una aplicación de manejo intuitivo y agradable (*User Friendly*[4]). A través de este manejo

[4] *Amigable para el usuario*, referido normalmente en software a todas aquellas aplicaciones fáciles de manejar, entender y aceptar por los usuarios

se obtiene información útil. Analizando los resultados, éstos devuelven más información sobre las necesidades del usuario, volviendo a aplicar al desarrollo modificaciones del producto.

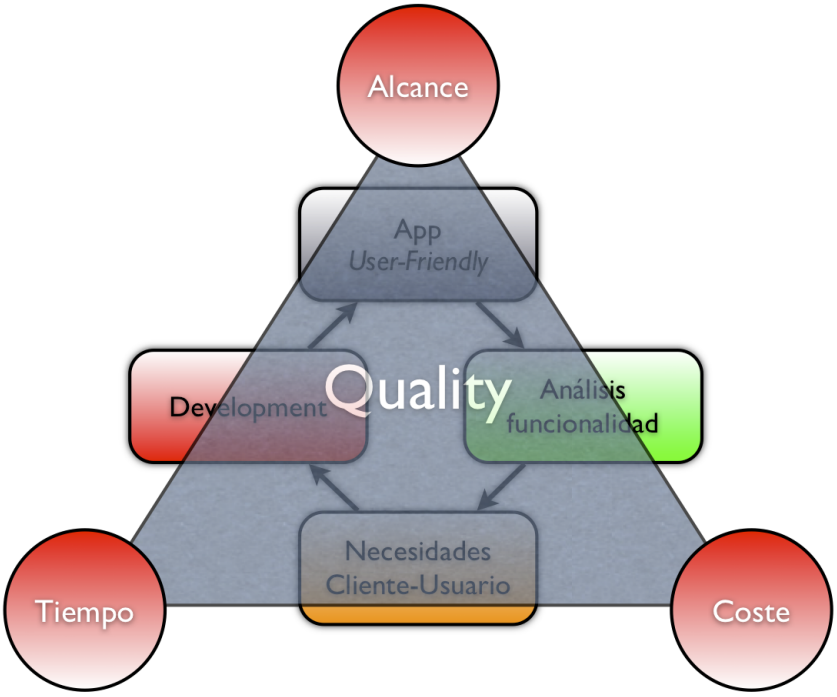

Figura 2.4: Triangulo de calidad y procesos de desarrollo.

Este bucle de desarrollo del software debe estar cubierto mediante un aseguramiento de la calidad, la cual dependerá de tres vértices esenciales en el proyecto de desarrollo (figura 2.4): Alcance del proyecto, que determina las funcionalidades a desarrollar; el tiempo empleado o necesario de desarrollo del producto; y evidentemente el coste necesario de todos los recursos disponibles.

Se trata de que precisamente el área del triángulo de la figura 2.4 sea lo más extensa posible y abarque todos los ámbitos del desarrollo, y dependiendo del movimiento de los vértices ese triángulo será mayor o menor. Por tanto, para asegurar la calidad en un proyecto se debe tener en cuenta que si varía, por ejemplo el coste, tanto el alcance como el tiempo se ven afectados para poder mantener la calidad buscada. Así, en el desarrollo

habrá que tener las herramientas y medidas adecuadas para que el triángulo, y por tanto la calidad, esté asegurado en toda su extensión. Se muevan como se muevan los vértices.

En el desarrollo software, para un aseguramiento de la calidad en la programación se suelen utilizar tests continuos del código. Además, algunas metodologías ágiles se basan incluso en procesos continuos de desarrollo controlados permanentemente mediante tests de código.

Tests de desarrollo

Es muy importante en Aseguramiento de la Calidad ó QA el diseño, planificación, ejecución y resultados de tests aplicados al código y al desarrollo en general del software a producir. Una de las técnicas muy conocidas y bastante implementadas en los últimos tiempos para un desarrollo de alta calidad es el Desarrollo Dirigido a Tests, TDD[5] por sus siglas en inglés.

Lo que propone la técnica TDD es precisamente basar el desarrollo de código software en función a los tests diseñados, para cumplir las funcionalidades buscadas del producto. Es decir, lo primero que se desarrollan son los tests que cumplan con las necesidades específicas y posteriormente programar el código software con el fin de pasar los tests desarrollados. Es por tanto de vital importancia en este tipo de técnicas desarrollar unos tests que cumplan fielmente con los requisitos.

Asegurar una calidad de desarrollo software no es sólo utilizar una técnica o metodología que dé prioridad a los tests de código o de uso, sino también establecer las métricas adecuadas, implementar el desarrollo de tests como una acción cotidiana en la programación de código y considerarlo como parte del propio código del producto. A fin de cuentas los tests van a ser la prueba de que cada línea de desarrollo del producto cumple con la calidad exigida.

[5] Test Driven Development

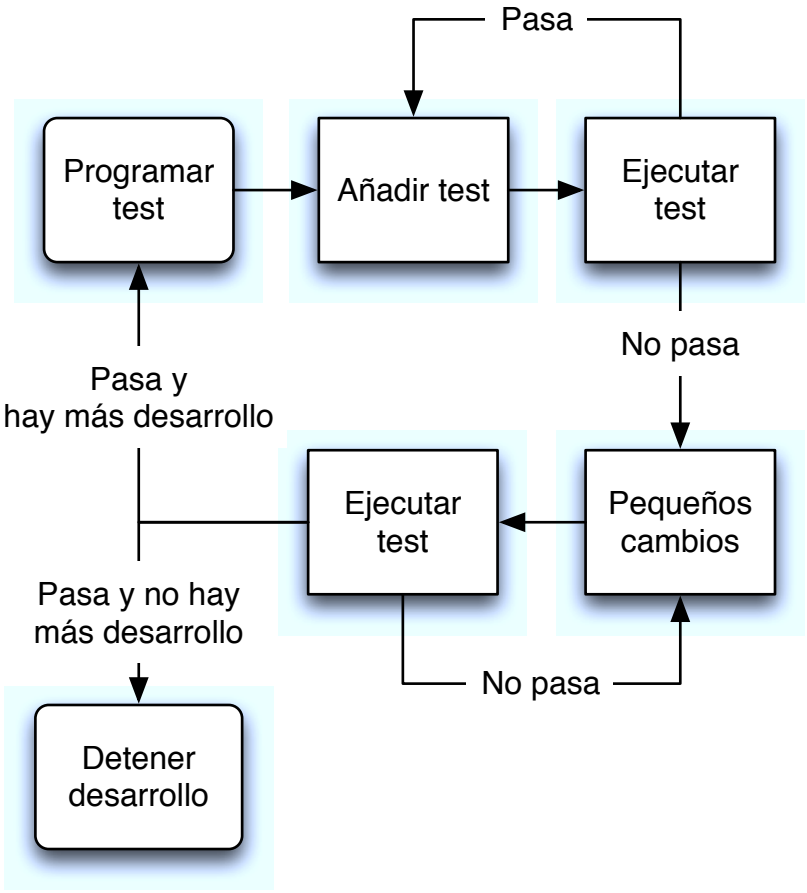

Figura 2.5: Ciclo de desarrollo de tests en TDD

Un cliclo de desarrollo centrado en tests, como el de la figura 2.5 estará en un marco de planificación del proyecto software donde se debe:

1. Establecer normas de calidad
2. Establecer métricas de éxito o fracaso
3. Planificar ejecución de tests
4. Distinguir entre tests unitarios y tests de integración.

Es importante, en el caso de utilizar los tests como herramienta central para el aseguramiento de la calidad, aplicar multitud de tests unitarios, donde se mide la calidad y

buen hacer del código por sí mismo. Y también tests de integración, donde se prueba que las *piezas* desarrolladas entre distintos programadores o equipos de desarrollo encajan a la perfección y con la calidad planificada que cumple el objetivo.

2.3. Integración Continua

En un equipo de desarrollo software la división de tareas es una práctica normal y cada programador se dedica al desarrollo de una tarea y un código específico, el cual es una pieza que posteriormente tiene que encajar en el puzzle del producto final. Por tanto, el trabajo de encajar esas piezas de la mejor forma posible es una tarea en sí complicada e importante en el desarrollo de un proyecto software. En este aspecto y muy familiarizado con el desarrollo ágil tiene cabida la Integración Continua.

La Integración Continua es definida por Martin Fowler [11], uno de los padres de esta, como:

> "Práctica de desarrollo software donde los miembros de un equipo integran su trabajo frecuentemente, por lo menos diariamente y llegando a múltiples integraciones al día. Cada integración se comprueba por una construcción [6] automatizada para detectar errores de integración lo antes posible. Muchos equipos encuentran esta aproximación como una forma de reducir significativamente problemas de integración y que permite desarrollar cohesivamente software de forma más rápida."

Una de las grandes ventajas de la Integración Continua está en el hecho de poder probar la integración del código "en tiempo real". Se podría englobar dentro del concepto *Kaizen* [7], cuyas ventajas se reducen en gran parte a la reducción de tiempo de integración

[6] Referido a'*building* en inglés, cuyo significado comprende la organización/construcción y compilación de código

[7] Mejora Continua. Concepto proveniente del japonés y de la organización industrial que referencia a procedimientos enfocados a mejorar las acciones en todo momento.

con una fiabilidad enorme.

Martin Fowler definió también unas prácticas indispensables para hacer realidad esta reducción de problemas en la integración de desarrollo. Las prácticas a llevar a cabo son:

- Mantener un repositorio de código
- Automatizar la compilación e integración
- Hacer tus propios tests
- *Commits*[8] diarios
- Cada commit debe compilar
- Compilar rápida y ágilmente
- Testar en clon de producción
- Hacerlo fácil
- Resultados accesibles por parte de todos
- Automatizar despliegue

Cada una de estas prácticas debe ser una disciplina de trabajo cotidiano en el desarrollo y completamente transparente al equipo de programación y sus programadores. Basándose entonces en estos aspectos es importante poder realizar las tareas necesarias para disponer siempre de una última versión del código y su integración de las distintas partes, que esté testada y probada. Para ello existen herramientas software que proporcionan un Servidor de Integración Continua, el facilita la realización de esas tareas.

Integrar el código de cada equipo para ir probando cómo encajan las piezas es algo que tiene que ser también iterativo y continuamente probado, para precisamente obtener un

[8]Acción de publicar oficialmente en el repositorio de código local o remoto los cambios realizados al código

producto final de calidad y con las menos incidencias posibles que son fácilmente evitables por una correcta integración. Además, este concepto de Integración Continua provoca que cada parte individual de desarrollo nunca pierda el foco de sentirse parte de un desarrollo global para el que está desarrollando. El desarrollo de un equipo no sólo es importante para el equipo encargado, sino también para el resto de equipos de desarrollo que tienen que integrar su parte del software final. La Integración Continua dirige el desarrollo para que esto no sea un problema en el proyecto.

2.4. Integración del desarrollo en un entorno ágil

Desarrollar y programar una aplicación o cualquier producto o implementación de software no es una tarea sencilla en el propio trabajo de la programación. Además, a mayor complejidad de funcionalidad final, mayor complejidad interna en el desarrollo del proyecto, la cual suele implicar mayor división de tareas, organización de equipos de trabajo, compromiso de ejecución y funcionalidad con el usuario final, mayor planificación, y unos estándares más rígidos de calidad.

Buscar entonces el compromiso de calidad, alcance y tiempo en el desarrollo del proyecto es una tarea difícil, la cual se ha visto que se puede llevar a cabo a través de prácticas de desarrollo y teorías de gestión diseñadas para tal fin, como son las metodologías ágiles. Pero el salto entre la teoría y la práctica no es tan obvio. Las metodologías ágiles usan el sentido común como denominador común en su aplicación. Pero la práctica ha demostrado que aplicar la lógica en los equipos de desarrollo de software no es tan sencillo y debido a las características intrínsecas de los perfiles de los programadores, adaptar una metodología ágil en un proyecto de desarrollo de software tiene sus peculiaridades.

Hablar de integrar el código de un equipo es una tarea importante en el proyecto, pero además el hecho de aplicar una metodología ágil en la gestión tiene sus implicaciones en ambos sentidos. Es decir, la metodología tiene que tener en cuenta la *integración continua*, y ésta estar enfocada completamente al desarrollo en un entorno completamente ágil. Nor-

malmente los conceptos de Metodología Ágil e Integración Continua son conceptos que se suelen tratar aparte, pero en realidad están muy ligados, e integrar un desarrollo en un entorno ágil tiene que tener muy en cuenta las partes donde encajan aquellos dos conceptos.

La integración del código del desarrollo de un equipo en un entorno ágil dependerá de múltiples factores de ambos conceptos. Para ello, es necesario la utilización de herramientas enfocadas a tal fin y la planificación y diseño de una arquitectura de desarrollo que permita sacar el mayor partido a las correctas prácticas.

No es usual enfocar la planificación de un equipo de desarrollo y sus herramientas solamente a las metodologías aplicadas, en el sentido en el que son las propias metodologías las que suelen adaptarse a las particularidades del proyecto y que determinan en numerosos casos las herramientas de desarrollo empleadas. Es por tanto una forma de sacar mayor rendimiento a un proyecto diseñar una arquitectura de desarrollo donde las herramientas proporcionen la mayor funcionalidad en un entorno de proyecto ágil, y faciliten enormemente el desarrollo "ágil" de los programadores.

Será importante entonces:

1. Diseñar una arquitectura de desarrollo adaptada al equipo ágil
2. Seleccionar herramientas de integración adecuadas
3. Seleccionar herramientas de desarrollo de código adaptadas
4. No olvidar el objetivo de calidad

Por tanto, integrar el desarrollo de un proyecto software en un entorno ágil dependerá mucho de la aquitectura de desarrollo del equipo o los equipos de programación y las herramientas seleccionadas en ese diseño, para cumplir así con el objetivo del proyecto de la forma más óptima posible.

Capítulo 3

ANÁLISIS DE HERRAMIENTAS DE LA ARQUITECTURA

3.1. Herramientas de Integración Continua

Tal y como se he mencionado en la sección 2.3 y la integración del código de desarrollo será necesario seleccionar herramientas específicas que permitan poder cumplir con las prácticas que definen dicha Integración Continua, y ayuden en la programación con el fin de maximizar las ventajas que proporciona.

Básicamente las herramientas disponibles que cumplen con estos requisitos son los denominados *Servidores de Integración Continua*. Estas aplicaciones son soluciones administrables gráficamente sobre motores de desarrollo y plataformas que proporcionan una centralización e integración del código de desarrollo, con la posibilidad de hacer un seguimiento exhaustivo en todo momento y capacidad de establecer métricas de calidad, tanto en el código individual como en la integración del trabajo del equipo.

La centralización del código es posible mediante los servidores de repositorio de código, y el servidor de Integración Continua se encarga de usar dicho código centralizado para establecer la monitorización, ejecutar tareas de integración y cumplir con las mé-

tricas de calidad exigidas del proyecto. El servidor de Integración Continua interactúa entonces directamente con el repositorio de código, el cual estará continuamente actualizado conforme cada desarrollador publique sus cambios al código. Esto ocurrirá, según las prácticas de Integración Continua, al menos diariamente.

Además, el servidor de Integración Continua interactúa con el equipo de desarrollo mediante un portal de conexión que hace de interfaz gráfico para la configuración y seguimiento o monitorización del estado de la integración y desarrollo. De esta manera se entra en un clclo de integración del desarrollo realimentado, como se observa en la figura 3.1, que abarca desde la programación de líneas de código de cada programador hasta la integración de todo el desarrollo, las pruebas de calidad del propio código y la información proporcionada al equipo de desarrollo.

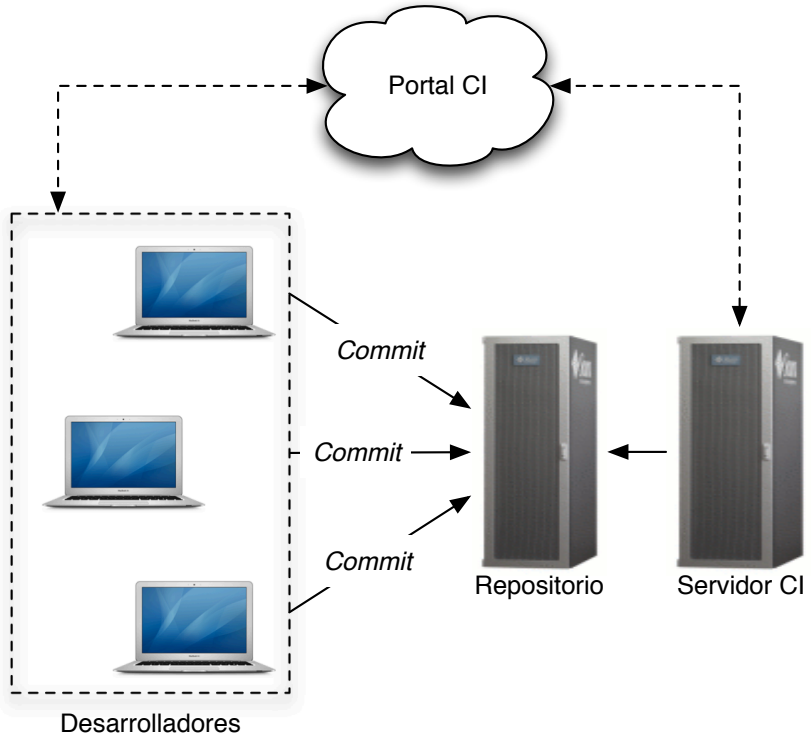

Figura 3.1: Servidor de Integración Continua

El servidor de Integración Continua proporciona multitud de información útil, entre la que destaca:

- Datos de actualización del repositorio de código
- Información en tiempo real e histórico de cada compilación/construcción configurada
- Métricas, errores y tendencias del código mediante tests unitarios
- Métricas, errores y tendencias en los test de integración
- Cambios históricos en el repositorio
- Aceptación o cumplimiento de requisitos de calidad

Gran cantidad de esta información no es proporcionada de por sí por el servidor de Integración Continua, pero otra gran parte proviene de otras herramientas de desarrollo, la cual es monitorizada a través del servidor de Integración Continua, proporcionando una información completamente centralizada y adaptada completamente a la integración del desarrollo.

En gran medida los servidores de Integración Continua son herramientas que proporcionan la realimentación necesaria del proceso de desarrollo para entrar en un círculo iterativo que mejora continuamente el producto y en este aspecto siempre contribuye el concepto de versiones de producto o de desarrollo, que suelen coincidir con iteraciones de programación. Es aquí donde la sincronización con el sistema de control de versiones es una parte muy importante. Así el servidor de Integración Continua no sólo debe escalar la información de estas herramientas y ejecutar las tareas de integración determinadas, sino presentarlas al equipo y sus responsables y reutilizarla internamente para optimizar sus tareas automáticas.

De una forma sencilla se puede entender a estos servidores de integración como motores de automatización de tareas de desarrollo que facilita enormemente la organización

del código y de los módulos programados, para integrarlos y encajarlos perfectamente. Ayudan en este sentido a los miembros del equipo de desarrollo a no preocuparse de las tareas repetitivas enfocadas a integrar el trabajo, sino que consigue establecer como enfoque la integración en sí misma y no la forma de lograrla.

La función básica de los servidores de Integración Continua está enfocada a:

- Automatizar tareas de *building*
- Generar informes y métricas de tests
- Proveer información útil para realimentar el proceso de desarrollo (*feedback*)
- Identificar problemas o errores en la integración del código
- Hacer el proceso de integración transparente al equipo de desarrollo
- Establecer uniformidad en la programación del equipo
- Mantener actualizado y centralizado el código de desarrollo

Por todas estas razones la elección de la herramienta que se utilice para desarrollar en un entorno de Integración Continua debe estar lo más adaptada posible al entorno de desarrollo, y debe cumplir su objetivo de la mejor forma posible. Esta elección vendrá determinada por ciertos criterios de selección que marcarán las necesidades de desarrollo.

Los criterios que suelen marcar la elección de las múltiples soluciones que existen como servidores de Integración Continua se pueden resumir en:

1. Características de la herramienta software, como

   Integración de sistema de control de versiones

   Capacidad de manejo e integración de plataformas de construcción y despliegue

   Metodología de feedback e informes

   Etiquetado

Si soporta dependencia de otros proyectos o no

Facilidad de extender

2. Fiabilidad y rendimiento. Productos open source permitirán hacer pruebas de rendimiento fiables

3. "Longevidad" del software. Sin perder la funcionalidad y robustez de una herramienta consolidada y establecida, hay que tener una visión de futuro y pensar a largo plazo mediante la posibilidad de extender el producto modularizado

4. Entorno objetivo. Hay que tener también una visión más allá de la calidad del código y no perder de vista el entorno de producción, trabajo y ejecución. Se deberá tener en cuenta los sistemas operativos soportados por los servidores, los requerimientos del sistema, el versionado y el soporte de versionado

5. Facilidad de uso. Aunque a veces este criterio puede ser subjetivo, el uso de consolas web, configuración a través de edición de archivos, utilización de asistentes o paneles de control vendrán determinados por los usuarios de la herramienta-servidor que va a ser utilizada y administrada.

Como se puede comprobar, estos criterios de decisión a la hora de seleccionar la herramienta de Integración Continua están enfocados tanto al objetivo del proyecto en sí como a las características del equipo de desarrollo y del entorno en el que se programa. Todos se deben de tener en cuenta, pero es lógico que en función de las necesidades y peculiaridades del proyecto y el equipo de trabajo se establezcan prioridades en los criterios.

Una vez descrito lo que un servidor de Integración Continua ofrece y lo que puede aportar, sería idóneo saber cuál es la oferta de soluciones que existe para poder escoger una solución determinada. No se olvidará, no obstante, que todas deben cumplir un mínimo de requisitos necesarios para la funcionalidad objeto de un servidor de estas características.

Tabla 3.1: Software de Integración Continua

Servidor CI	**Herramientas**	**Tipo de licencia**
Apache Continuum	Maven Ant CVS Subversion Shell script ...	Open Source
CruiseControl	Maven Ant CVS Shell script ...	Open Source
Hudson	Maven Git CVS Mercurial Ant Shell script Plugins	Open Source
Luntbuild	Maven Subversion CVS Mercurial Ant Shell script Rake ...	Open Source

En realidad el concepto de servidor de Integración Continua no está enfocado como una herramienta software específica, sino que en general muchas herramientas de automatización de tareas de desarrollo se configuran para funcionar como servidores de Integración Continua. Por tanto, los servidores que existen en el mercado son productos de software desarrollados para automatizar ciertas tareas de desarrollo, algunos más específicos con el fin de la Integración Continua. Así, en [14] se encuentra una extensa lista de soluciones, y resaltando las más importantes y algunas de sus características están en la tabla 3.1.

Lo curioso de los servidores de Integración Continua más conocidos es que los más utilizados son herramientas *open source*[1], pero que cubren perfectamente las necesidades de desarrollo de cualquier departamento de desarrollo, independientemente del tamaño de la empresa. Son herramientas dirigidas a la Integración Continua del desarrollo software, ya sea de forma comercial o libre. Las herramientas comerciales para este fin suelen ser mucho más específicas y cerradas, y normalmente suelen estar muy especializadas en las necesidades de la propia compañía que la desarrolla.

En realidad, de los servidores mencionados en la tabla 3.1, todos cumplen con los requisitos adecuados, y las valoraciones estarán más dirigidas a cómo encajan en un entorno de desarrollo específico. En este sentido se puede diferenciar entre herramientas más o menos flexibles, más o menos "usables", más o menos integrables con los sistemas, etc.

En este caso, la mayoría de las herramientas estudiadas están pensadas para entornos de desarrollo Java, pero gran parte de ellas son extensibles a otros lenguajes de desarrollo. Además, las propias herramientas están programadas en Java y utilizan librerías y tecnologías basadas en este lenguaje, más considerado en la actualidad como una plataforma de desarrollo. Así por ejemplo, si se quiere emplear el servidor de Integración Continua para automatizar tareas con otros lenguajes existen distintas posibilidades según el servidor que se emplee.

[1] De código abierto, pero con licencias más específicas según el caso y la posibilidad de distribuirlo

CruiseControl, por ejemplo, provee de soluciones distintas para lenguajes de programación como *.NET* o *Ruby*, pudiendo instalar respectivamente las soluciones CruiseControl.NET ó CruiseControl.rb. Por otra parte, el servidor de Integración Continua Hudson, uno de los que más peso está ganando en el sector de la automatización de tareas de desarrollo, se basa en *plugins* o extensiones instalables para añadir funcionalidad extra al servidor básico. Así para el caso expuesto anteriormente de tareas de programación tipo Ruby, existen extensiones instalables al servidor que lo hacen posible, dotándole en este caso de mayor flexibilidad.

Una de las razones por la que se suele usar Java para el desarrollo de estos servidores es precisamente la necesidad de ser una herramienta multi-plataforma y poder ser instalada en cualquier sistema o entorno sin mayor complicación. Pero esto no implica que todas las tareas automatizadas de los servidores de Integración Continua no puedan estar configuradas para diferentes lenguajes de desarrollo o programación. La gran flexibilidad a este respecto suele estar en la posibilidad de uso por parte del servidor de las herramientas del propio sistema o plataforma del servidor, normalmente del sistema operativo. La diferencia de cada herramienta estará en la facilidad o mejor integración precisamente con la plataforma del servidor.

Así, de los servidores de Integración Continua más completos que existen y que mejor se adaptan a las necesidades de este estudio es Hudson, siendo además uno de los que más aceptación en distintos entornos está teniendo. Para comprender más a fondo la funcionalidad de una herramienta de automatización, y más detalladamente un servidor de Integración Continua, se estudiará Hudson, comparándolo también con algunas funcionalidades de servidores también muy extendidos.

Hudson

Hudson es un servidor de Integración Continua muy flexible y fácilmente extensible mediante *plugins*. Es una herramienta completamente abierta de software libre que puede ser usada en cualquier entorno, ya sea para desarrollos software de pequeñas aplicaciones como para programación compleja de soluciones más extensas.

Hudson se define como una herramienta de construcción/compilación y pruebas de proyectos de software de forma continua, que provee de una monitorización de ejecución de tareas internas y externas relacionadas con el proyecto, incluso de forma remota. En definitiva, estas tareas, una vez configuradas, las automatiza el servidor, de forma que la productividad en la construcción del código del proyecto se incremente de forma notable. Los errores y fallos se detecten lo más pronto y de la mejor forma posible, enfocando el desarrollo del código a un alto grado de calidad interna. Esta calidad interna, quedará reflejada en calidad externa una vez se complementen todos los elementos de calidad del producto software.

Las características de Hudson son bastante positivas respecto a los criterios de selección de una herramienta de Integración Continua, referidas anteriormente en este capítulo. Así, enumerándolas según las características de herramientas en los criterios de selección:

1. **Integración de control de versiones**. Hudson se integraba por defecto con CVS y Subversion, aunque en las últimas versiones se han separado mediante *plugins*. También se integra con otros sistemas de repositorio distribuidos, como Git ó Mercurial.

2. **Integración con plataformas de despliegue**. Hudson es completamente compatible con Maven y Ant, dos de las más extendidas herramientas de despliegue para Java, además de la posibilidad de diversos *plugins* para otras herramientas de despliegue de otros lenguajes de programación

3. **Metodología de *feedback***. El servidor web de monitorización de Hudson ofrece

multitud de información de todas las tareas y sus fases asociadas a funcionalidades distintas según sus plugins instalados. Además, se pueden instalar extensiones que permiten el informe mediante correo electrónico, mensajería instantánea o RSS. Se observa en la figura 3.2 una pantalla de monitorización gráfica de tests

4. **Etiquetado**. Todas las tareas de automatización de Hudson tienen un histórico de las compilaciones/construcciones que están marcadas, y guarda toda la información referente a cada una. Además, se puede hacer un seguimiento de la tendencia del histórico mediante gráficos y estadísticas asociadas a la herramienta

5. **Dependencias entre proyectos**. Gracias a la integración que Hudson posee con herramientas de *Construcción del Ciclo de Vida de Desarrollo*, como Maven, la cual será estudiada extensamente en la sección 3.3.1, las dependencias entre proyectos de desarrollo no es un problema. Además, Hudson tiene la posibilidad de definir dependencias secuencialmente de sus propias tareas

6. **Facilidad de extensión**. Precisamente Hudson es una de las herramientas de Integración Continua más extensible mediante *plugins*, ya que el núcleo de su funcionamiento e integración con otras herramientas está basado en estas extensiones. Además, la facilidad de programar extensiones para Hudson es enorme y existe gran variedad de documentación para ello. Una ventaja es la buena predisposición de los creadores de Hudson para proveer de las herramientas necesarias de desarrollo en un lenguaje tan extendido como es Java, en el cual está desarrollado Hudson.

Pero estos seis puntos no son sólo las características definitivas para decantarnos por Hudson como herramienta de Integración Continua en la arquitectura de desarrollo ágil. Se encuentran en estos criterios de selección de servidores de Integración Continua otros puntos fuertes de este servidor, tal y como se visualiza en la tabla 3.2.

Uno de los puntos fuertes de Hudson como elección de herramienta de Integración Continua es precisamente su facilidad de manejo, de instalación y de integración en cualquier entorno de desarrollo. Estos son puntos importantes a considerar en una arquitectura de

Tabla 3.2: Características Hudson de elección de herramientas de Itegración Continua

Criterios de selección	**Características Hudson**
Fiabilidad / Rendimiento	Open Source Extenso soporte en la Comunidad Continua evolución Muy extendido y probado Plataforma Java
Longevidad	Herramienta *joven* Alta proyección de futuro Multitud de plataformas soportadas Fácil disponibilidad de extensiones
Entorno objetivo	Independencia de la plataforma de desarrollo Curva de aprendizaje corta Multiplataforma
Facilidad de uso	De los más fáciles de manejo reconocido Instalación Inmediata y fácil Intuitivo de manejar Fácil interpretación de resultados Cliente web a través del navegador

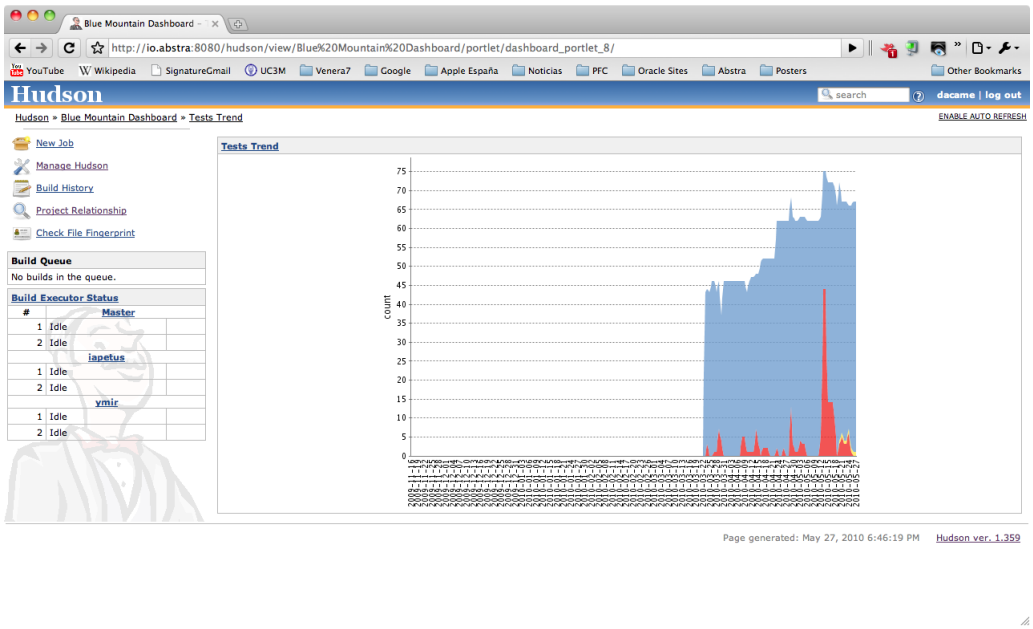

Figura 3.2: Número de tests en el tiempo de las tareas Hudson

desarrollo de un proyecto gestionado mediante metodologías ágiles.

Un servidor de Integración Continua debe ser una herramienta que precisamente ayude a llevar a cabo esa respuesta al cambio en las metodologías ágiles y es comprensible que cuanto más fácil sea de manejar, instalar y desplegar en un entorno de desarrollo de software, más óptimo será el procedimiento de seguimiento del proyecto y de las métricas de calidad en la programación. También, la reacción al cambio será lo más rápida posible. Pero esto no quiere decir que Hudson sea el único servidor que cumpla con estos requisitos, pero sí tiene una gran ventaja al respecto en cuanto a manejo del usuario.

Es importante en un servidor de estas características no cargar al programador de software con tareas externas a la programación, como el hecho de tener que aprender un complicado manejo de una herramienta de desarrollo para poder integrarse con el desarrollo de otros miembros del equipo. Dicho de otra forma, la parte de configuración, instalación y tareas de integración del código debe ser lo más transparente al programador posible. Así, una herramienta intuitiva de manejo no supondrá una carga extra al hecho

de programar código. Si se tiene en cuenta además que la herramienta de Integración Continua ayuda a medir la calidad del código y detectar errores lo antes posible, es de gran ayuda lidiar con una herramienta fácil de usar.

Por esto, herramientas muy gráficas, como pueden ser en este caso Hudson y también Luntbuild, suelen ayudar bastante al trabajo del programador para entender de un vistazo la integración de su código en el proceso de desarrollo.

¿Pero cómo funciona Hudson? Es fácil de entender el funcionamiento de esta herramienta. Básicamente el servidor de Integración Continua tiene configuradas "tareas". Estas tareas no son otra cosa que acciones automatizadas de construcción/compilación de código, o también acciones de configuración de entornos de desarrollo o instalaciones necesarias cada vez que hay un cambio en el código del proyecto. Es decir, el funcionamiento básico de Hudson y de cualquier herramienta de Integración Continua consiste en la automatización de tareas. Se puede ver en la figura 3.3 que la pantalla inicial de Hudson consiste en la visualización de la información básica de esas tareas.

Entonces el funcionamiento crítico de Hudson consistirá en la ejecución de tareas que integran el código que cada programador está continuamente actualizando en el repositorio de código compartido del proyecto. Es como se vio en la figura 3.1. El servidor está continuamente comprobando mediante sus tareas que el código que está desarrollando cada programador cumple con los requisitos exigidos, y si hay un problema Hudson ayuda a detectar y monitorizar los errores.

Cada tarea automatizada de Hudson tiene su configuración propia. Dependiendo del tipo de tarea que se quiere automatizar tendrá unas características determinadas. Por ejemplo, una tarea automatizada de despliegue puede tener que tener configurado un script de ejecución para el volcado de datos en una base de datos. Sin embargo, otra tarea de compilación de código de un framework del mismo proyecto no debe hacer conexiones

Figura 3.3: Pantalla Inicial Hudson

con ninguna base de datos y simplemente debe compilar, pasar tests y ejecutar otras tareas o no, dependiendo de su configuración.

La información que se obtiene de cada tarea es muy extensa y dependerá de los plugins que se hayan añadido a al sistema. La información básica de las tareas viene dada en el menú de estado de cada tarea, al cual se accede mediante la selección de la tarea elegida. En este menú de estado siempre se accederá a los archivos del proyecto, a su estructura y a información visual sobre el estado actual de la tarea, resultados de tests, últimos cambios realizados en el proyecto, dependencias con otros proyectos, espacio ocupado por los datos y archivos, accesos directos a información de construcciones críticas, etc. Esta vista siempre es configurable para Hudson.

Dependiendo de la información global y de cada tarea que se quiere obtener para la integración del proyecto de desarrollo existirá una visualización personalizada de los

resultados de las tareas automatizadas. Por ejemplo, una visualización de estadísticas de tests y de sus construcciones/compilaciones puede ser como la de la figura 3.4.

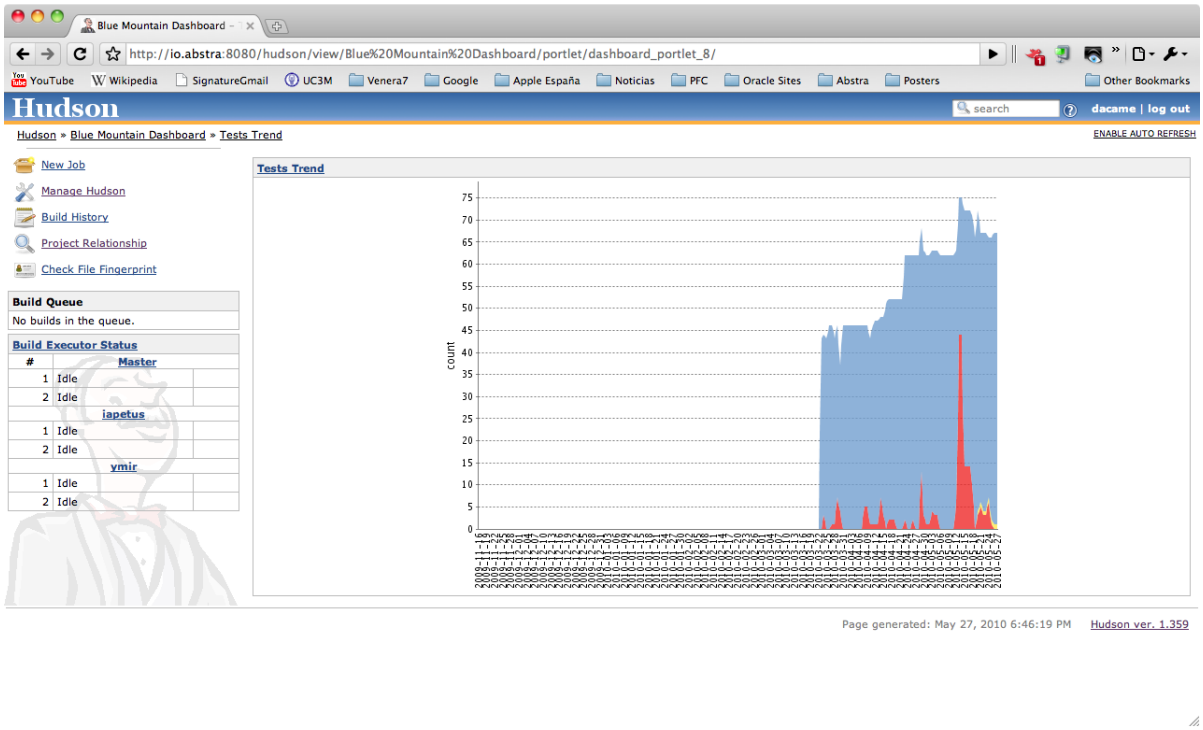

Figura 3.4: Gráfica estadística de tests

En definitiva, Hudson como herramienta de Integración Continua es una herramienta muy fácil de instalar, muy visual y lo sufcientemente flexible como para poder mantenerse acorde a los cambios exigidos en un proyecto de desarrollo adaptado mediante metodologías ágiles.

3.2. Plataformas de desarrollo

Anteriormente se introdujo de forma extendida su conocimiento mediante alguna de sus herramientas el concepto de Integración Continua, base fundamental para poder integrar lo más ágilmente posible las partes de desarrollo de un proyecto software. Pero para desplegar de la mejor forma posible una arquitectura que permita desarrollar el proyecto ágilmente no se debe olvidar dos partes muy importantes, como son las plataformas y

herramientas de desarrollo.

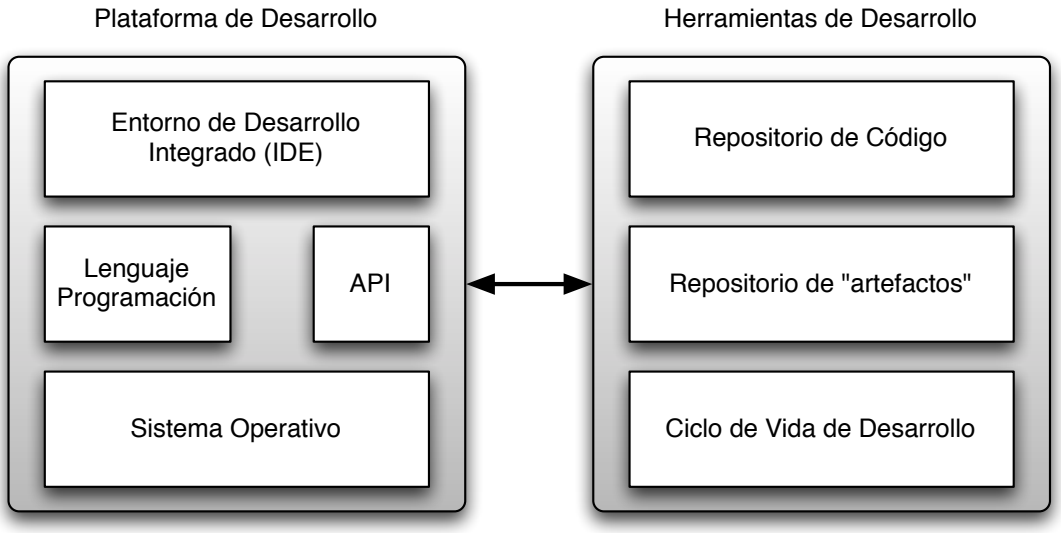

Figura 3.5: Plataforma y Herramientas de Desarrollo

Normalmente las características del proyecto pueden definir a priori las plataformas que a emplear para ejecutarlo, además de ciertas herramientas que ayudarán al cometido de la consecución del proyecto. Pero es una relación en dos sentidos, es decir, hay que tener en cuenta que las plataformas usadas para la programación en el desarrollo software pueden variar las características del proyecto, sobre todo de una ejecución de proyecto que se adapta a los cambios durante el desarrollo, como es un entorno ágil.

Ciertas decisiones que se toman durante el desarrollo de un proyecto donde los cambios aparecen de principio a fin pueden estar afectadas por las plataformas y herramientas utilizadas y es por ello muy importante no perder el foco en esta variable importante, que afecta directamente a la realización de acciones y decisiones de ejecución. Son decisiones tecnológicas, que en mayor o menor medida vienen impuestas por las características necesarias del proyecto, y en otros casos de mayor flexibilidad, son aquellas las que deciden ciertas características del producto.

Es por tanto importante hacer la diferenciación entre los proyectos que marcan los límites tecnológicos por sus características intrínsecas, y los proyectos que son más abiertos en este sentido, y por tanto se pueden ver afectados por la tecnología usada en su desarrollo. Si se trabaja en entornos de programación de productos cerrados, como por ejemplo un conector de comunicaciones para un producto muy específico, el marco que delimita la tecnología a usar será muy estrecho y éste producirá limitaciones al desarrollo del proyecto. Este caso se puede encontrar últimamente en el desarrollo de ciertas tecnologías móviles, donde las aplicaciones deben desarrollarse con un Entorno de Desarrollo Integrado determinado para poder disponer de las librerías y APIs necesarias para poder programar adecuadamente la aplicación, además de cumplir ciertos requisitos de acuerdo de licencia.

Sin embargo, los desarrollos más abiertos, sobre todo en los que las tecnologías estandarizadas tienen mayor grado de implicación, son entornos más flexibles de programación donde las decisiones tecnológicas de la arquitectura de desarrollo pueden afectar algunas características del proyecto, y en algunos casos del propio producto final a desarrollar.

El objetivo entonces a este respecto es que una vez definidas las variables del proyecto que no deben verse afectadas por las decisiones tecnológicas de desarrollo, como pueden ser las características finales del producto, esas decisiones deben adaptarse lo mejor posible a una integración ágil del proyecto, dependiendo de los factores que afectan al mismo. Dicho de otra forma, las decisiones tecnológicas de programación en una arquitectura ágil vendrán marcadas por:

- Características del software a desarrollar
- Capacidad de integración en un entorno ágil

Estas dos variables vienen marcadas a su vez por varios factores que determinan el uso de la plataforma de desarrollo, como se comprueba en la tabla 3.3

Como la variable de las características del software es algo que depende del proyecto

Características Software	Características de Integración
Fiabilidad	Infraestructuras
Rendimiento	Programadores
Multiplataforma	Integración Continua
Flexibilidad	Manteniemiento de la plataforma
Especificación de uso	Tamaño del equipo

Tabla 3.3: Factores Tecnológicos de la Plataforma de Desarrollo

a desarrollar y no de su desarrollo en un un entorno ágil o no, habrá que centrarse entonces en los factores de las características de integración del desarrollo, donde sí afectan directamente en la implementación de una arquitectura ágil de desarrollo. Desde un punto de vista, las características del software ofrecen el marco de trabajo del cual no se puede salir, y las características de integración ayudan a decidir dentro de ese marco establecido.

Entonces, según la figura 3.5, la plataforma de desarrollo se compone de:

1. Entorno de desarrollo (integrado o no)
2. Lenguaje de programación empleado
3. API's necesarias
4. Sistema operativo sobre el que se desarrolla

Los lenguajes de programación y las API's no son elementos que se deciden normalmente según la integración que necesitadas en el entorno de programación, ya que sus factores tecnológicos están más enfocados al producto final y a otras decisiones de desarrollo propias que siempre afectarán a las características del software desarrollado. Así, en la búsqueda de la implantación de una arquitectura de desarrollo ágil, sí va a ser muy interesante analizar desde este punto de vista los Entornos de Desarrollo Integrado, más conocidos por sus siglas en inglés, IDE, y los Sistemas Operativos, en este caso de trabajo

del programador.

3.2.1. Entorno de Desarrollo Integrado

Los Entornos de Desarrollo Integrado ó IDEs proporcionan la funcionalidad de varias herramientas de programación de una forma compacta e integrada entre sí. Normalmente estas aplicaciones ofrecen las funcionalidades de:

- Edición de código con ayudas a la escritura
- Compilador de código (Intérpetre en caso de lenguajes no compilados)
- Detector de errores o *debugger*
- Herramientas de automatización de tareas de código

Estas son las básicas encontradas en la gran mayoría de IDEs, pero lo que marca la diferencia entre una gran parte de ellos es la posibilidad de múltiples funcionalidades extra que facilitarán la programación de código y desarrollo de software según las necesidades del proyecto.

En la multitud de aplicaciones diseñadas para este fin hay que destacar la distinción entre los IDEs gráficos y los no gráficos. Los primeros son entornos de desarrollo donde la visualización gráfica del desarrollo de la programación es esencial para el programador y proporciona herramientas gráficas para poder programar de la forma más cómoda posible. Visualizadores gráficos de partes de código, posibilidad de comparar visualmente distintas partes del código, navegación visual entre partes del proyecto de programación, menús gráficos de acciones, asistentes visuales a la programación, etc. Y por supuesto, con esta capacidad gráfica, los IDEs gráficos ofrecen la posibilidad de convertir partes del código en flujos de procesos o diagramas gráficos.

Los entornos no gráficos suelen estar basados en líneas de comando mediante terminales, y están más enfocados a la rapidez de uso a través de teclado y la simplicidad de visualización, normalmente dirigido a programadores que necesitan mantener más el foco en líneas de código complejas e independientes. También en algunos casos suele ser una decisión de preferencia del propio programador.

Por norma general los IDEs gráficos, que son la amplia mayoría, ofrecen mucha más funcionalidad en entornos de programación de alta complejidad de integración. En proyectos de desarrollo con multitud de programadores que actualizan continuamente el código mediante herramientas de repositorio de código necesitan una política de actualización del código local con el remoto del repositorio para poder saber en qué momento se debe integrar. Esta acción puede estar determinada en el IDE y no se una preocupación continua de usar la herramienta del repositorio para actualizar e integrar código, ya que el propio IDE se encarga de ello con una configuración previa.

El Entorno de Desarrollo Integrado ofrece muchas posibilidades de integración con herramientas de desarrollo utilizadas para sincronizar el código del equipo de trabajo, a la vez que integrarlo con las herramientas de *testing* y pruebas. También es una gran ayuda para detectar errores en el código de forma rápida, visual y evitar errores con antelación. Desde un punto de vista los IDEs hacen que la programación sea mucho más agradable para el programador cuando se encuentra en un entorno de programación de alta integración, donde la parte de su código es una pequeña parte de un gran proyecto.

También depende mucho el estilo de trabajo del programador y numerosas veces el hecho de usar un IDE es decisión de gusto del programador o quien va a escribir o editar código. Esto es algo que afectará en gran medida a la optimización en la integración del código en un entorno ágil de desarrollo, ya que lo que hay que lograr es agilizar lo más posible la parte de sincronización e integración del código sin que el trabajo del programador se vea afectado negativamente.

Para la implementación de una arquitectura ágil se va a ver la necesidad de disponer de IDEs para los programadores, que ayuden a realizar las tareas de actualización e integración del código del equipo con el menor impacto posible en su trabajo individual. Desde el punto de vista de la gestión de este tipo de proyectos las tareas básicas de integración, como cargar el nuevo código al repositorio o tener el repositorio actualizado, debe ser lo más transparente posible al programador, ayudando a éste a mantener el foco en su trabajo de programación y no en tareas de gestión e integración del propio proyecto.

El problema en la necesidad de uso de IDEs está en las preferencias de los programadores, ya que encontrarse con miembros del equipo que no están acostumbrados a usar IDEs y que no suelen ser de su agrado no es tan extraño. En estos casos se debe tener en cuenta la problemática de decidir emplear tiempo en adaptar a los miembros del equipo al uso de estas herramientas o intentar adaptar las herramientas independientes de uso del programador a la arquitectura y la metodología.

Este problema es difícil de resolver, y normalmente hay que intentar no tomar decisiones precipitadas en el uso de las herramientas del programador. No se puede intentar obligar a un programador a usar una herramienta que no es de su agrado y que probablemente no termine usando, siendo esto un freno a la agilidad del proyecto. Se debe ser flexible en las herramientas usadas por los miembros del equipo del proyecto a la hora de programar código, pero no dejar tampoco el uso de herramientas y plataformas que "desagilicen" la ejecución del proyecto de software.

Por tanto, en una arquitectura de desarrollo ágil lo correcto sería buscar un IDE lo suficientemente flexible para el programador no acostumbrado a ello y que no suponga un alto impacto negativo en su trabajo. El objetivo es poder usar un IDE que se adapte a la metodología y que facilite la integración con las múltiples herramientas empleadas en la arquitectura de desarrollo, pero que a su vez sea del agrado del programador. Para esto

no es necesario que todos los miembros del equipo trabajen con el mismo IDE, pero sí que trabajen con uno que tenga la capacidad de integrarse con las herramientas empleadas.

La solución por tanto es encontrar la gama de IDEs que tengan las características adecuadas de adaptación necesaria para un entorno ágil, siendo normalmente aquellos que sean extensibles o que tengan capacidad para ello. Se observa entonces que las características de los IDEs se dividen según la figura 3.6. Es decir se pueden encontrar de la forma:

1. Según objetivo
    - Desarrollo específico
    - Desarrollo múltiple
2. Según compatibilidad de plataforma
    - Plataforma específica
    - Plataforma múltiple

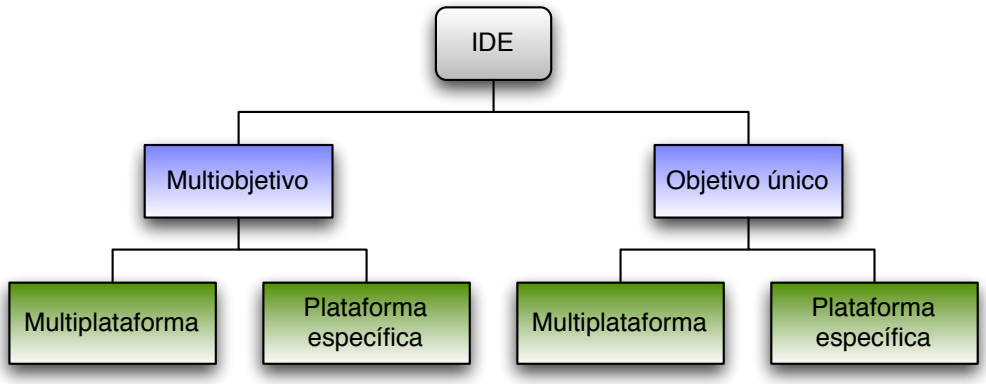

Figura 3.6: Tipos de IDE

En los entornos de desarrollo cuyo objetivo es un desarrollo de un software o tecnología específica es mejor usar IDEs creados para tal fin, es decir, de objetivo específico o único.

Puede ser el caso de aplicaciones de desarrollo creadas por grandes empresas que facilitan el uso de sus tecnologías como ayuda a la programación. Se puede encontrar el ejemplo de la solución de Oracle [2], que tiene a JDeveloper como uno de sus IDEs empleados en sus desarrollos.

Por otro lado existen entornos cuyo objetivo de desarrollo puede ser múltiple, empleando distintas tecnologías según proyecto, distintos lenguajes de programación o distintas herramientas de construcción en función del desarrollo que se esté creando. Para este tipo de desarrollos están las soluciones como NetBeans o Eclipse, dos de los IDEs más conocidos entre la comunidad libre y también empresarial del software. También estos casos suelen ser ideales para entornos en los que durante el desarrollo del proyecto se producen cambios continuos, incluso en las tecnologías usadas. En principio esto suele ser así en proyectos basados en metodologías ágiles, pero no siempre será así.

En la tabla 3.4 se observan algunos de los distintos IDEs más conocidos dentro de la gran multitud que se pueden encontrar según sus necesidades. Se pueden ver sus características según su plataforma y función objetivo.

En definitiva, para una implementación correcta de una arquitectura de desarrollo ágil se ve necesario el empleo de entornos de desarrollo que se integren fácilmente con otras herramientas usadas en este tipo de arquitecturas, como servidores de Integración Continua o repositorios de código y paquetes de desarrollo, y además faciliten enormemente la labor de programación a los miembros del equipo de desarrollo. Normalmente se tiende a una programación visual, donde se tenga en todo momento una percepción del tamaño del proyecto global y la parte a la que pertenece el desarrollo de cada programador.

No hay un IDE que sea mejor que los demás y la elección del uso de IDEs dependerá de todos los factores del proyecto, tanto tecnológicos del producto o implementación software a desarrollar como de las variables de gestión del proyecto, ya mencionadas en la

[2]Empresa que nació en el sector de bases de datos y que actualmente es una de las mayores corporaciones en software empresarial del mundo

IDE	Características generales	Licencia
Eclipse	Multiplataforma Multiobjetivo Extensible Basado en Java	Libre
NetBeans	Multiplataforma Multiobjetivo Extensible Basado en Java	Libre
IntelliJ IDEA	Plataforma específica (Java) Multiobjetivo Semi-extensible	Comercial(Edición Libre)
Visual Studio	Plataforma específica (Windows) Multiobjetivo Funcionalidad según distribución Soporte personalizado	Comercial
JDeveloper	Multiplataforma Objetivo específico (Oracle) Extensible Soporte personalizado	Comercial(Edición Libre)
XCode	Plataforma específica (Apple OS) Objetivo específico (Apple) No extensible	Comercial(Edición Libre)

Tabla 3.4: Principales IDEs

tabla 2.1. Así, también se deberá tener en cuenta las preferencias de los programadores, su trabajo diario y también el tiempo que se necesita emplear en formar a algunos miembros para el empleo de IDEs si fuera necesario. Es en este sentido esencial crear un ambiente de empleo de IDEs como algo positivo para los programadores.

Normalmente, cuando los factores no están bien definidos en un principio, la elección de entornos de desarrollo que sean flexibles, extensibles y muy conocidos y con gran documentación es una opción buena. Así los dos más conocidos como son NetBeans y Eclipse siempre pueden estar en cualquier proyecto de desarrollo que necesite de una arquitectura ágil. Y es importante siempre tener opciones de empleo de IDEs para no tener que obligar a los programadores a usar sólo una solución.

En la arquitectura y caso de estudio se seleccionarán como IDEs adecuados NetBeans y Eclipse indistintamente, por sus características flexibles, su posibilidad de extensión mediante la instalación de paquetes o *plugins*, y la perfecta adecuación a las herramientas de desarrollo utilizadas: Maven y Mercurial.

3.2.2. Sistemas Operativos

Es importante mencionar, aunque tampoco es un factor decisivo en una arquitectura de desarrollo ágil el papel que toman los sistemas operativos empleados dentro de la propia plataforma de desarrollo. La elección de los sistemas operativos de las máquinas empleadas para programar y las que contienen todos los entornos de pruebas, integración y producción pueden ayudar en mayor o menor medida a la implementación de la arquitectura ágil de desarrollo.

En toda la arquitectura de desarrollo es importante diferenciar el hardware empleado para el desarrollo del software, y por tanto los sistemas operativos sobre los que funcionarán las aplicaciones empleadas en el desarrollo. Normalmente se diferenciarán cuatro

grandes entornos:

- Desarrollo. Cada miembro programa localmente su parte.
- *Testing* o Pruebas. Se pasan los tests de calidad del código total actualizado.
- Integración. Se integran todas las partes y se comprueba su correcto funcionamiento.
- Pre-Producción. Se prueba el software en un entorno idéntico al que tendrá el software a desarrollar.

Según la figura 3.7 cada entorno posee una o más máquinas, dependiendo de su cometido y de su integración en la arquitectura de desarrollo. Además, los servidores empleados como servidores de la propia arquitectura de desarrollo, como pueden ser el repositorio de código, el servidor de Integración Continua o servidores de bases de datos de infraestructura, tendrán también sus sistemas operativos sobre los que funcionan las herramientas.

Se distinguen entonces dos partes importantes en referencia a la instalación de sistemas operativos en la arquitectura de desarrollo:

1. Servidores y clientes con herramientas de desarrollo
2. Servidores de los entornos de pruebas y ejecución del proyecto software

La segunda sería la parte sombreada de la figura 3.7, donde están los entornos de Testing, Integración y Pre-producción.

La diferenciación de estas dos partes se basa en la capacidad de adaptación que necesitan los sistemas operativos como parte de la plataforma de desarrollo con las herramientas empleadas en la arquitectura. Por tanto, los servidores y clientes del entorno de desarrollo tendrán que tener como sistemas operativos de la plataforma aquellos que mejor rendimiento, estabilidad y robustez proporcionen a las herramientas de construcción, de repositorios, de bases de datos y que además proporcionen facilidad de uso en los clientes

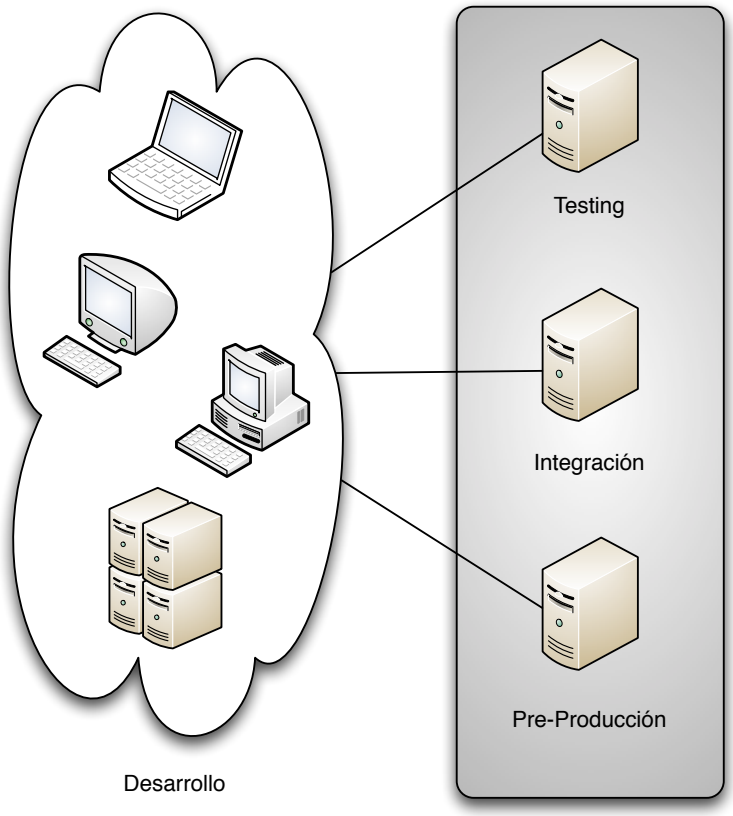

Figura 3.7: Entornos de Desarrollo

que empleen los programadores.

Normalmente las herramientas de desarrollo que se emplearán son soportadas por la mayoría de los sistemas operativos más extendidos, como Windows, distribuciones Linux, Mac OS X y distintos operativos tipo UNIX. Por tanto la decisión de la instalación de los sistemas operativos como plataforma de las herramientas de desarrollo se basará en una decisión de robustez, estabilidad, rendimiento y adaptación a los frecuentes cambios de la metodología ágil.

Clientes de desarrollo

En estos equipos, normalmente ordenadores portátiles o de sobremesa empleados por los programadores para programar, es recomendable tener sistemas operativos que ofrezcan una facilidad de uso al programador, a la vez que se integre perfectamente con los sistemas de gestión de las herramientas de desarrollo. En estos equipos es donde se instalarán los IDEs y también donde se realizan las primeras pruebas unitarias de código por parte del programador. En estos equipos de uso personal lo más empleado suele ser:

- Microsoft Windows. La ventaja es que es el sistema más usado y disponibilidad de software para él. El inconveniente suele estar en la inestabilidad y la dependencia de pagos de licencia para el soporte.

- Distribuciones Linux para sobremesa. Suele ser una buena solución intermedia entre estabilidad y manejo de uso para desarrolladores, con la ventaja de ser muy flexible para usuarios avanzados, como suele ser el caso de muchos programadores. La desventaja es que el mantenimiento e instalación de nuevas aplicaciones necesarias no siempre es tan sencilla.

- Mac OS X. Su gran ventaja es la facilidad de uso y de instalación de software de desarrollo, además de la ventaja de poder ejecutar aplicaciones para UNIX. El inconveniente está en la necesidad de usar ordenadores específicos de la marca Apple.

De entre estas tres opciones, normalmente tienen influencia de decisión varios factores como las preferencias del programador, las características del software a desarrollar y la posibilidad de interconexión con los servidores de desarrollo. Existen otros criterios que no se deben olvidar, como el económico y la facilidad de soporte del sistema.

Lo importantes entonces, como parte del diseño e implementación de la arquitectura de desarrollo, es escoger sistemas operativos de los clientes que se adapten a la flexibilidad necesaria del proyecto por sus características de la metodología ágil en su gestión. Por norma general, es buena idea mantener una homogeneidad en los sistemas operativos

instalados en los equipos de desarrollo y dejar más como preferencia del programador la selección del IDE dentro de los adecuados para la arquitectura.

Servidores del entorno de desarrollo

Estos servidores son:

1. Servidor de Integración Continua
2. Servidor de repositorio de código
3. Servidor de bases de datos.

En estos casos prima la estabilidad y robustez, además del rendimiento, debido a la alta disponibilidad que debe tener y estar siempre activo las veinticuatro horas del día y actualizado en todo momento. Por esto, normalmente se seleccionan sistemas operativos que cumplan perfectamente estos requisitos, como suelen ser distribuciones Linux de servidores y sistemas del tipo UNIX.

En el caso del estudio de la arquitectura se optará por el sistema operativo OpenSolaris, que es un UNIX, debido a que además de cumplir los requisitos de estabilidad requeridos, da la opción de dividir en una sola máquina física varios servidores de forma virtual con un sólo sistema operativo central, sin virtualización de software propiamente dicha. Se verá más detenidamente el funcionamiento de esta parte en el capítulo 4.

Servidores de Integración, Testing y Pre-producción

Al igual que en los servidores de desarrollo, la estabilidad, robustez, rendimiento y disponibilidad, son indispensables, y sus sistemas operativos serán seleccionador según estos criterios. La excepción en este caso está en el servidor de Pre-producción.

El servidor de Pre-producción debe ser un entorno gemelo o clon al entorno sobre el que se ejecutará y correrá la aplicación o solución software que se está desarrollando como

objetivo del proyecto. Esto es así porque antes de desplegar la solución final al cliente es necesario comprobar su funcionamiento en un entorno lo más parecido posible al que servirá de base a la solución final y sobre la que se ejecutará. Este entorno además, en un proyecto basado en metodología ágil y Scrum más específicamente, el cliente debe tener siempre acceso a él y poder realizar pruebas sobre las últimas versiones desplegadas en este entorno.

Por tanto, el sistema operativo del entorno Pre-producción viene dado por las necesidades del cliente final y características del proyecto. El caso de estudio necesitará de Windows XP, Windows Vista y Windows 7 para las pruebas.

En los otros dos entornos se escogerán soluciones muy estables de Linux, que esta vez pertenecerán a un sistema de virtualización de software, y por tanto, los tres entornos estarán ejecutándose bajo un sistema operativo central de virtualización, que permitirá tener los entornos Linux y Windows instalados en un sistema central estable y robusto que controla toda disponibilidad de pruebas e integración. La ventaja es que sin tener que disponer de hardware diferente se pueden tener servidores diferentes según las necesidades de cada entorno, además de la estabilidad que ofrece el operativo de virtualización.

Todos los entornos y sus sistemas operativos deben poder tener acceso desde los otros, por lo que es importante que los sistemas permitan conexiones de forma segura, estable y sin pérdida de conexión o datos. Aquí también entraría en escena las capacidades hardware, pero no son motivo de estudio ni diseño en la arquitectura de desarrollo, ya que la elección de hardware viene más impuesta por otros motivos de la empresa, ya sean económicos o de cualquier otra índole. Pero sí hay que tener en cuenta que los sistemas operativos que utilizados para la arquitectura de desarrollo deben sacar el máximo partido al hardware, e incluso la posibilidad de adquirir alguno nuevo si es necesario y es rentable.

3.3. Herramientas de desarrollo

Cuando se tratan las herramientas de desarrollo en la arquitectura se refieren a aquellas aplicaciones o soluciones que una vez creado el código ayudan a la hora de construirlo, probarlo, integrarlo y provee de las partes necesarias para el correcto desarrollo iterativo, necesario en el proyecto de software.

Las herramientas de desarrollo irán en concordancia a las plataformas de desarrollo ya explicadas en la sección anterior 3.2, pero también éstas van a ayudar a seleccionar las plataformas, tal y como se ha visto anteriormente. La importancia de no pensar en una secuencia de instalación es importante y las herramientas de desarrollo no es una cuestión de seleccionarlas o elegirlas antes o después de la plataforma, sino durante la planificación de la arquitectura y por tanto al mismo tiempo de la elección de plataformas escogidas.

Los servidores de Integración Continua son entendidos también como herramientas de desarrollo, ya que al fin y al cabo son los responsables de la automatización de tareas que integran toda la programación para obtener un resultado final con una calidad determinada, y que ayude a iterar en una metodología ágil lo mejor posible. Pero como se ha visto en la sección 3.1, su importancia es tal que ha tenido una mención a parte. Entonces las herramientas de desarrollo que aquí ocupan y que necesitan una mención especial por su importancia en la implantación de una arquitectura de desarrollo adecuada a un proyecto software gestionado mediante metodología ágil serían:

- Herramientas del ciclo de vida de construcción [3]. Sirven para la compilación, interpretación o "construcción" de código y su automatización.
- Gestión de artefactos. Son herramientas de centralización de librerías y artefactos de programación necesarios para las anteriores.
- Gestión de repositorios. Son herramientas que gestionan el repositorio de código global del proyecto.

[3] En inglés conocidas como las *Lifecycle Build Tools*

En cada una de las anteriores herramientas se ha escogido la más adecuada para este tipo de proyectos basados en metodología ágil y se explicarán las razones, exponiendo también las alternativas y sus ventajas e inconvenientes.

Maven será la herramienta de construcción del ciclo de vida del código, Nexus será la herramienta de gestión de artefactos de Maven, y Mercurial será el repositorio de código.

Es muy importante decir, que estas herramientas están encaminadas a la implementación de una arquitectura ágil cuyo desarrollo tiene relaciones directas con Java como plataforma, y como lenguaje de desarrollo indirecto. Esto se verá más detalladamente en el capítulo 4

Cabría mencionar también alguna herramienta de seguimiento y gestión [4] de tareas del proyecto, pero estaría más enfocado a la gestión de todos los proyectos de software de una empresa, y normalmente estas herramientas ya están instaladas en la mayoría de empresas que desarrollan software. Por tanto, no entraría en la implementación de una arquitectura de desarrollo, sino más bien en la elección de herramientas de la empresa de desarrollo para realizar el seguimiento de todos sus proyectos y trabajo realizado.

Sólo se mencionarán entonces que dos herramientas de seguimiento son muy conocidas en el mundo del desarrollo software, como son Redmine, cuya información completa puede encontrarse en [17] y es de licencia libre; y JIRA, solución comercial muy extendida donde se puede encontrar documentación e información en [6]. Otras soluciones son propias de algunas empresas comerciales o más específicas.

[4] Más conocidas normalmente por su nombre en inglés *Issue Trackers*.

3.3.1. Maven

Maven es una de las herramientas más conocidas, extendidas y completas que existen de construcción del ciclo de vida del código software. Es una herramienta que está creada para todo tipo de proyectos software que tengan alguna relación con el lenguaje de programación Java, aunque su extensión a otros lenguajes ha sido inevitable por su funcionamiento efectivo y óptimo a la hora de flexibilizar la parte de integración y despliegue del código en proyecto software.

De forma más detallada Maven se define como:

> *Herramienta de gestión y comprensión de proyectos de desarrollo de software, normalmente en Java, que permite compilar, testar integridad y compilación, desplegar, documentar y gestionar las dependencias de código de forma sencilla y extensible.*

Es una herramienta abierta y libre, ya que pertenece a la Fundación Apache [5] claramente orientada al trabajo en equipo, siendo esta es una de las principales razones por las que es elegida como herramienta ideal para metodologías ágiles de programación, y posterior Integración Continua con otras herramientas software.

La herramienta es compleja pero fácil de usar y muy rápida a la hora de compilar/interpretar y desplegar proyectos. Entre las ventajas de usar Maven como herramienta de desarrollo en el trabajo en equipo, se puede destacar:

1. La dependencia entre proyectos no es problema
2. Un mismo proyecto se puede ejecutar en distintos entornos. Sólo hay que cambiar un archivo de configuración
3. La integración del trabajo de desarrolladores es transparente

[5] Fundación de software que ayuda a la comunidad *open-source* con múltiples proyectos. Se encuentra en la referencia [3]

4. Hay una maximización de la cohesión del código y una minimización del acoplamiento

5. Fácil reutilización del código

Para los desarrolladores familiarizados con herramientas similares, como Ant, también del proyecto Apache, Maven ofrece funcionalidades similares, con la diferencia de gestionar las dependencias de proyectos de una forma mucho más sencilla y comprensible, optimizando notablemente el tiempo de desarrollo en trabajos multiproyecto.

Es decir, Ant es otra herramientas de gestión del ciclo de vida del software que se utiliza en gran cantidad de proyectos basados en plataformas Java, entendiendo en este caso a Java más como una plataforma y no como lenguaje de programación fundamental. Ant ha sido durante mucho tiempo una herramienta empleada en el despliegue de proyectos Java que automatiza la integración del desarrollo a la hora del despliegue.

En este sentido Maven surgió de la necesidad de poder desplegar e integrar el código en esos proyectos Java gestionando automáticamente las dependencias con otros proyectos también integrados y desplegados mediante la misma herramienta. Con Ant se tenían que programar las dependencias mediante complicados *scripts* o programas, mientras que Maven tiene un motor de gestión de las dependencias que se configuran fácilmente en un archivo XML.

Una de las grandes ventajas de Maven como parte de una arquitectura de desarrollo ágil es precisamente la gran flexibilidad que ofrece gracias a su modularidad y su extensibilidad mediante *plugins* y extensiones. Muchos de ellos vienen incluidos en el paquete básico de Maven y otros muchos se pueden obtener de múltiples sitios o incluso ser desarrollados por el propio equipo de desarrollo si las necesidades son muy específicas.

Para entender el funcionamiento de Maven se puede observar el diagrama de la figura 3.8. Se distinguen tres partes: la parte del programador o “desarrollador”, la parte del

núcleo o motor de Maven y la salida que se obtiene (*output*).

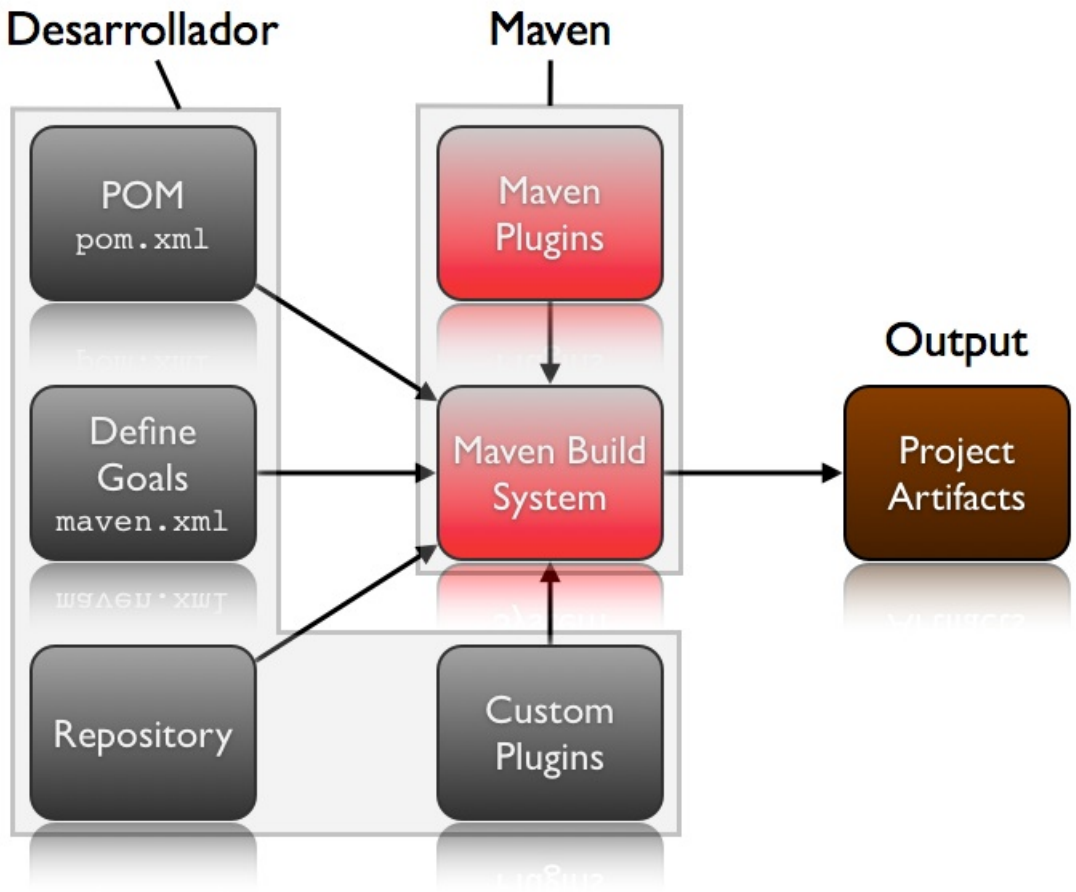

Figura 3.8: Núcleo de Maven

La parte del programador consiste en la configuración de Maven para automatizar las tareas:

- Define el llamado POM, *Project Object Model*, que básicamente es el modelo del proyecto del programador (entendiendo proyecto como parte que programa para el proyecto global del software a desarrollar) donde define dependencias de compilación o interpretación del código, *plugins* necesarios usar, parámetros específicos de la programación y configuraciones de tests

- Define las acciones de Maven, denominadas *goals*, que son las fases que se deben ejecutar en el "ciclo de vida" para obtener la salida final
- Configura los repositorios de los que necesita Maven para resolver las dependencias definidas
- Configura y/o programa los plugins necesarios que no incluya Maven

Una vez el programador ha configurado y definido todos los pasos Maven se encarga de procesar el código, compilarlo/interpretarlo, pasar los tests definidos, resolver las dependencias configuradas, publicar la documentación incluida y desplegar los paquetes o artefactos necesarios, entre otras acciones o *goals* que haya definido el programador.

El resultado de la construcción es un paquete en lenguaje Java (paquetes tipo JAR, WAR ó EAR), que podrá ser utilizado por cualquier otra construcción Maven, siempre que se le indique en el POM de cualquier otro proyecto. Esta es la forma con la que es muy sencillo automatizar tareas de construcción de código de pequeños proyectos interdependientes que forman parte de un proyecto de mayor envergadura.

Estos paquetes que se obtienen en la salida (*output*) son los denominados *artefactos*, y son esenciales en la forma que Maven depende siempre de artefactos para resolver sus dependencias. Todos los plugins y artefactos que Maven usa están desarrollados y programados en Java, aunque también se pueden crear con otros lenguajes, normalmente orientados a objetos[6], que finalmente se comprimen en los artefactos con extensión Java.

Los artefactos que usa Maven suelen estar disponibles en la red o internet y depende de los repositorios definidos de donde los usa y donde los despliega una vez construidos. Por esta razón, cuando el proyecto software global tiene interdependencias complejas, con multitud de artefactos a gestionar y con multitud de repositorios, es recomendable tener herramientas de gestión de artefactos que no hacen dependientes de la conexión a la red.

[6]Entendida como el tipo de programación denominada Programación Orientada a Objetos

Se hablarán de estas herramientas en el siguiente punto de este capítulo.

Un ejemplo de POM es el mostrado en el el código 3.1, que es la configuración muy básica de una aplicación web que se despliega en un paquete en formato WAR. Con este archivo y una estructura de directorios determinada, ejecutando el comando Maven adecuado según la acción o *goal*que se quiera hacer, ya sea instalar, pasar tests, compilar/interpretar, procesar las fuentes, desplegar la aplicaión, o incluso crear documentación, se construye automáticamente la configuración determinada.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>cc.abstra.mavenbook.ch05</groupId>
    <artifactId>simple-webapp</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>simple-webapp-abstra Maven webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
```

```
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    <build>
        <finalName>simple-webapp</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>maven-jetty-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Listing 3.1: Plantilla POM para una aplicación web

Lo único que hay que hacer para que Maven pueda gestionar automáticamente el ciclo de vida de la construcción del código es programar las fuentes del código en una estructura determinada que Maven pueda reconocer para hacer su trabajo, tal y como se ve en la figura 3.9. Los directorios que reconoce se detallan en la tabla 3.4. Dependiendo del lenguaje del proyecto se podrán añadir directorios específicos que están documentados según los *plugins* empleados.

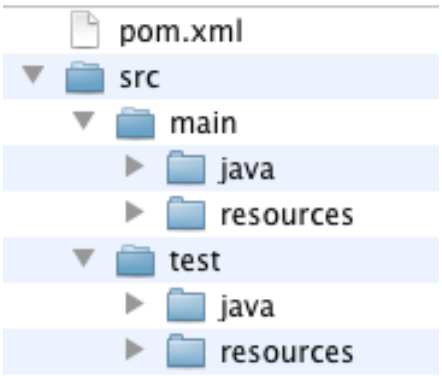

Figura 3.9: Estructura de directorios Maven en aplicación Java

Las estructuras de directorios no siempre hay que conocerlas. Una gran ventaja de Maven es que contiene numerosos *arquetipos*, que son plantillas iniciales para empezar a trabajar con un proyecto determinado. Así plantillas de determinadas aplicaciones web, de programación determinados módulos, de lenguajes específicos... están disponibles para cualquier instalación de Maven y que facilita enormemente el trabajo del programador no teniendo que preocuparse tanto de los recursos, directorios y dependencias, provistos en estos *arquetipos*.

La automatización que consigue el ciclo de vida básico de Maven consiste en:

- Procesar código fuente
- Compilar o interpretar

Tabla 3.5: Directorios Maven

Directorio	**Descripción**
src/main/<*java, scala, ruby, ...*>	Código fuente o librerías del lenguaje determinado
src/main/resources	Recursos del código o de las librerías
src/main/filters	Filtros de recursos
src/main/assembly	Descriptores de "ensamblaje"
src/main/config	Ficheros de configuración
src/main/webapp	Código fuente de aplicaciones web
src/test/<*java, scala, ruby, ...*>	Código fuente de tests del lenguaje
src/test/resources	Ficheros de recursos de tests
src/test/filters	Filtros de los recursos de los tests
src/site	Ficheros relacionados con el portal de documentación
LICENSE.txt	Términos de licencia del proyecto
README.txt	Instrucciones, sugerencias e indicaciones

- Procesar clases (orientación a objetos)
- Procesar código de tests
- Compilar o interpretar tests
- Ejecutar tests
- Preparar empaquetado (según el tipo de empaquetado, WAR, JAR, EAR, etc..)
- Instalar local o remotamente el paquete

Además de estas acciones se pueden ejecutar otras opciones dependiendo de los *plugins* que tenga instalado Maven. Así su versatilidad y flexibilidad son enormes, pudiendo realizar multitud de tareas en una construcción del código. Por ejemplo se pueden realizar despliegues de aplicaciones web, generación de documentación y reportes, acciones específicas de un lenguaje determinado o actualización de paquetes y *artefactos*, todo en una misma construcción.

En principio Maven es una herramienta que funciona mediante línea de comando, pero gracias a la integración que tiene con la mayoría de IDEs del mercado se pueden ejecutar todas sus tareas desde ellos y configurarlo todo en el mismo entorno. Esta es una gran ventaja para todo tipo de desarrollo ágil, teniendo centralizado toda la acción sensible al cambio en el proyecto. Además, Maven y en general las herramientas de gestión de construcción de código de proyectos son claves en la Integración Continua, proveyendo al servidor de Integración Continua de los informes y datos necesarios para automatizar las tareas y generar los informes globales.

Queda claro que Maven tiene grandes ventajas para la automatización de la construcción de código, que facilita al programador todo el proceso de programación del código y la visualización de los resultados a través de la conexión con las herramientas de Integración Continua. En definitiva, Maven es una de las distintas herramientas de gestión de proyectos de programación software diseñada para la automatización de tareas para la

Ventajas de Maven	**Desventajas de Maven**
Dependencias automáticas con otros proyectos	Complejidad en proyectos específicos
Resolución de dependencias transitivas	Curva de aprendizaje compleja
Fácil configuración	Sensible al tiempo de implementación
Transparente al programador	
Integración Continua	
Amplia documentación	

Tabla 3.6: Ventajas y desventajas de Maven

construcción de código, y dirigidas a obtener una mayor transparencia en los resultados de la programación, una menor complejidad en la relación de proyectos y, por tanto, una mejora de la integración del desarrollo en la agilidad de proyectos.

Además de las ventajas observadas, Maven también tiene sus inconvenientes, y como se ve en la tabla 3.6 el gran problema que puede tener esta herramienta es su alto grado de complejidad si se quiere llevar más allá de las tareas simples de la programación, lo que implica una curva de aprendizaje no deseable para proyectos demasiado complejos o específicos. Como ejemplo para el caso de un proyecto Java 100 % y con tecnologías bien documentadas y extendidas puede ser muy fácil gestionar el ciclo de vida de programación del proyecto con Maven, pero para otro proyecto basado en lenguajes poco conocidos o muy específicos alejados de la programación orientada a objetos puede ser mucho más complicado, siendo necesaria la intervención de expertos en estas herramientas para poder automatizar la construcción de código con Maven.

Otras herramientas alternativas a Maven que cumplen la misma finalidad, pero con características algo diferentes pueden ser:

1. **Apache Buildr**. Herramienta basada en el lenguaje Ruby, con funcionalidades implícitas de otros lenguajes además de Java y que funciona con los mismos repositorios de artefactos de Maven, además de otros. Su configuración es muy sencilla mediante

archivos de tipo Ruby, pero no está tan extendida ni documentada como Maven.

2. **Ivy**. Es una extensión de Ant como herramienta de despliegue de proyectos basados en lenguaje Java. Básicamente Ivy es una capa que resuelve y automatiza las dependencias de las tareas de la herramienta Ant. La conjunción de las dos herramientas, también de Apache, ofrecen la misma funcionalidad que Maven. La desventaja respecto a Maven es que es más compleja de configurar, pero con la ventaja de ofrecer aún mayor flexibilidad.

3. **Gradle**. Se basa en la configuración mediante un propio Lenguaje Específico de Dominio (DSL) a través del lenguaje dinámico *Groovy* proporcionando una estructura intuitiva en la configuración. Con características similares a Maven esta herramienta está más enfocada a la construcción y automatización de proyectos software para la *Web*. Es la que menos extendida está de las mencionadas.

Adecuadamente configuradas e implementadas estas herramientas en el equipo de desarrollo de software, proporcionarán fácilmente la adaptación de los pequeños proyectos de código en los que se divide el proyecto de desarrollo global, y permite integrar toda la programación en el proceso de calidad del proyecto.

3.3.2. Nexus

Herramientas como Maven, encargadas de la automatización de la construcción de código del proyecto dependen de recursos que suelen estar disponibles en los llamados repositorios. Éstos, por norma general están a disposición de los programadores en la *Nube* [7] y dependen en numerosas ocasiones del funcionamiento de la conexión al exterior. El funcionamiento en este sentido de Maven es según demanda, es decir, se descarga de su respectivo repositorio los artefactos necesarios según la necesidad a la hora de la construcción del código .

[7] Entendiendo el término *nube (cloud)* como la infraestructura de Internet.

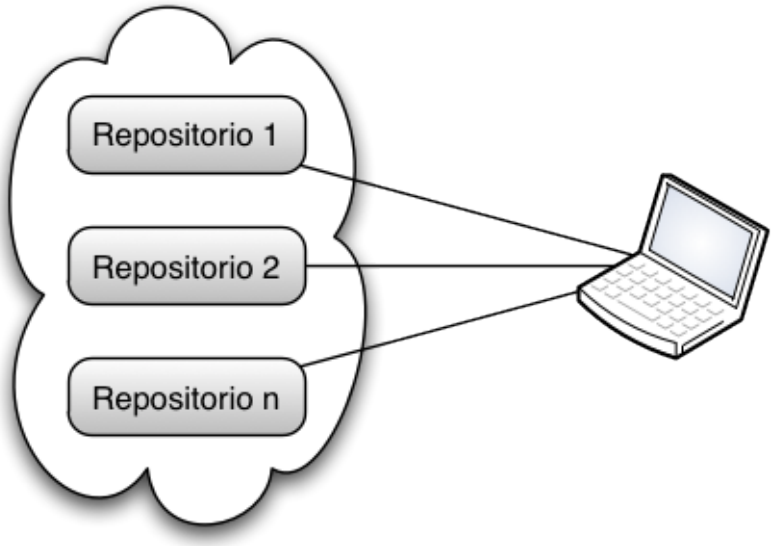

Figura 3.10: Conexión de repositorios Maven

Por defecto Maven está configurado con unos repositorios determinados en Internet, pero si se necesitan añadir repositorios de artefactos que no están disponibles en los preconfigurados se puede hacer especificando en el archivo de configuración POM del proyecto la dirección donde se encuentra el repositorio.

Cuando el proyecto de software a desarrollar es muy complejo se suelen añadir multitud de repositorios para descargar y hace que la configuración Maven de cada proyecto se haga más compleja, además de tener los mismos repositorios configurados varias veces en distintos proyectos individuales que pertenecen al mismo global. Además, la ubicación en Internet de los repositorios supone ciertas ventajas e inconvenientes.

Las ventajas de disponer de los repositorios en la Nube está en la disponibilidad continua que existe, pero con la desventaja de depender de la conexión al exterior, con el consecuente problema de seguridad para entornos que deben estar controlados. Estas y las anteriores razones demuestran una necesidad de gestión de repositorios de artefactos tipo

Maven que debe asegurar la centralización de los artefactos usados para la programación por cada uno de los programadores, teniendo que gestionar además las incompatibilidades que puede haber entre versiones existentes.

Existen así los llamados gestores de repositorios Maven. Son herramientas tipo *proxy*[8] con una alta configuración y personalización, gestionando el acceso a los artefactos requeridos por los programadores para el desarrollo del código a internet. En definitiva lo que permiten es que cuando se requiere cierto artefacto de desarrollo o versión especifica, el gestor de repositorios se conecta una sola vez a internet, lo descarga y lo mantiene disponible en la red interna para el resto de peticiones que se realicen de ese determinado artefacto.

Evidentemente, los gestores de repositorio son óptimos en entornos donde la conexión a la red e internet no es un problema, y normalmente en la actualidad los entornos de desarrollo de software no plantean problema alguno en este aspecto. Así, aparecen los dos gestores de repositorio de artefactos Maven más conocidos en el mercado:

- Artifactory, una solución *open source* de la compañía JFrog
- Nexus, de la empresa Sonatype, especializada en soluciones de desarrollo y creadores[9] de Maven.

Las dos soluciones anteriores son altamente conocidas y proporcionan funcionalidades muy similares, siendo algunas características técnicas y el foco de cada solución lo que las diferencia.

Artifactory es una solución muy bien lograda desde la facilidad de manejo y uso gracias a su interfaz intuitivo y la simplicidad de configuración. Sin embargo, Nexus es una solución más completa para empresas de programación con altas exigencias y complejidad

[8] En el sentido de "servidor proxy", permitiendo el acceso gestionado y distribuido a internet
[9] Jason Van Zyl, creador de Maven, trabaja para Sonatype

de configuración, teniendo la opción de la solución libre y gratuita o la versión profesional, más completa y con soporte para empresas. Artifactory por otro lado también ofrece una solución profesional de pago, pero en forma de extensiones a la aplicación, como integración con autenticación LDAP, búsquedas inteligentes, etc.

En este caso, la decisión como herramienta de gestión de los repositorios necesarios de los artefactos de desarrollo fue a parar a Nexus por diferentes razones:

1. Extensa documentación y soporte para la versión gratuita, y por tanto más económica
2. Integración propia con autenticación LDAP
3. Configuración completa y extensa
4. Fácil instalación en servidor sin necesidad de servidor de aplicaciones
5. Compatibilidad con multitud de entornos
6. Disponibilidad de numerosas extensiones o *plugins*
7. Integración completa con diferentes IDEs

En definitva, Nexus, como gestor de repositorios de artefactos va a ser de gran utilidad y va a proporcionar una gran ventaja para poder establecer una arquitectura de desarrollo enfocada a la agilidad y calidad, donde todos los artefactos usados por los desarrolladores están alojados en el mismo servidor Nexus, el cuál se los descarga en el mismo momento de la necesidad por parte del programador para poder programar.

Se logra con ello una centralización y unificación de los artefactos, entendidos como librerías de programación en muchos casos, que atienden a razones de calidad del desarrollo y evitan problemas de diferenciación del uso por parte de los programadores.

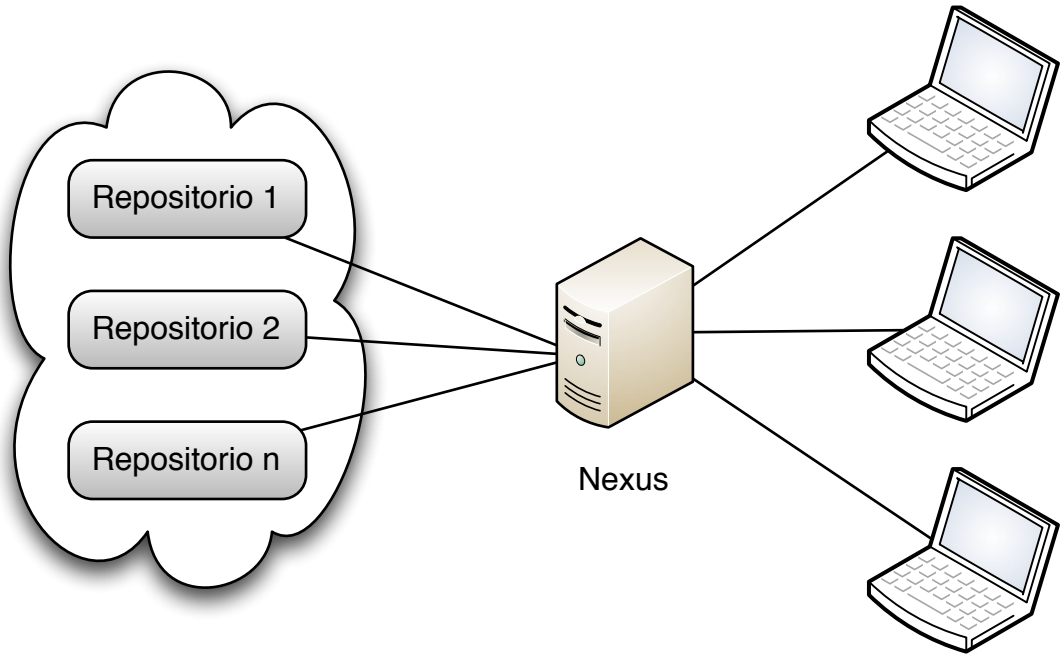

Figura 3.11: Centralización de repositorios Maven

En la programación y la centralización del código, la gestión de repositorios de artefactos para la construcción y automatización es esencial, pero también para ello debe serlo también la centralización y disponibilidad automática del código. Para ello existen entonces las herramientas de repositorio de código y control de revisiones.

3.3.3. Mercurial

Sin centrarse mucho en las herramientas de repositorio de código y control de versiones o revisiones, ya que son herramientas ampliamente extendidas en la mayoría de empresas y equipos de desarrollo de software, se analizarán brevemente las ventajas e inconvenientes de seguir un desarrollo de programación mediante una herramienta de repositorio de código distribuida como Mercurial, pero haciendo una pequeña introducción con las diferencias de los sistemas centralizados y distribuidos.

Se ha estado reseñando la importancia del entorno continuamente cambiante de la programación de software y de lo difícil que es gestionar ese continuo cambio y desarrollo de la programación. En el momento que un equipo encargado de una parte del desarrollo está trabajando, otros equipos también están programando e influyendo directamente en la programación de otros equipos distintos, lo que significa que cuando el código de desarrollo se deba compartir para ir desarrollando la solución global se debe tener en cuenta el cambio que se ha ido actualizando.

Precisamente para este cambio continuo las herramientas de repositorio de código o control de versiones hacen posible que el código que cada programador o equipo cambie esté disponible para los demás participantes en el desarrollo. Y aquí hay dos diferentes formas de llevarlo a cabo, las cuales definen las características de las herramientas a utilizar:

- Sistema de sincronización centralizada
- Sistema de sincronización distribuida

Control de versiones centralizado

Es el tradicional y muy empleado durante mucho tiempo. Básicamente su funcionamiento está basado en que todos los cambios hechos en el código se van publicando en la rama central del desarrollo del servidor del repositorio de código para que todo el código nuevo esté disponible para el resto de programadores y toda la actualización del código se resuelve a través de la rama central. Y cuando se refiere a una rama se entiende como una línea de cambios consecutivos que se desarrollan paralelamente al desarrollo principal pero sin afectarle hasta que se integren los cambios.

Los repositorios centralizados de código son un modelo cliente-servidor, donde los cambios del programador (cliente) se propagan al repositorio central (servidor). Los cambios, denominados normalmente como revisiones, son quienes van a marcar los distintos puntos de identificación en en el desarrollo temporal del código, así siempre se puede volver a

cualquier punto del desarrollo siempre que se desee, pero en estos sistemas cliente-servidor se tiene que hacer siempre a través del sistema central en el servidor.

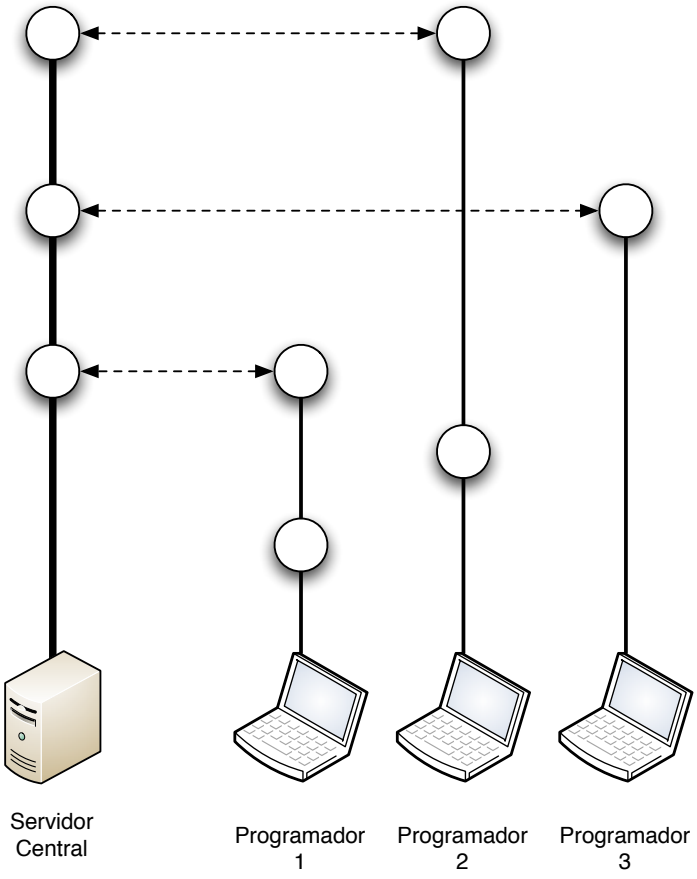

Figura 3.12: Control de Versiones Centralizado

La ventaja de estos sistemas de control de revisiones es la marcada centralización que tienen, en el sentido de la disciplina que marcan en el desarrollo para publicar el código y tenerlo continuamente disponible en la rama central para todos los participantes. La desventaja, sin embargo también está marcada en su sistema central, y es que en proyectos donde la complejidad de integración es grande el tiempo de desarrollo aumenta si se necesitan de amplios tests antes de publicar el código con los requisitos de calidad. Hace además complejo la tarea de grabar las modificaciones, ya que cada una de ellas tiene que copiarse del servidor, aplicar la modificación y volver a salvarlo en el servidor.

Las dos soluciones *opensource* centralizadas más conocidas son CVS, que desde los años 90 se emplea en el desarrollo de software centralizado, y más actualmente Subversion, proyecto perteneciente a Apache y muy utilizada exitosamente en la actualidad. Como solución comercial muy conocida está la herramienta de Microsoft denominada Team Software Foundation, que ofrece más funcionalidad que un control de revisiones, como seguimiento de tareas y otras más.

Control de versiones distribuido

Las herramientas de repositorio de código para el control de revisiones distribuido presentan funcionalidades añadidas a los de cliente-servidor con un funcionamiento diferente. Es decir, pueden funcionar de forma centralizada, guardando una copia en el servidor, pero la línea de desarrollo se sigue mediante la copia de referencias a las modificaciones, que se realizan localmente en la máquina del programador. La diferencia de funcionamiento radica en un sistema *punto a punto* en vez de cliente-servidor, así las distintas copias del código en las diferentes máquinas se sincronizan entre ellas y el servidor central es simplemente una más donde se guarda una copia que no se modifica de forma local. Dicho de otra forma, en el sistema distribuido cada código local de programación es cliente y servidor a la vez.

Las diferencias y ventajas de los repositorios de código distribuido están en:

- Operaciones sobre el código rápidas al no tener que estar continuamente comunicándose con el servidor
- Cada copia local sirve de copia de seguridad de los cambios y el código.

La desventaja de estos sistemas puede estar en que la descentralización puede llevar a cierto comportamiento caótico de los programadores, pero para eso se puede ir comunicando con una rama creada en un servidor central que vale de sistema de copia de seguridad

de los proyectos creados, tal y como se observa en la figura 3.13, donde los programadores se sincronizan los cambios entre sí y se comunican con el servidor central del código sólo para ir cargando o descargando la copia integrada y sincronizada.

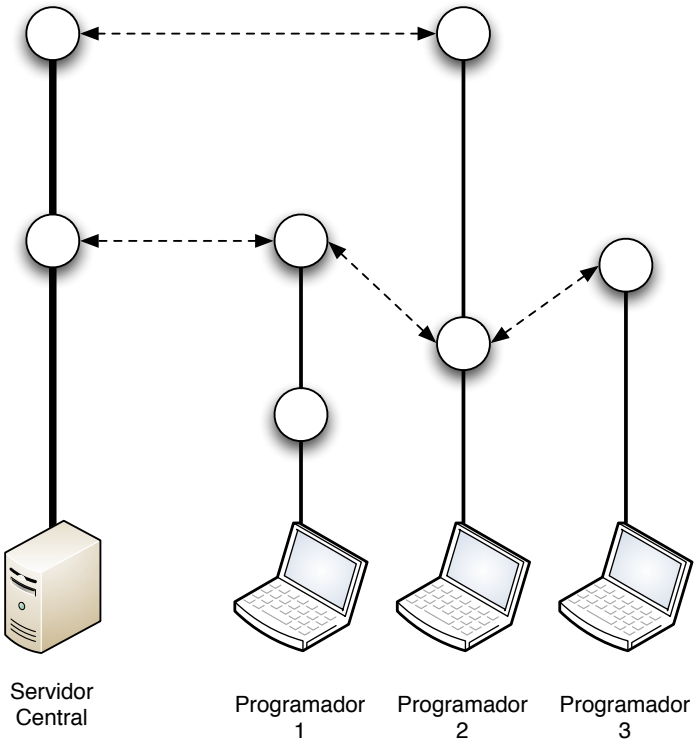

Figura 3.13: Control de Versiones Distribuido

Elección del sistema distribuido

Entre los servidores de sistemas de control de revisiones distribuidos más conocidos y usados están Git y Mercurial, ambas soluciones *open source* y que están muy extendidas en todo tipo de proyectos de desarrollo software. Tal vez Git esté más extendido que Mercurial, pero las dos soluciones son igualmente accesibles y con cada vez más aceptación.

En este caso se empleará un sistema distribuido por las siguientes razones:

- Equipo de desarrollo pequeño
- Programación de los miembros interdependiente entre sí
- Integración rápida que encaja perfectamente en un desarrollo ágil
- Posibilidad de mantener centralización necesaria en el entorno

En definitiva, el sistema distribuido ofrece ventajas necesarias en la arquitectura y que además mantiene la ventaja de centralización del modelo cliente-servidor, aunque para ello se tiene que apoyar desde la gestión del proyecto, hecho que en la gestión ágil está implícita de por sí.

Siendo estrictos, la elección entre Mercurial o Git, por ejemplo, es más una decisión de preferencias de los implantadores de la arquitectura que de funcionalidades. Teniendo esto en cuenta la elección de Mercurial se basa en el conocimiento ya adquirido de la herramienta en la empresa que desarrolla el producto, de la facilidad de instalación y de la continua mejora del producto con un alto grado de actualización y soporte de la comunidad, por no hablar de su bajo precio por ser *open source*.

Capítulo 4

DISEÑO Y DESPLIEGUE DE LA ARQUITECTURA

4.1. Fases

Teniendo en cuenta que la implantación de la arquitectura de desarrollo a ejecutar va a ser la base de un desarrollo con una metodología ágil no significa que la propia implantación deba ser así, sobre todo debido a que no necesita del desarrollo específico de ningún tipo de producto software, sino que se basa en la elección de las herramientas y recursos específicos y las estrategias de desarrollo del proyecto correctas. Pero para una mejor adaptación de la arquitectura a la posterior gestión de los proyectos que se vayan a desarrollar sobre ella se seguirá una implantación ágil, con una gestión mediante Scrum y una fijación en el tiempo de proyectos clásicos, para así proveer al cliente de la arquitectura rápidamente y adaptado a su trabajo. Entonces en este proyecto las fases a desarrollar son: Estudio o diseño, planificación, implementación y despliegue.

4.1.1. Estudio o diseño

En esta fase se analizará el conocimiento obtenido y se estudiará la problemática para su implantación sobre las herramientas necesarias en una arquitectura de desarrollo sobre

entorno ágil. Además se debe analizar la forma de implantar una arquitectura de este tipo y realizar un diseño adecuado para el entorno de desarrollo elegido. Los objetivos en esta fase serán:

- Analizar los conocimientos teóricos adquiridos sobre arquitecturas de desarrollo
- Conocer las metodologías aplicadas en este tipo de arquitecturas
- Conocer las variables críticas en este tipo de entorno
- Analizar métodos de calidad del software y sus implicaciones
- Encontrar un punto óptimo para la correcta integración de herramientas

Según lo visto en los capítulos anteriores 2 y 3 las arquitecturas de desarrollo software poseen una complejidad ligada no sólo al desarrollo específico a llevar a cabo sino también a la realimentación continua de este tipo de proyectos en su ejecución. Todo esto lleva a la necesidad de analizar las variables expuestas en la tabla 2.1 y que son las que determinarán el grado de adaptación a la metodología ágil de desarrollo o no.

Este estudio se basa en la implantación de una arquitectura óptima para el desarrollo de un producto software mediante metodología ágil, por lo que entonces las variables que se deben tener en cuenta deben aproximarse a lo establecido, es decir, la implantación se dirige a equipos de desarrollo pequeños o medianos, colaboradores, no dispersos y en continua formación; una gestión del proyecto dinámica, retroalimentada y que fomente la participación de sus miembros; un cliente accesible e implicado en todo el proceso; uso de las herramientas necesarias, con procedimientos de participación del equipo y cambios de adaptación continuos; y flexibilidad en las entregas y coste no fijo, sino según recursos.

En todo proyecto software, tal y como está en la figura 4.1, el flujo de información entre quien demanda el producto (cliente) y quien lo desarrolla (empresa u organización de desarrollo) está delimitado normalmente por el contrato en el *input* (entrada de información) y por el producto resultante en el *output* (salida de información). La arquitectura

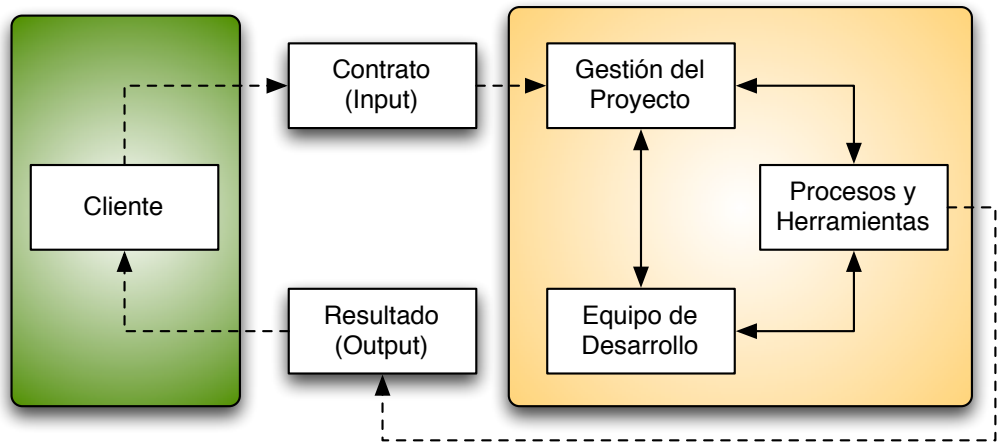

Figura 4.1: Proyecto Desarrollo Software

de desarrollo se sitúa entonces entre ese *input* y ese *output* y es la que permitirá la agilidad en el flujo de información y realización del proyecto a la hora del desarrollo.

Se observan entonces las tres categorías de variables en el entorno que son críticas a la hora de analizar las características del proyecto: el **equipo de desarrollo**, la **gestión del proyecto** y los **procesos y herramientas** empleados.

Los recursos importantes para estas tres categorías vienen determinados de la forma:

1. **Equipo de desarrollo**. Personal de programadores y administradores de sistemas, y equipamiento informáticos empleados para cada uno.

2. **Gestión del proyecto**. Jefes de proyecto y líderes de equipos.

3. **Procesos y herramientas empleados**. Equipamiento hardware y software para el desarrollo de la programación y también para el seguimiento y verificación de la calidad del proyecto. También es la capa de comunicación entre el equipo de desarrollo y la gestión del proyecto.

Estos recursos son los que van a estar implicados en el uso de la arquitectura de desarrollo y por tanto quienes en un entorno de desarrollo ágil necesitarán las herramientas y

un correcto diseño en la arquitectura para el uso que optimicen su labor en la programación ágil.

Según lo estudiado y aprendido en el capítulo 2 existen ciertas consideraciones a tener en cuenta en el diseño de la arquitectura que van a determinar las herramientas a emplear y el diseño de su intregración e implantación en el sistema. Estas consideraciones van a marcar ciertos enfoques y objetivos a tener en cuenta:

- Calidad del código como foco en el desarrollo
- Integración Continua como medio y fin en la programación
- Comunicación transparente entre miembros del equipo y gestión del proyecto
- El uso en el desarrollo de metodologías ágiles enfocadas a la gestión, como Scrum
- Automatización de gran cantidad de procesos desarrollados por el programador que son de infraestructura y no de desarrollo del producto.

Para la consecución de estos objetivos la arquitectura a diseñar deberá entonces tener las herramientas descritas en el capítulo 3 de forma que funcione tal y como se describe en la figura 4.2. Todos los programadores poseen en sus equipos individuales de desarrollo a Maven como herramienta de construcción del código y su respectivo IDE, el cual puede hacer del mencionado Maven. Éste obtiene los artefactos de desarrollo necesarios para el código y sus dependencias del repositorio de artefactos Nexus, el cual los descarga de internet en el momento que un programador los necesita por primera vez en toda la arquitectura. El resto de ocasiones ya está almacenado de forma local en la arquitectura. Cada IDE, además, tiene instalado Mercurial como herramienta de repositorio de código distribuido, que descarga y sincroniza los cambios en el código con el resto de programadores y el repositorio de código central. A su vez el servidor de Integración Continua Hudson tiene una instalación Maven y el código actualizado desde el repositorio de código y realiza automáticamente la integración del código, la ejecución de los tests de integración y desarrollo global y la implantación de los sistemas en Pre-Producción. Estos tres

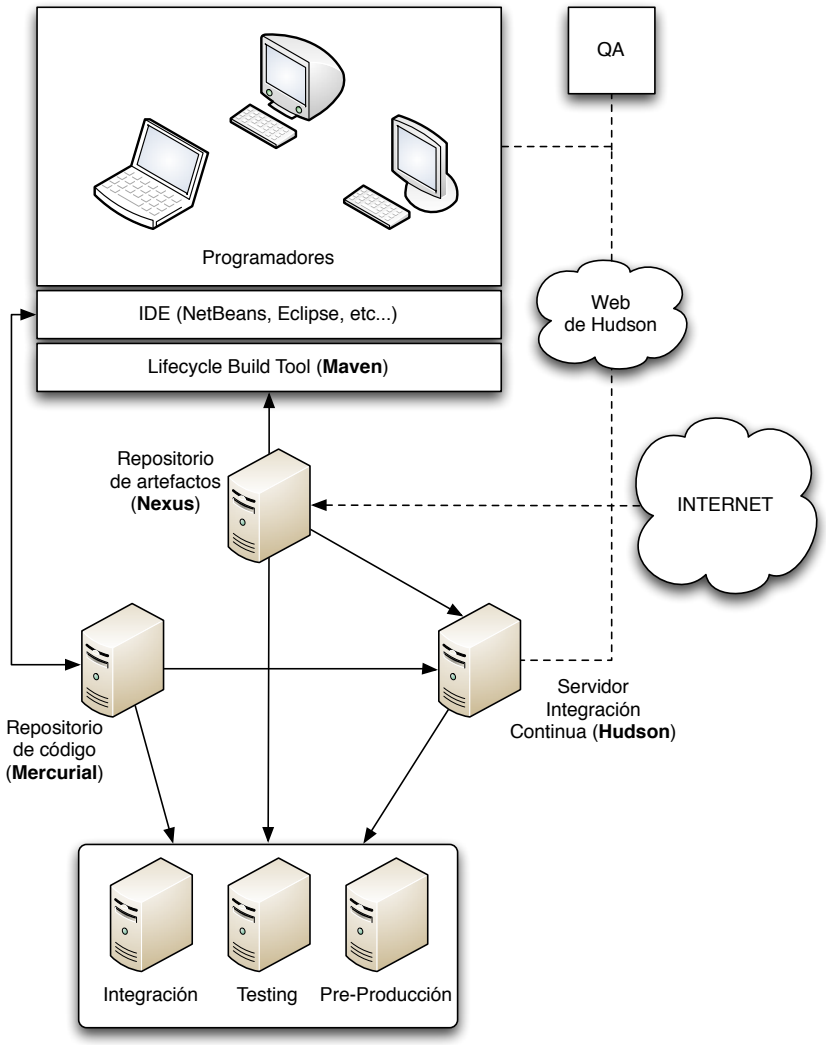

Figura 4.2: Diseño de la arquitectura ágil

entornos a su vez necesitan del código y de los artefactos necesarios para desarrollar su función, la cual la obtienen del servidor Mercurial y Nexus respectivamente. Sin olvidar que el servidor de Integración Continua Hudson provee de una web donde los programadores y los encargados del aseguramiento de la calidad (QA) automatizan y consultan el desarrollo de las construcciones de código según los distintos entornos.

Quien orquesta todos los acontecimientos es el servidor de Integración Continua, Hudson en este caso. Es por tanto de vital importancia que este servidor esté bien configurado

y que las tareas que ejecute sean monitorizadas continuamente. Dependiendo del tipo de desarrollo a llevar a cabo la configuración será de una forma u otra.

En la figura 4.2 no están representados los servidores físicos tal y como aparecen, ya que por temas de costes, rendimiento y fiabilidad, no siempre se debe emplear una máquina hardware distinta para cada uno, sino que tendrá sentido, por ejemplo, que el repositorio de código y de artefactos estén en la misma máquina, ya sea física o virtual, siendo sólo una instalación software la que diferencie a cada uno. Como muchas de las soluciones usan bases de datos, ya sean relacionales o no, algunas tendrán que poseer copias de seguridad, y esta vez en máquinas distintas. Pero desde el punto de vista del desarrollo, donde el código y sus cambios son el recurso que es imprescindible no perder, no hay problemas con las copias de seguridad, ya que los repositorios de código y artefactos están por duplicado en cada máquina del programador que trabaja con ello, además de poseer siempre la copia de seguridad que existe en el servidor central de repositorio.

Normalmente, dependiendo también de la propia infraestructura de la empresa u organización de desarrollo del producto software, las copias de seguridad se diseñarán de una forma u otra, pero en una arquitectura como esta lo más importante para guardar esas copias o *backups*[1] es tener en cuenta que tanto el servidor de Integración Continua como los entornos de desarrollo son prioritarios. Además, las bases de datos que se empleen en las distintas soluciones de la arquitectura siempre deberán tener copias disponibles en caso de corrupción de datos. Normalmente las empresas que desarrollan software suelen tener este tipo de políticas de *backups*, pero es necesario indicarlo a la hora de implementar la arquitectura.

[1]Término en inglés de copia de seguridad y empleado más comúnmente en la jerga informática en la mayoría de idiomas

4.1.2. Planificación

En esta fase de planificación hay que tener en cuenta el desarrollo ágil y la propia implantación así se guiará. Es decir, se plantea una planificación base para la ejecución en la implantación de la arquitectura pero que se irá desarrollando también conforme se ejecute el proyecto y se implante, con tal de mejorar el proceso y encontrar la aplicación perfecta para el uso de la arquitectura. Los objetivos de esta fase por tanto serán:

- Planificar las tareas necesarias para la instalación de herramientas y servidores
- Diseñar las configuraciones que se deben realizar sobre las tareas planificadas
- Planificar las pruebas que se deben realizar para retroalimentar el proceso
- Establecer la documentación necesaria
- Determinar arquitectura de la metodología a seguir.

Se debe tener en cuenta que la implementación de la arquitectura, la cual se realizará en la fase siguiente no es algo que se realiza en un entorno donde no hay una sola línea de código, sino que es algo que se realiza a la misma vez que los programadores están escribiendo líneas. Es decir, la implementación de la arquitectura completa no debe ser bloqueante al desarrollo de cada programador. Se va implementando a la misma vez a la que se va desarrollando, razón por la cual la metodología ágil cobra más sentido aún.

Todas las tareas que se deben realizar y por tanto a planificar van a estar marcadas dentro de la propia metodología del proyecto de desarrollo del producto o productos software, y en este caso se contempla sobre un equipo de desarrollo que se guía bajo metodología Scrum. Por tanto, las tareas, al desarrollarse paralelamente al inicio del desarrollo del producto sobre la arquitectura estarán dentro del *Product Backlog* de la metodología Scrum, y estarán desarrollándose durante los primeros Sprints de la misma metodología. Así, para crear la primera iteración, los grupos de tareas planificadas a tener en cuenta son:

1. Instalación en los equipos de programadores de Maven y los IDEs especificados por cada uno

2. Instalación de los repositorios de desarrollo, tanto del servidor Mercurial como del gestor de repositorios de Maven, Nexus

3. Instalación del servidor de Integración Continua y configuración global del propio servidor Hudson

4. Preparación e Instalación de los entornos de desarrollo

5. Integración y personalización de Maven, los IDEs y Mercurial, de las máquinas individuales

6. Configuración de Nexus para el uso continuado conforme evoluciones el proyecto y los repositorios requeridos

7. Adición de los repositorios básicos en Nexus e integración con las instalaciones Maven individuales

8. Configuración y automatización del servidor de Integración Continua

9. Preparación de la documentación para el uso de las herramientas por los programadores

10. Configuración del entorno de pruebas de desarrollo para el control de calidad del código

11. Despliegue de la arquitectura mediante ejecución de pruebas con el código desarrollado

12. Análisis de pruebas y resultados

13. Modificación de tests y reconfiguraciones en Integración Continua

14. Seguimiento y evolución del código y arquitectura.

Con este grupo de tareas, en unas dos iteraciones, unas cuatro semanas, se tendrá implantada una arquitectura básica para el desarrollo ágil de un proyecto software. Pero al ser las dos primeras iteraciones donde se ha montado básicamente el esqueleto de la arquitectura, y al ser una metodología ágil la que guía el proyecto del producto software, éste a su vez es quien da la forma final a la arquitectura de desarrollo y se tendrán que emplear las iteraciones necesarias hasta que el producto en sí tenga también un desarrollo básico sobre el que trabajar. Por tanto, y dependiendo de la envergadura del proyecto que se desarrolle sobre la arquitectura, lo menos unas cuatro iteraciones más serán necesarias, lo que implica unas ocho semanas (dos meses) más.

Esta primera iteración de la implementación, junto con la fase anterior y las siguientes a describir, planificadas en una iteración cero (entendiendo aquí iteración como el proceso global de las fases y no de la metodología) de todo el proceso, se puede representar en un diagrama clásico de Gantt, para entender cómo se describiría el proceso de formación de las tareas en la metodología ágil. Es decir, se puede entender para establecer prioridad en las tareas de una iteración Scrum, dentro de sus dos semanas, cómo afrontar su implementación según el diagrama de la figura 4.3. Las tareas de la implementación dentro de las dos primeras semanas tienen las dependencias establecidas en el gráfico y se gestionarán según estas prioridades.

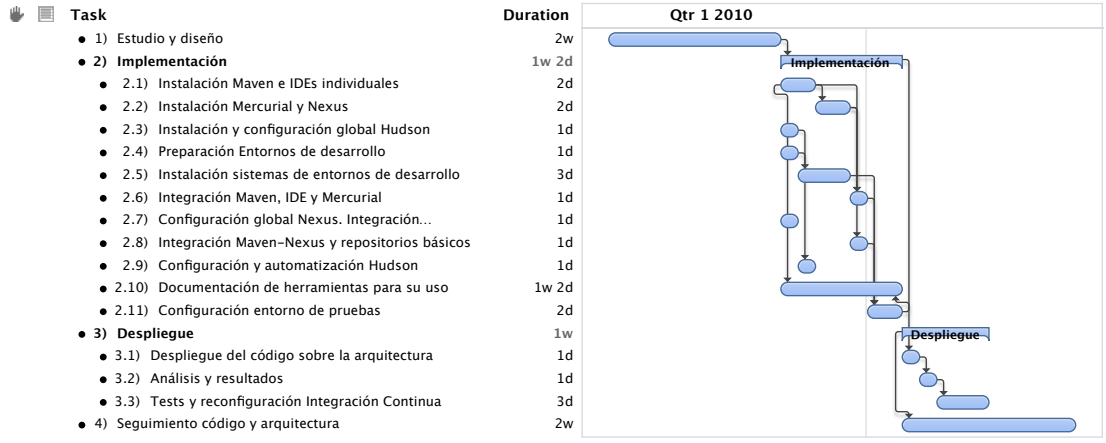

Figura 4.3: Diagrama Gantt orientativo del proyecto de implementación

Así por ejemplo, se entiende que las instalaciones de Hudson por un lado, y de Maven, los IDEs y Mercurial por otro, son tareas que pueden empezar sin que haya acabado la otra, pero las configuraciones e integraciones sí dependen de la secuencia. Se entiende perfectamente de forma que para Instalar Hudson como servidor de Integración Continua no es necesario tener todavía el gestor de repositorios Maven ni el repositorio de código. Sin embargo, cuando se necesiten configurar las tareas de automatización de la Integración Continua será necesario tener las instalaciones de la plataforma de desarrollo y de la Integración Continua completas.

Durante el proceso de tareas instalación y configuración inicial se tiene que ir documentando el uso de las herramientas de la arquitectura, para así una vez finalizada la fase de implementación el despliegue se pueda llevar a cabo de la forma más eficiente y rápida posibles.

Pero no se puede usar como plan el diagrama Gantt de la figura 4.3 para la planificación exacta en el tiempo, debido a que se está implantando para una metodología Scrum, y por tanto siguiendo el proyecto también con parte de la metodología. Si se recuerda el triángulo de calidad de la figura 2.4, en un proyecto basado en metodologías ágiles la entrega será flexible en función de los vértices del coste y el alcance del proyecto, por lo que calidad como fin último del proyecto a desarrollar no se verá comprometida. Por tanto, la línea temporal de esas 6 semanas aproximadas aumenta hasta que se llegue a una solución de compromiso con la arquitectura en la que las métricas de calidad de desarrollo del producto software en la arquitectura implantada sean satisfactorias.

En la planificación, debido a los objetivos orientados a la calidad del desarrollo, las pruebas que se realizan en el despliegue y posteriormente durante el seguimiento del proyecto, retroalimentan inmediatamente el proceso de implementación, volviendo a implementar ciertas tareas, la mayoría de configuración, para mejorar el desarrollo de la arquitectura y por tanto del proyecto del producto software. Esto implica, que de esas

6 semanas en el primer proceso completo, en la tercera iteración Scrum, que es cuando se inicia el primer despliegue, ya se está re-implementando paralelamente. Y en cuatro iteraciones Scrum más, según la planificación, se debería estar próximo a la solución de compromiso, resultando los dos meses posteriores a las dos iteraciones iniciales. Se obtiene así un horizonte temporal de tres meses de proyecto, lo que significan 6 iteraciones Scrum para implantar la arquitectura de desarrollo ágil.

Dicho de otra forma, la implantación de una arquitectura ágil, ya integrada en el propio desarrollo ágil de un proyecto software no deberá suponer más de 6 iteraciones de dos semanas con un alto grado de calidad de compromiso.

Una vez establecido el tiempo de implementación en la metodología se pasará a ejecutar las tareas según lo planificado. Así debe implementar de forma planificada la arquitectura y posteriormente desplegarla, de forma que la implementación se adapte adecuadamente al equipo de trabajo del proyecto software y posteriores trabajos.

4.1.3. Implementación

Tras la fase de planificación en este proyecto de implantación de una arquitectura de desarrollo ágil, se debe ejecutar tal y como se han determinado aquellas tareas proyectadas. Así, se puede decir que la fase siguiente a la planificación sería la ejecución, normalmente entendida en la teoría básica de proyectos como la siguiente en la secuencia natural.

Esta fase de ejecución de la tareas del proyecto se puede entender en dos fases distintas, bien diferenciadas, y que en proyectos ligados al software tiene sentido separarlas. Son las llamadas fases de implementación y despliegue. Entendidas estas de forma que:

- Implementación se entiende como la aplicación de las tareas planificadas de instalación, configuración y ejecución de las herramientas estudiadas y seleccionadas a tal

fin.

- Despliegue se entenderá como la puesta en funcionamiento de la arquitectura como tal, dejando toda la arquitectura lista para desarrollar, construir y ejecutar el código del proyecto de desarrollo software.

Evidentemente no se puede realizar un despliegue sin haber implementado con anterioridad, por lo que tras la planificación la fase posterior es la implementación.

¿Qué tareas o grupo de tareas entonces componen la fase de implementación? Tal y como se ha determinado en la planificación, los grupos de tareas del número 1 al 11 componen todas esas tareas de configuración, instalación, preparación, integración de herramientas, y por tanto de implementación, que son necesarias.

No se puede olvidar en esta fase de implementación que enfocando el uso hacia una arquitectura ágil, se debe tener en cuenta en las tareas que no hay una secuencia estricta de ejecución, sino que será óptimo iterar las acciones en las instalaciones y configuraciones para cumplir el objetivo de la implementación. Entonces es importante no perder el foco en esta fase, por lo que recapitulando, los objetivos importantes en esta fase son:

- Preparar equipos y servidores de la arquitectura para su uso
- Dejar integradas para su funcionamiento las herramientas seleccionadas
- Conseguir un alto grado de automatización para el posterior despliegue
- Tener disponible la documentación necesaria para despliegue y uso del software y herramientas
- Cumplir las especificaciones de requisitos de la arquitectura

Teniendo en mente estas metas de la fase, las tareas deben ejecutarse de forma que ninguna suponga un cuello de botella para el desarrollo de otras. Si no se olvida la agilidad

del desarrollo se debe distinguir perfectamente aquellas tareas que son independientes de otras y las que dependen del desarrollo de tareas anteriores o posteriores. Entonces, tal y como se vio en la fase de planificación, la ejecución, y más específicamente la implementación, debe tenerse en cuenta según la consecución de sus distintas acciones.

Es decir, según las fases de planificación y diseño no se puede integrar la arquitectura si no se parten de instalaciones anteriores de las herramientas de desarrollo y las plataformas adecuadas. Pero además, la configuración de éstas deben enfocarse a un entorno de integración continua, por lo que al integrar se puede volver a configurar dichas herramientas y plataformas. Desde otra perspectiva se podría decir que la integración es crítica para el cuello de botella en la implementación.

Según la figura 4.4 la instalación de los repositorios de código, las herramientas de programación y los repositorios de artefactos o librerías para el desarrollo serán una base necesaria para la integración, pasando por una etapa de configuración de las herramientas y plataformas. Pero una vez realizada la integración y configuradas las tareas para la integración del código se entra en una iteración que hará reconfigurar ciertos aspectos de las plataformas y herramientas. Es decir, en la implementación es importante saber, que además de iterar para implementar, la integración supone el cuello de botella en las iteraciones para dejar implementada la arquitectura para su posterior despliegue.

Siendo una arquitectura de desarrollo que está centrada en la calidad y por tanto en la Integración Continua es de esperar que la parte de integración de herramientas, plataformas, servidores y el propio servidor de Integración Continua sea crítico y por tanto un punto en el que poner especial atención a la hora de implementar.

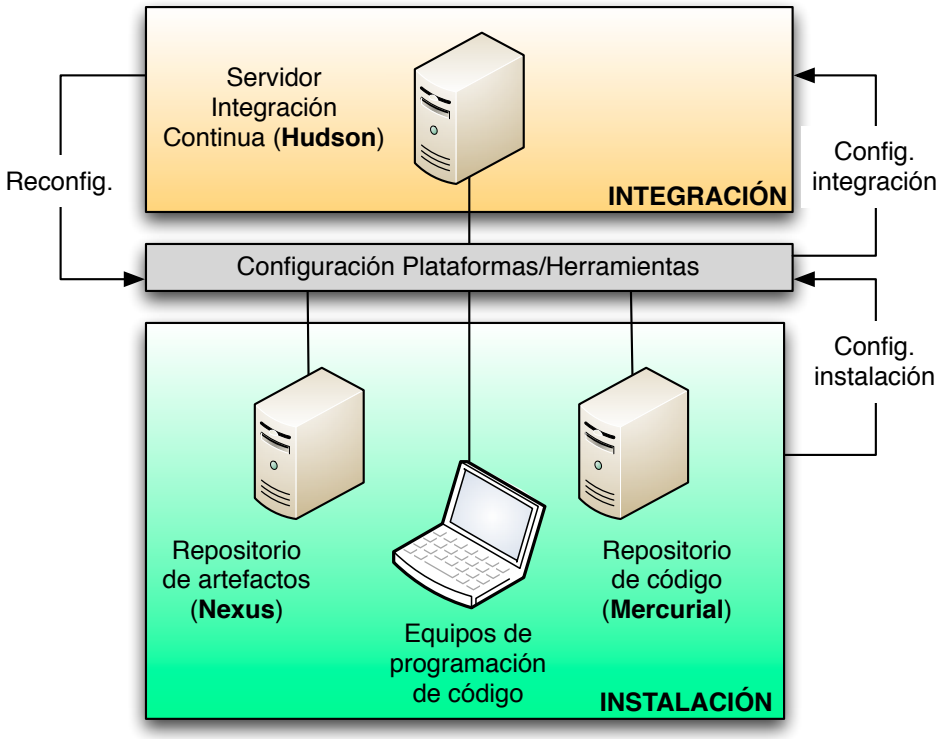

Figura 4.4: Lógica de implementación. Instalación-Configuración-Integración.

4.1.4. Despliegue

Una vez desarrollada la implementación del diseño y la planificación de la arquitectura de desarrollo, llega una de las fases más tangible de todo el proyecto, que es el despliegue de la arquitectura. Es decir, una vez configuradas e instaladas las diferentes herramientas, aplicaciones y plataformas, se necesita poner en funcionamiento los distintos elementos para poder entrar en la programación de productos software mediante metodologías ágiles.

Un despliegue como este tiene como objetivo:

- Poner en funcionamiento las herramientas
- Iniciar la integración global de la plataforma de la arquitectura
- Resolver las dependencias no resueltas en la implementación (iterar implementando)

- Comprobar el éxito de la automatización en la implementación.

Es interesante ver cómo el despliegue está muy ligado a la implementación, formando conjuntamente aquella fase de ejecución de la que se ha hablado con anterioridad y la cual es uno de los objetivos importantes de todo proyecto: *"foco y ejecución"* [2].

Hay que hacer hincapié también en cómo el despliegue tiene como objetivo limar los problemas derivados de la fase de implementación. Esto es así por el hecho de desarrollar el proyecto de forma iterativa y de buscar la calidad en la propia ejecución de las fases, lo cuál inducirá por defecto un mayor éxito en la calidad del desarrollo del proyecto software que utilizará la arquitectura implementada. Buscar la propia iteración en las fases es una práctica ágil, pero se puede cometer el error de no hacer las cosas correctamente por el hecho de pensar que se pueden solucionar problemas en la siguiente iteración.

El despliegue, por tanto, tiene el riesgo de no acabarlo correctamente si se piensa que en volver a realizar tareas de implementación que mejoren una posterior iteración en el nuevo despliegue. Igual que se viene diciendo que la acción de iterar en el desarrollo de un proyecto es otra buena práctica, también hay que decir que un exceso de iteraciones por una incorrecta planificación puede ser un error y llevar incluso al fracaso de un proyecto de este tipo, o de cualquier otro proyecto de desarrollo o implementación software.

Volviendo a los objetivos de esta fase de despliegue se deben saber las tareas que implican según sus objetivos.

[2]En un proyecto, sea ágil o no, es vital no perder el foco del objetivo a conseguir y que sin ejecución de las tareas no se llega a él, intentado evitar el riesgo de perderse o dispersarse en las fases de diseño y planificación.

Iniciar funcionamiento de herramientas

Según cada herramienta se deben realizar ciertas tareas de despliegue que inician el primer correcto funcionamiento de cada una de ellas.

IDEs Realizada la instalación de los IDEs en las máquinas del equipo de desarrollo habría que comprobar que los repositorios de código configurados se descargan y se integran en el equipo tal y como debe estar según el diseño y el funcionamiento. La configuración y creación de proyectos de código se debe actualizar con el actual estado del desarrollo de los proyectos software que utilicen la arquitectura.

Maven Con ayuda del IDE se deben realizar las primeras construcciones del estado actual de desarrollo y también la descarga automática de Maven de los artefactos necesarios para este estado. Se debe por tanto comprobar que construye tal y como debe y que las librerías de artefactos son las necesarias.

Nexus La ejecución de Maven en las máquinas de los miembros del equipo con las primeras construcciones descargarán localmente a los repositorios de Nexus los artefactos necesarios. Se debe comprobar que los repositorios, permisos y configuraciones son correctos, además de añadir los nuevos repositorios requeridos con las primeras construcciones de código.

Hudson Ejecución de las primeras tareas automatizadas de construcción y pruebas de código. Se deben analizar los primeros resultados para posteriormente analizar el estado del proyecto y su correcta puesta en funcionamiento.

Integración global de la arquitectura

Las tareas de despliegue que se realizan en esta fase deben conseguir que las acciones esperadas en una metodología ágil funcionen, y que por tanto, la integración global de la gestión del proyecto con los miembros del equipo y las herramientas de uso se hagan de forma óptima para la consecución siguiente de la programación en el desarrollo del

proyecto de software.

Las tareas que entonces conciernen a este objetivo están dirigidas a comprobar que las herramientas de la arquitectura se integran correctamente con las demás variables del proyecto y que tanto los programadores, como los gestores del proyecto, e incluso los clientes, están usando de forma óptima la arquitectura. Si el proyecto de desarrollo se gestiona mediante metodología Scrum se tendría que comprobar que la arquitectura se integra perfectamente tanto con los roles “cerdoçomo con los roles “gallina", suponiendo una agilidad en el desarrollo.

Entonces estas tareas agrupan las acciones de comprobación de uso de herramientas de los programadores, de monitorización correcta de los gestores, y de funcionamiento de los canales de comunicación con el cliente.

Resolver dependencias

Por norma general el primer inicio y arranque deja a la vista la mejora de la integración de herramientas y de la plataforma, por lo que las comprobaciones de la integración y del inicio de las distintas herramientas software crea nuevas tareas de configuración en las instalaciones e incluso algunos cambios en la planificación actual.

Lo que conlleva estas tareas es creación de nuevas tareas de implementación, y por tanto su nueva planificación. Es también usual tener que replantear la secuencia de algunas tareas anteriores y en algunos casos aislados incluso puede llevar a replantear la correcta instalación de las herramientas y tener que reinstalarlas desde otra planificación distinta.

Las tareas con el fin de resolver las dependencias son aquellas que iteran la reimplementación y que al fin y al cabo agrupan tareas de configuración y automatización

nuevas.

Comprobar el éxito de la automatización

En el despliegue es tarea básica comprobar que efectivamente la implementación se ha realizado tal y como debe y que cumple con los objetivos del desarrollo del proyecto. En este proyecto en particular, donde la Integración Continua y la calidad del desarrollo son objetivos esenciales, la correcta automatización es clave. Esta automatización viene configurada a alto nivel desde el servidor de Integración Continua y a más bajo nivel con programaciones específicas para la configuración de ciertas herramientas o plataformas, ya sea a través de *scripts*[3], de programas más complejos, o de configuraciones automáticas de software.

Para comprobar estas automatizaciones simplemente hay que "monitorizar" las configuraciones siguientes:

- Comprobar gráficos y tests de Hudson
- Visualizar *logs* de servidores de aplicaciones, *scripts* y otros servidores
- Comprobar sistemas de notificaciones de las herramientas
- Reconfigurar seguimiento de tareas de automatización.

Es importante en estas tareas los tiempos de ejecución y la secuencia de tareas de automatización para plantear nuevas configuraciones y la paralelización de tareas.

4.2. Seguimiento

En todo proyecto, ya sea ágil o no, o como este, donde el resultado final tenga que adaptarse a una metodología ágil, es necesario un seguimiento de todas las tareas, etapas

[3]Pequeños programas que automatizan secuencias de comandos o acciones software

y fases desarrolladas. El seguimiento de un proyecto siempre debe de realizarse desde la primera fase, pero bien es cierto que el seguimiento tangible no empieza a desarrollarse hasta que se realiza el primer despliegue, o dicho de otra forma, una vez se ha implementado satisfactoriamente en la primera iteración de la implementación.

Es decir, desde el momento que se estudia el diseño de la arquitectura de desarrollo ya se debe realizar un seguimiento de todo lo que se está haciendo para poder iterar desde el primer momento, pero sin realmente sacar conclusiones materiales verificadas de lo planificado y diseñado hasta que se instala la primera herramienta o se escribe la primera línea de código o configuración. El seguimiento desde el punto de vista de la ejecución de las tareas del proyecto siempre comienza en la segunda iteración del desarrollo del mismo.

Se entiende entonces por seguimiento a las acciones de control sobre todas las tareas ejecutadas en cada una de las fases y etapas del proyecto de implantación de la arquitectura. Precisamente la agilidad del proyecto que se desarrolla y del que se va a desarrollar con la utilización dicha arquitectura hará más sencillo tomar decisiones de control y "monitorización" para un correcto seguimiento de todas las acciones y decisiones.

Para este seguimiento del proyecto se van a diferenciar dos tipos distintos de seguimiento, parecidos a los que se introdujeron en la sección 1.3, igual de importantes, pero que requieren distintos puntos de vista, y por tanto, distinta tipología de acciones y roles participantes. Se distinguirán entonces entre las fases de no ejecución y las fases de ejecución

4.2.1. Fases de no ejecución

Estas fases de no ejecución son Estudio/Diseño y Planificación. Estas fases componen tareas que van a provocar una ejecución de otras basadas en ellas. Por decirlo de otra forma son tareas estratégicas más que ejecutivas, entendiendo entonces que la acción de una buena estrategia conlleva a una mejor ejecución.

Las preguntas que surgen del seguimiento y que ayudan a realizarlo de forma correcta son:

- ¿Cómo controlar las tareas y quién realiza el control?
- ¿Se están escogiendo las acciones adecuadas?
- Algo no funciona, ¿qué debo hacer?
- ¿Cuáles son los indicadores?

En las fases de no ejecución entonces se debe tener muy claro que lo que se está haciendo es desarrollar una estrategia y un plan con el fin de implementar y desplegar una arquitectura de desarrollo para programar bajo un entorno ágil (dirigido mediante metodologías ágiles).

Para contestar la primera pregunta, teniendo en cuenta la fase de no ejecución, se sabe que las tareas están directamente relacionadas con la gestión del proyecto y por tanto lo que se debe hacer es controlar que los elementos de gestión son los correctos. Es decir, mientras se planifica y se diseña se debe comprobar continuamente que los requisitos quedan claros, que la comunicación con los miembros del equipo y con el cliente es fluida y clara, y saber en todo momento de los recursos disponibles. Por tanto, las acciones de seguimiento son:

1. Estudiar requisitos en cada tarea de diseño y planificación
2. Establecer otros canales de comunicación si es necesario y no perder contacto
3. Analizar la disponibilidad de recursos en cada planificación
4. No perder el foco. Preguntarse en todo momento cuál es el objetivo.

Son tareas de gestión, por tanto el seguimiento en todo caso debe realizarse por parte del gestor o los gestores del proyecto. Además, por la naturaleza ágil del uso del "producto"

final del proyecto desplegado, es importante implicar al cliente (y usuario) en este caso en el seguimiento de estas fases de no ejecución. Es decir, si es importante saber que en todo momento se están capturando correctamente los requisitos, nadie mejor que quien los establece para verificar que se están entendiendo, y por tanto se entiende el objetivo a atacar con la consecución del proyecto.

¿Se están escogiendo las acciones adecuadas? Es la gran pregunta que se debe resolver para sacar rendimiento al seguimiento del proyecto. Reformulando la pregunta sería: ¿el diseño y planificación cumplirá con el objetivo? Y esta otra forma de hacerse la pregunta resuelve la acción de seguimiento a tomar, que es no perder foco y tener en mente la claridad del objetivo del proyecto, al igual que una de las acciones de la primera pregunta.

Algo no funciona, ¿qué debo hacer? La respuesta se intuye por la naturaleza del proyecto: Iterar. Si algo no funciona debo retomar el problema y volver a la tarea tomando las decisiones pertinentes en función del problema causante. La acción de seguimiento no es atacar el problema en sí, sino tener claro en todo momento cómo actuar y sobre todo no dispersarse en las tomas de decisión.

¿Cuáles son los indicadores? En este caso son todos indicadores de gestión. Grado de implicación de los recursos involucrados, medir la redundancia de requisitos en los documentos, medición de la satisfacción en la comunicación con el cliente, indicación de no entendimiento en los distintos canales de comunicación, etc. Serán indicadores que dependerán enormemente de las personas implicadas y que además se deben construir y medir según se realicen las fases de no ejecución y su seguimiento. En este sentido debe ser ágil y adaptable a los cambios de forma eficaz. Es importante que los indicadores sean efectivos para poder analizar si las cosas se realizan de la forma correcta.

Al estar actuando sobre una etapa estratégica del proyecto el seguimiento que se realiza en esta etapa de "no ejecución" es un seguimiento en la profunda gestión del proyecto, más

allá de tareas medibles cuantitativamente o de ejecuciones materiales. Una mala gestión en estas primeras fases de diseño y planificación puede suponer un fracaso en las siguientes de implementación y despliegue.

4.2.2. Fases de ejecución

Una vez que la etapa estratégica deja paso a la táctica se llega a las fases de ejecución propiamente dichas, que son las fases Implementación y Despliegue. Estas fases sí ejecutan tareas tangibles y como se explicó anteriormente incluso la línea de separación entre las dos es muy delgada, pudiendo unirlas en una sola fase de ejecución propia como tal.

Las preguntas anteriores utilizadas para el seguimiento de las fases anteriores de no ejecución son las mismas. Pero se debe tener en cuenta que en estas fases existen dos vertientes distintas, un punto de vista de gestión y otro de ejecución de tareas. Desde el primero las acciones de seguimiento son muy parecidas a las anteriores y desde el punto de vista ejecutor hay que responder de nuevo las preguntas.

¿Cómo controlar las tareas y quién realiza el control? Al estar tratando con tareas de instalación y configuración de herramientas software el control se debe realizar a través de monitorización de las ejecuciones y mediante grabación de *logs* o registros de lo que se está ejecutando en código. También se debe documentar las tareas que se ejecutan en el momento de la propia ejecución con el fin precisamente de un seguimiento claro y con el fin de mejorar las siguientes iteraciones. Este control, por tanto, se debe realizar por las propios instaladores encargados de la tarea y además los documentos deben ser revisados desde la gestión del proyecto y por el propio cliente o usuario involucrado mediante la metodología ágil.

¿Se están escogiendo las acciones adecuadas? Las acciones son adecuadas si partiendo de las planificadas se observa que las tareas cumplen con el objetivo. Si desde el punto de vista de la gestión es vital no perder foco, en la parte de ejecución se debe observar

entonces que las acciones de instalación y configuración están acercando el objetivo. Las acciones de ejecución son las adecuadas si las instalaciones son efectivas y las herramientas instaladas cumplen con su objetivo. Esto será más visual en la parte de despliegue que en la de implementación.

Algo no funciona, ¿qué debo hacer? La respuesta es la misma que en las fases de "no ejecución", es decir, iterar. Pero la diferencia en estas fases está en que además la iteración puede llevar a realimentar el sistema en fases anteriores a las de ejecución, como modificaciones en la planificación o incluso en el propio diseño.

¿Cuáles son los indicadores? Además de los indicadores globales de la gestión, que maduran conforme al propio desarrollo del proyecto, o dicho de otra forma, el seguimiento también entra en el proceso iterativo, se deberán tener indicadores para las tareas de instalación de las herramientas que se están ejecutando. En este caso los indicadores para el seguimiento de las tareas son los errores que aparecen en los registros de instalación, los datos de carga en el rendimiento de los equipos a instalar y las estadísticas que se obtienen en los servidores de Integración Continua.

Estas acciones de seguimiento durante las tareas de instalación, configuración y la propia comprobación en el despliegue deben realizarse de forma continuada sin que afecten al trabajo vital de las tareas que se están desarrollando en ese momento. Es decir, las acciones de seguimiento de las propias tareas de ejecución deben ser intrínsecas a la propia tarea. Así, por ejemplo, para una tarea de instalación de Maven, las comprobaciones que se van realizando durante su instalación pertenecen a la misma tarea de instalación en sí misma, y lo que va a marcar como una tarea distinta será el seguimiento por parte de la gestión y la documentación de la misma. Por tanto, el gestor del proyecto es quien debe comprobar que esto se está realizando de forma correcta, mientras que la persona o personas que ejecutan la instalación realizan el seguimiento intrínsecamente.

Se comprueba también que las tareas de comprobación realizadas en la fase de despliegue pueden considerarse también tareas de seguimiento, pero como se dijo anteriormente son tareas intrínsecas a las propias tareas de las fases de ejecución. El seguimiento, ahora entendido como una tarea específica a controlar que todo se ha realizado de forma correcta en las fases anteriores, es entonces como se planificó en el diagrama de Gantt orientativo de la figura 4.3.

El seguimiento, por tanto, es una fase que no se considera como tal en el proyecto, pero que es vital para el correcto desarrollo del mismo, ejecutándose durante todas las fases especificadas. Es importante que el seguimiento del proyecto sea un hábito en su desarrollo, induciendo de forma eficaz a la detección más rápida y ágil de los problemas, y convirtiendo así el proceso iterativo en acciones más naturales en su consecución.

Capítulo 5

CASO DE ESTUDIO

5.1. Desarrollo de un ERP para sector específico

En los capítulos anteriores se ha diseñado, planificado y estudiado cómo ejecutar, implementando y desplegando, una arquitectura de desarrollo enfocada al desarrollo software mediante metodología ágil. Pero una vez especificado cómo se debe gestionar y desarrollar el proyecto se estudiará un caso donde se realice la implantación, de forma que se puedan sacar conclusiones, aplicaciones y líneas de trabajo, con el fin de obtener los resultados esperados, mejorar y perfeccionar el proceso de este tipo de implantaciones.

Se va a analizar la implantación de una arquitectura de desarrollo en una empresa dedicada a la programación de aplicaciones, para producir un software empresarial dirigido a un sector específico y que pretende cubrir las necesidades de la planificación de recursos de las empresas del sector. Con el proyecto de desarrollo se pretende mejorar las soluciones actuales y buscar una evolución en el software actual de dicho sector, que se adapte a las nuevas tecnologías de la información y que suponga un avance en las soluciones informáticas de las empresas con el mismo fin.

El desarrollo software que se va a analizar para la implantación de su arquitectura de programación del producto es un proyecto de creación desde cero de un Planificador de

Recursos Empresarial de nicho, es decir, un ERP muy enfocado a un sector muy vertical y específico del negocio a tratar.

En definitiva, y es importante tenerlo en cuenta, este caso de estudio analiza un proyecto de desarrollo de una aplicación software empresarial que trata la gestión de la información como base del negocio.

La *Gestión de la Información* es una parcela muy importante de cualquier sector y tratada en sí como un negocio propio. Por esta razón la llegada del software en la gestión y procesado de la información ha sido un hito que ha producido la mejora en la optimización del flujo de proceso de la información de las grandes empresas. Hace tiempo se descubrió que la forma en la que se gestionaba la información tenía una implicación enorme en el negocio de las empresas, sobre todo tras la llegada de la informática y el software empresarial a los distintos sectores de negocio. Los grandes esfuerzos en muchas grandes empresas ha sido mejorar mediante el uso de software este flujo de información y su gestión.

Tal y como se puede ver en la figura 5.1, la información suele ser un bien intangible, ya sea abstracto o estructurado, que está almacenado de forma dispersa y que por medio del conocimiento se gestiona para obtener resultados de su valor añadido. A través de la organización esta información se procesa físicamente o mediante herramientas software más o menos complejas, ya sean planificadores de recursos (ERP), gestores de clientes (CRM), de contenidos (ECM), etc. Tras el procesamiento de la información se obtienen unos resultados, que pueden volver a ser editados (realimentan el proceso de gestión) o no editados, obteniendo el bien buscado y volviendo de forma procesada al medio donde se almacene, ya sea para salvarla de forma histórica o para servir como base para otro flujo de proceso distinto.

Todas las herramientas software de gestión de la información siguen este flujo global

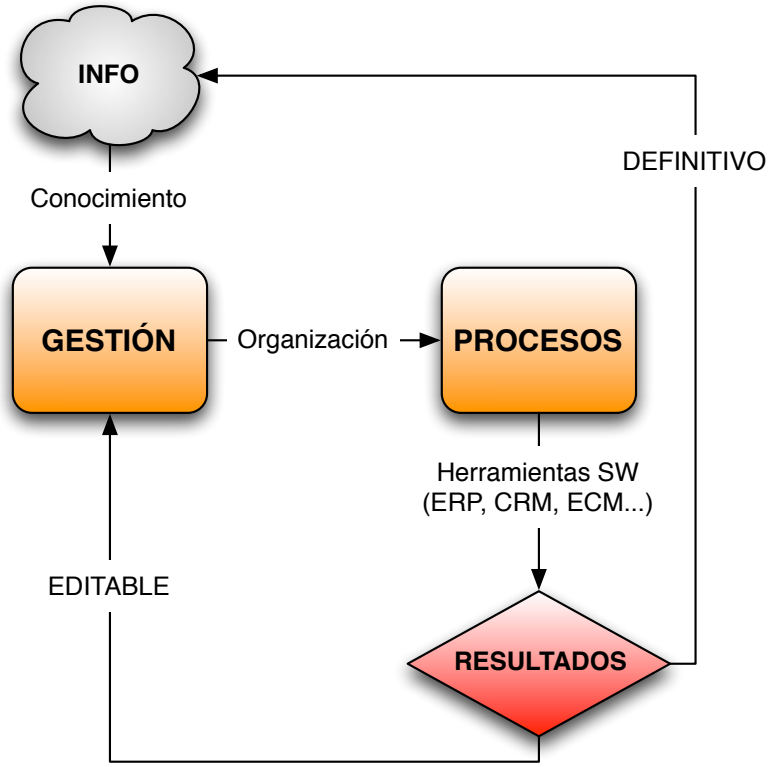

Figura 5.1: Procesos de Gestión de Información.

con el fin de obtener valor como parte de procesos más complejos utilizados por las empresas, organizaciones o individuos y cumplir con sus distintos objetivos.

El caso que aquí se va a estudiar ocupa a un fin económico, es decir, una empresa dedicada a la producción de software que sirve como herramienta de Gestión de Información para la obtención de un beneficio económico en el entorno empresarial utilizado. Es más, la gestión de la información es el propio núcleo del negocio del usuario final de la herramienta.

5.1.1. Negocio objetivo del producto software

El proyecto de desarrollo software que utilizará la arquitectura según este proyecto de implantación consiste en una herramienta ERP específica para el negocio *Transitario*[1]. Dicho proyecto trata de crear el ERP para el sector desde la base, es decir, sin partir de ningún producto similar de otro negocio o de adaptar cualquier otra herramienta global, con el fin de sustituir algunas herramientas actuales del sector y complementar otras.

Para entender la complejidad de la herramienta a desarrollar y programar, y por tanto la arquitectura de desarrollo sobre la que se trabajará, hay que entender el negocio objetivo y la finalidad del software. En este caso un producto de estas características debe planificar de manera eficaz todos los recursos implicados en el negocio de los transitarios y gestionarlos para operar de forma eficaz en el negocio. Así, las tareas a las que una empresa dedicada a este negocio desarrolla son:

- Gestión de clientes
- Operaciones de tráfico de mercancías
- Gestión logística
- Facturación de ventas y operaciones
- Gestión aduanera
- Gestión documental
- Gestión contable

Todos estos servicios son ofrecidos por los distintos transitarios a sus clientes y a la vez los clientes, que suelen ser en ocasiones otros transitarios, utilizan sus planificadores

[1]Brevemente, se define como el sector de negocio consistente en la gestión de cualquier transacción relacionada con mercancías, internacional o nacional. Se suele definir como *Transporte Internacional Multimodal*

de recursos para gestionar los suyos propios.

El negocio Transitario es en realidad un negocio bastante complejo, ya que en el momento en el que se realiza un movimiento de mercancía, información o cualquier actividad que comprenda movimiento o comunicación de un punto geográfico a otro, estas empresas entran en la oferta de servicios del negocio en cuestión. Para entender de forma más práctica los servicios que ofrecen estas empresas se puede pensar en los grandes operadores logísticos, que son los que actualmente forman el núcleo fuerte en este sector y que realizan la mayoría de operaciones y servicios demandados. También las empresas de transporte pueden ser pequeños transitarios, pero dedicados más específicamente a la gestión del tráfico de las mercancías.

Se puede comprobar entonces que la complejidad del negocio es enorme desde el punto de vista de gestionar todos sus servicios de forma central mediante una sola herramienta software, o relacionando de forma efectiva distintas soluciones para fines más específicos. Además, el sector no tiene unos flujos de proceso definidos y acotados por sí mismos, sino que la forma de trabajo y por tanto el funcionamiento de anteriores herramientas, está diseñado según la experiencia dispersa de los distintos clientes en el sector. También ciertos servicios del negocio, como puede ser la gestión aduanera son muy estrictos bajo una normativa muy estricta y que además está en continua evolución.

Bajo el conocimiento del sector y las necesidades de los clientes para el desarrollo de la solución, queda muy claro que la problemática del negocio para gestionar de forma eficaz sus recursos y servicios es muy compleja y las actuales herramientas software que las gestionan son muy pesadas, complejas de uso, tecnológicamente obsoletas y con una proyección de utilización bastante incierta. Estas razones son las que han llevado a los clientes a demandar una nueva herramienta que cubra más eficientemente las necesidades de los transitarios y que se adapte de forma más flexible al trabajo, que además evoluciona muy dinámicamente en la actualidad.

Entonces, las necesidades y problemática del negocio Transitario muestran que el desarrollo de una herramienta como tal es un proyecto de gran envergadura y que es crítico para el sector en cuestión. No sólo es muy importante cubrir al ciento por ciento las necesidades de los usuarios, sino que hay que ir más allá en este tipo de desarrollos y pensar en la evolución que puede tomar la herramienta para no suponer un fracaso en su lanzamiento, ya que el desarrollo de la programación supondrá un intervalo de tiempo largo.

5.1.2. Participantes en el desarrollo

Introducidas de forma breve las necesidades del negocio y qué debe cubrir una aplicación empresarial de estas características se debe saber quién va a ejecutar el proyecto de desarrollo software (proveedor), quién demanda la solución (cliente), los recursos de los que se disponen, el objetivo que se quiere cumplir y las razones que han llevado a la empresa a desarrollar una herramienta de estas características.

En el caso de estudio existe un proveedor de soluciones para transitarios y un implantador del software de la herramienta actual, la cual ha sido desarrollada hace 20 años por el proveedor de soluciones. Según la figura 5.2 el proveedor entrega la solución software al implantador para que despliegue la solución en el usuario transitario que demanda la solución. Pero el usuario, después de mucho tiempo utilizando la misma solución y sus actualizaciones demanda una nueva solución que cubra de forma más eficaz sus necesidades transmitiendo una necesidad de cambio en el software.

Tras los participantes habituales en la solución, el proveedor, debido a la demanda de sus clientes decide comenzar un proyecto de envergadura para crear una nueva solución, para lo cual demanda una nueva aplicación a la empresa de desarrollo, que creará un nuevo ERP, el cual será implantado y desplegado al cliente por la empresa de implantación.

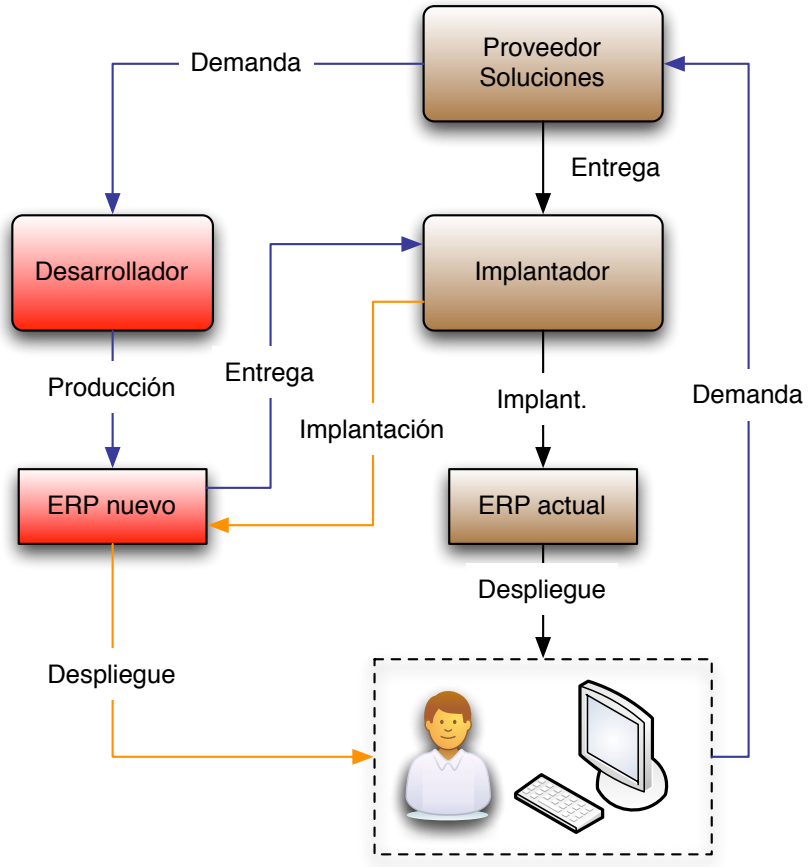

Figura 5.2: Proceso de demanda de nuevo ERP Transitario.

Los actores en el proyecto son entonces 4:

1. Proveedor
2. Desarrollador (productor)
3. Implantador
4. Usuario

Pero en este caso, el proveedor e implantador pertenecen al mismo grupo de proveedores de soluciones para clientes transitarios, y en definitiva, el proyecto de desarrollo

del ERP del caso de estudio se reduce a tres participantes desde el punto de vista del proyecto: *Cliente* (proveedor e implantador), *Productor* (desarrollador) y *Usuario*.

Es decir, un cliente que necesita un nuevo producto software para implantar, demanda el ERP a la empresa de desarrollo software, más específicamente una factoría de software. Ésta es quien entonces necesita la arquitectura de desarrollo para poder ejecutar sus proyectos, y además, el caso estudiado es el de dicha factoría de software, la cual es creada en el momento que se demanda el ERP. Por tanto, no existe una arquitectura de desarrollo software madura anterior que haya que adaptar a este estudio, sino que la implantación a realizar está enfocada para el proyecto particular del ERP y más generalmente para otros proyectos de desarrollo que se van aceptando en la empresa.

Esta empresa factoría de software comienza además a gestionar sus proyectos mediante metodologías ágiles, y más específicamente comienza con una metodología Scrum, como ya se avanzó en capítulos anteriores, por lo que en principio es un caso de estudio perfecto para analizar la implantación.

Los participantes tienen un tamaño y recursos de forma que el grupo proveedores de soluciones es líder en el sector, con aproximadamente 250 empleados localizados en varias capitales de la geografía española; la empresa de desarrollo del software posee 7 programadores, un administrador de sistemas, un gestor de proyectos e ingeniero de calidad y dos directores ejecutivos. Los usuarios pertenecen a la cartera de clientes del proveedor y son cientos de empresas usuarias dedicados al negocio transitario nacional e internacional, de pequeño, mediano y gran tamaño, por lo que la aplicación estará dirigida para el empleo simultáneo de miles de usuarios.

Se concluye por tanto que la solución software, aunque esté dirigida a una mediana empresa por norma general, será lo suficientemente flexible para adaptarse a entornos más grandes y pequeños, dando valor al negocio y siendo eficaz para suplantar a la actual

Tabla 5.1: Parámetros de gestión del proyecto ERP

Variable	**Recursos**
Duración del proyecto	2 años
Programadores	7 (jornada completa)
Gestores de proyecto	1 Scrum Master y 2 directores técnicos
Quality Assurance	1 ingeniero dedicado
Metodologías de gestión	Scrum, Kanban
Metodologías de ejecución	Extreme Programming, TDD

solución software que existe.

La arquitectura de desarrollo a implantar va a tener estas consideraciones para adaptarse de forma eficaz no sólo a la metodología utilizada en la programación, sino también a la calidad del desarrollo de un proyecto de estas características y envergadura.

5.1.3. Gestión del proyecto

El estudio se realiza sobre el desarrollo y programación del ERP para el sector Transitario, realizado por la descrita empresa factoría de software, dedicada al cliente proveedor de servicios. El proyecto de la creación de la herramienta es un proyecto de gran envergadura donde tendrá un tiempo de realización, unos recursos disponibles y unas metodologías de gestión y ejecución determinadas para la programación. En la tabla 5.1 se comprueban los recursos utilizados por la empresa para el desarrollo de la aplicación.

Teniendo en cuenta el horizonte temporal del proyecto y la metodología ágil utilizada, la cual para la implantación de la arquitectura de desarrollo es esencial, se puede determinar que la implantación de una arquitectura adecuada de desarrollo puede suponer un tiempo mínimo de seguimiento de la mitad del tiempo empleado para el desarrollo del

proyecto, ya que al tratar una metodología ágil, en un proyecto de 2 años de duración, el desarrollo no está lo suficientemente maduro en el primer año como para saber si se puede dar por concluido el seguimiento para establecer concluidas las iteraciones. Es decir, aunque según la planificación se pueda tener implantada en tres meses una arquitectura de desarrollo estable para programar la aplicación con la calidad requerida, no es posible sacar conclusiones del desarrollo sobre la arquitectura implantada hasta, por lo menos, pasado el primer año de desarrollo.

El caso de estudio tendrá en cuenta que desde el momento que se inicia la implantación se recogen datos para posterior análisis de resultados y estudio de las conclusiones obtenidas. El seguimiento, por tanto, tras el despliegue de la arquitectura se prolonga hasta el primer año de desarrollo de la solución software a producir.

Metodologías

La gestión mediante metodología Scrum se realiza de la siguiente forma:

- **Iteraciones**: *Sprints* de dos semanas que coinciden con el versionado del software
- **Artefactos Scrum**: El *Product Backlog* y *Sprint Backlog* se crea, modifica y establece en el sistema de seguimiento de tareas de la empresa, mediante la herramienta *Redmine*[2].
- **Roles gallina**: Empresa proveedora de soluciones a transitarios e implantadores, es decir, los clientes, y los *Managers* de la empresa y el proyecto.
- **Roles cerdo**: Se compone por las nueve personas que están involucradas directamente con el desarrollo del proyecto. Siete programadores, un ingeniero de calidad, el cual hace el rol de *Scrum Master*, y el director técnico de la solución.

[2]Herramienta de seguimiento de proyecto utilizada por la empresa de desarrollo y que debe integrarse con la arquitectura a implantar.

- **Reuniones**: Los viernes por la mañana de la semana anterior al inicio de cada iteración se realiza la Reunión de Planificación de Versiones. El mismo viernes por la tarde, pero de la iteración que concluye se realiza la Revisión de Sprint y la Restrospectiva de Sprint. El siguiente lunes, los gestores de proyecto con ayuda del equipo realizan la Planificación de Sprint.

Pero en el caso de estudio la metodología Scrum tan sólo es empleada para poder gestionar de forma eficaz e intuitiva el proyecto, por lo que las iteraciones, que coinciden con versiones del software, al ser ejecutadas mediante *Kanban*[3], no tienen un vencimiento estricto en el tiempo, por lo que algunas iteraciones tienen duraciones de cuatro semanas, y otras de una. Hace que el sistema sea más flexible.

La ejecución Kanban, implantada durante el proyecto de desarrollo tiene en este caso la implicación de establecer un flujo de trabajo secuencial y sin tiempos muertos en el equipo de desarrollo, y no mantener desocupado a nadie en ningún momento. Además, al ser una programación de código enfocada a la calidad y a un estricto control de tests de código, las versiones e iteraciones no son aprobadas hasta que las pruebas se realizan con éxito o con el resultado esperado.

Conforme se van añadiendo funcionalidades y tareas de programación el número de tests que se debe pasar en la ejecución del código y en la integración aumenta de forma notable, por lo que los datos de calidad van siendo más significativos conforme uno de los objetivos de la programación de la herramienta, que es la excelencia en calidad de la programación.

En la gestión del proyecto, además, el cliente está continuamente involucrado, resolviendo en todo momento las dudas por parte del equipo de desarrollo en cuanto a funcionalidades buscadas, en requisitos cumplidos y en indicadores de satisfacción del

[3]Esta metodología no se utilizó al inicio del proyecto, pero se implantó al final del primer año de desarrollo para optimizar la mejora continua

desarrollo. Un hecho interesante del caso de estudio es que una persona experta en el negocio de transitarios, perteneciente al cliente (proveedor de soluciones a transitarios), se incorpora al equipo de trabajo para mejorar la comunicación en el análisis funcional y su ejecución en la programación.

Arquitectura de desarrollo

Inicialmente en la gestión del proyecto se tienen en cuenta los recursos disponibles para la programación y desarrollo de la aplicación, así como los recursos disponibles para la comunicación y seguimiento del proyecto por parte del cliente.

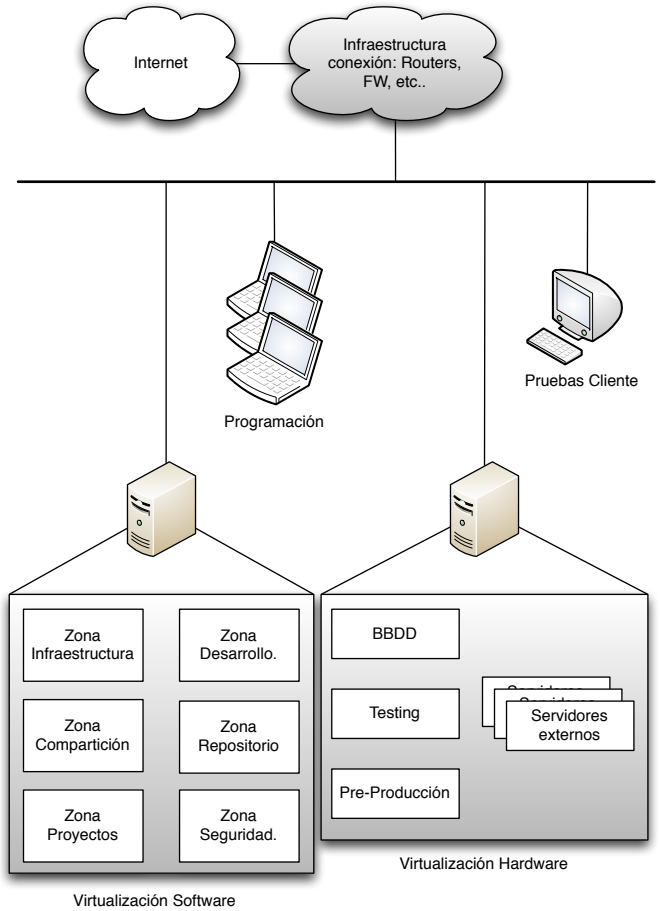

Figura 5.3: Infraestructura de la empresa de desarrollo.

En la figura 5.3 se observa cómo se conforma la infraestructura de la empresa en cuanto a recursos materiales informáticos para la consecución del desarrollo. Sería la base material sobre la que se implantará la arquitectura, teniendo en cuenta además que la empresa de desarrollo ha apostado por virtualización de máquinas para la instalación de distintos elementos necesarios durante el desarrollo de cualquier proyecto. A partir de esta arquitectura, ya instalada en la empresa al inicio de su creación según sus necesidades de servicios internos y su negocio empresarial, se instalarán las herramientas estudiadas según las fases del proyecto en la implantación.

La gestión para implantar de forma efectiva toda la arquitectura de desarrollo según la infraestructura de recursos de la empresa, encaja con las metodologías utilizadas por la empresa para el desarrollo del software, y se realizará mediante las iteraciones oportunas del desarrollo siguiendo la planificación formada en la sección 4.1. Es decir, la implantación de la arquitectura de desarrollo no va a cambiar ningún elemento físico en la programación del proyecto software, sino que se va a adaptar de forma que optimice su desarrollo y mejore la consecución del mismo. Ese es el objetivo de utilizar una arquitectura de desarrollo como la que en este proyecto fin de carrera se estudia.

5.1.4. Arquitectura del software a programar

El proyecto que se va a llevar a cabo mediante esta arquitectura de desarrollo es complejo y de gran envergadura. Pero para hacerse una idea del alcance conviene saber cómo se va a estructurar su programación, qué elementos necesitan ser programados y qué tecnologías son utilizadas.

Con la información de la que se dispone[4], a grandes rasgos, la aplicación a desarrollar pretende cumplir con el flujo de proceso de gestión de la información. Si se revisa la figura 5.1, la forma en que este ERP cubre el funcionamiento, y por tanto, formando la

[4]Por razones de propiedad de la empresa no se pueden mostrar todos los datos.

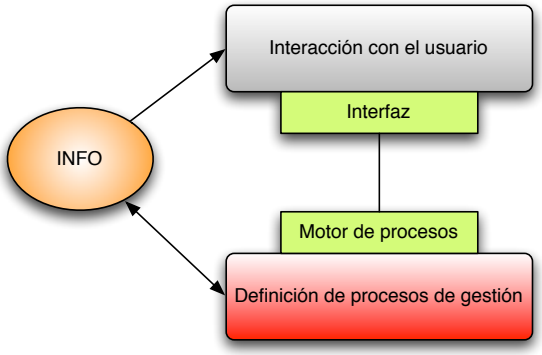

Figura 5.4: Arquitectura estratégica del ERP.

arquitectura global de la aplicación, se determina según la figura 5.4.

El diagrama muestra que la información es gestionada mediante un interfaz por el usuario, obteniendo como resultado la definición de los procesos de gestión que modela el motor de procesos de la aplicación. La relación que tiene la definición de procesos con la información es bidireccional, es decir, la propia definición provee de información para entrar en otro flujo de proceso de nueva información, y aquella también provee a la definición de procesos como motor de procesamiento para otros procesos más complejos.

La forma en la que el ERP a programar de este caso de estudio afronta la manera de gestionar la información es mediante un modelo cliente-servidor, donde el cliente instalado en la máquina del usuario es el *Front End*, representado en la figura 5.4 como el interfaz (es en realidad el interfaz con el usuario), y la entidad servidor es el denominado *Back End*, donde se almacena toda la información y además se ejecuta el modelado de la definición de procesos. La programación de este modelo se enfoca en el caso de estudio hacia una alta calidad global que debe ser percibida por el usuario final. Por tanto, la parte del *Front End* debe reflejar de forma fiel no sólo una buena manejabilidad y apariencia en el interfaz, sino que el *Back End* que existe por debajo debe hacer funcionar el motor de procesos y datos de la forma más óptima para que el trabajo que demanda el usuario se realice de la manera más efectiva posible.

En término tecnológicos y según la arquitectura del ERP, la aplicación se estructuraría según la figura 5.5.

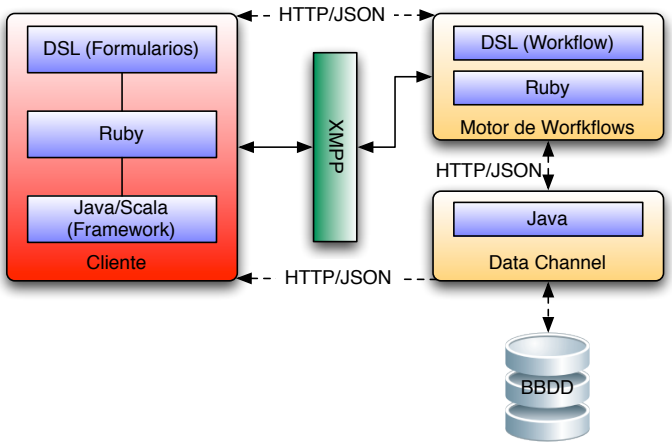

Figura 5.5: Arquitectura tecnológica del ERP.

Pensando entonces en la implantación de la arquitectura de desarrollo ya se puede visualizar una necesaria integración entre el mencionado cliente (*Front End*) y el servidor (*Back End*), por lo que la Integración Continua del desarrollo ya visualiza un claro elemento global de integración que es crítico. Además, si se atiende a las tecnologías, plataformas y lenguajes descritos en la figura se puede intuir que la configuración del *Build Lifecycle Tool* o herramienta de construcción del código debe resolver dependencias de forma solvente y transparente al desarrollo global.

Este caso posee una arquitectura compleja, enfocada a resolver un caso complejo de requisitos. Es por tanto vital que una complejidad heredada de las dependencias en el proyecto de programación no sea un lastre para el desarrollo y no induzca fisuras en la calidad del proyecto, de forma que una resolución no clara de las dependencias pueda producir un mayor impacto en la ejecución global de la aplicación. Por ejemplo, al programar los formularios visuales al usuario de la aplicación, si no se resuelven eficientemente las dependencias entre sus componentes programados en Ruby y las librerías necesarias en

Java para visualizarlos, se puede tener un componente que no cumple con los requisitos necesarios para obtener del servidor los datos realmente requeridos. Esto se traduce en una percepción de mala calidad a los ojos del usuario final.

A primera vista parece que el caso de estudio es un ejemplo muy interesante para implantar la arquitectura enfocada a la metodología ágil, ya que es un proyecto con cierto grado de complejidad, desarrollado mediante metodología Scrum y otras técnicas de programación ágil. También, como se comprueba en su arquitectura, la integración continua del desarrollo es un elemento crucial, no sólo desde el punto de vista del código, sino desde la integración global de las tecnologías de la arquitectura y de las partes de la aplicación claramente diferenciadas. Las decisiones tecnológicas que llevan el desarrollo del producto con varios lenguajes y plataformas muy distintos y enfocados a diferentes soluciones de la aplicación hacen todavía más crítica la buena integración entre todas ellas.

Las entregas de la aplicación para ir mostrando al cliente se realizan por módulos desarrollados del ERP, los cuales tienen que integrarse a la perfección en la aplicación por requisitos de funcionalidad del software y también del negocio. La relación de módulos además, no es sólo una relación horizontal y bien definida, sino que las relaciones que deben existir entre aquellos son cruzadas y de forma que el negocio no tiene definidas mediante flujos de proceso. Por tanto, la captura de requisitos funcionales y las definiciones de tareas en el equipo de desarrollo en función de aquellas es crítico para el desarrollo, no debiendo perder el foco en decisiones de construcción e integración del código, tarea que debe cumplir a la perfección la arquitectura de desarrollo y ser completamente transparente al equipo de programación.

La primera decisión entonces desde el punto de vista de la arquitectura de la aplicación a desarrollar es simplificar al máximo la complejidad de las dependencias entre los módulos, ya que una de las razones por las que el software anterior empezaba a dar problemas de escalabilidad y uso es por las malas relaciones entre módulos en forma de *tela de araña*,

Tabla 5.2: Agrupación de módulos ERP

Grupo	**Módulos**
CRM	Clientes, Agenda...
Comercial	Ofertas, Tarifas...
Fin / Adm	Facturación, Contabilidad, Financiero...
Operaciones	Servicios, Tráficos, Logística...
Configuración	Configuración global, Usuarios, Gestión Documental...

donde múltiples caminos de relaciones hacías redundantes multitud de dependencias.

Con esta decisión de simplificación, si se acompaña en la plataforma de desarrollo herramientas capaces de gestionar relaciones y dependencias complejas sin mayor problema, hace el desarrollo del proyecto aún más ágil si cabe. Con esta aproximación los módulos de negocio de la aplicación se agruparon básicamente en los de la tabla 5.2.

Estas agrupaciones, a las que la aplicación llama *"metamódulos"* hacen una visualización mucho más clara de la organización y por tanto integración del desarrollo y así la adaptación a la plataforma y arquitectura de programación será mucho más visible a nivel del equipo de desarrollo.

5.2. Implantación de la arquitectura y pruebas

Una vez propuesto este caso de estudio, un caso real, en el que se ha trabajado cerca de dos años en la implantación y seguimiento de la arquitectura de desarrollo propuesta, se va a exponer el desarrollo de este proyecto, y las pruebas y datos que se han recolectado para probar que efectivamente el proyecto cumple con sus objetivos y funciona correctamente en un estudio como este.

El intervalo temporal estudiado, realizando el proyecto de implantación, su posterior seguimiento y el estudio de conclusiones, comprende desde junio de 2009 hasta agosto de 2010, es decir, durante un año y tres meses aproximadamente se han estado capturando datos y trabajando en modificaciones para adaptar aún mejor la arquitectura a las evoluciones naturales del desarrollo de la empresa en el proyecto. Además, la aceptación de otros proyectos de desarrollo software durante la programación del ERP han llevado a otras pequeñas modificaciones de la arquitectura, demostrando que su adaptabilidad a estos tipos de proyectos es realmente rápida y ágil en sí misma.

En realidad la arquitectura propuesta en la implantación es una base que ha resultado ser muy buena para la reutilización de las herramientas en otros proyectos y tecnologías para las que no fueron diseñadas previamente. El gran ejemplo es Maven, donde una herramienta de construcción pensada en principio para proyectos Java, ha resultado ser muy útil en el desarrollo de proyectos programados mediante otras tecnologías afines. Pero esto se verá posteriormente cómo ha resultado ser.

La implantación de la arquitectura de desarrollo se desarrolla inicialmente al mismo tiempo que la programación del ERP en un estado prematuro del proyecto. Es decir, mientras se diseña la arquitectura de la aplicación ya existe una necesidad de instalación de una arquitectura de desarrollo robusta y estable para estar preparado en la ágil implementación del código de la arquitectura de la aplicación, y empezar a producir módulos y partes distintas.

5.2.1. Implantación inicial y fases

Durante el primer mes de estudio de las herramientas el proyecto de desarrollo estaba en una etapa de proyecto inicial muy poco madura y aún había bastante desconocimiento de ciertos factores del desarrollo que eran determinantes en muchos aspectos. También

mientras se conformaba el proyecto de desarrollo de la aplicación también la empresa estaba en su fase inicial de vida, siendo el propio proyecto de creación del ERP en el sector Transitario el lanzamiento de la empresa hacia el desarrollo de software para este negocio y otros.

La implicación que tiene el inicio de la implantación de la arquitectura de desarrollo según el estado del proyecto del software a programar sobre ella es que las primeras iteraciones de implementación y despliegue se alejan bastante del estado final del proyecto, aunque sus datos, que más adelante se van a exponer, son significativos para comprobar el estado del desarrollo del producto y también su evolución. Además, la propia metodología ágil según la cual se gestionaba el proyecto fue evolucionando hasta dejar la metodología Scrum como una mera guía para la comunicación y gestión del equipo de desarrollo.

Las fases que se han ejecutado durante el proyecto de implantación han sido las definidas en el capítulo 4. Se han ejecutado tal y como se especifican en él y siempre bajo las necesidades de la empresa objetivo, según determinados costes, recursos y planificación. Todos estos parámetros están contemplados en la planificación del proyecto y no son un problema inicial en las primeras fases del proyecto. Hay que mencionar que tecnologías utilizadas en el proyecto de desarrollo del ERP se han desarrollado en conjunción con la comunidad *Open Source*, y como requisito de la empresa de desarrollo para la plataforma y arquitectura de desarrollo es que las herramientas empleadas se adaptaran perfectamente sin problema a las herramientas de código libre.

Por tanto las decisiones durante la fase de diseño de la arquitectura, donde la elección de las herramientas de desarrollo, de Integración Continua y las plataformas, estaban influídas por la característica de ser siempre herramientas maduras y probadas por la comunidad de código libre, se han visto afectadas por ello. Precisamente no supuso nunca un problema porque la elección inicial de las herramientas para todo tipo de desarrollo ágil siempre se ha realizado bajo estas premisas, siendo además esta comunidad la que

más activa ha estado en el desarrollo de este tipo de software.

Diseño y planificación de la arquitectura

En el caso de estudio se muestra que la comunicación entre la empresa ejecutora del desarrollo y el cliente es muy fluida. Se presenta el hecho de que la empresa que desarrolla el software participa económicamente en el grupo de empresas del cliente, facilitando en todo momento los canales de comunicación. Esto supone una ventaja a la hora de la gestión del proyecto mediante la metodología ágil, y en el momento de diseñar la arquitectura sobre la que se va a programar no es un problema establecer la mecánica de comunicación y documentación funcional del código hacia el cliente.

Los elementos físicos y software determinantes para la arquitectura que se encuentran ya pre-instalados en el departamento de desarrollo son los definidos en la tabla 5.3. Es decir, partiendo de estos elementos ya disponibles en el entorno de desarrollo se puede iniciar un diseño como el propuesto en la sección 4.1. Pero no hay que olvidar que estos recursos son necesarios para una arquitectura de desarrollo. Los recursos están dimensionados de forma que algunos son necesarios para la empresa como tal, es decir, para poder arrancar como organización con empleados y cumpliendo las normas de formación empresarial se necesitan parte de ellos; y otros son instalados como parte de una arquitectura para desarrollar proyectos de software.

Los recursos que son parte de la arquitectura son recursos necesarios para implantarla, por tanto se han de contabilizar como parte de la implantación. Si estos recursos no hubiesen existido tendrían que haberse adquirido o según la distinta composición de la empresa se podrían haber estructurado de otra forma.

En este caso de estudio se ha optado por la virtualización como parte del uso de servidores necesarios para conectar las distintas herramientas. La disminución de costes

Tabla 5.3: Recursos iniciales de desarrollo

Entorno	**Elementos**
Equipo de desarrollo	4 programadores *senior*, 3 *junior*
Gestión	Iterativa. Scrum Master y QA responsables
Lenguajes	Java, Ruby, JRuby y Scala
Hardware de programación	7 portátiles Mac. Discos portátiles de *backup*
Servidores	2 Dell de media capacidad
IDEs preinstalados	NetBeans, XCode, ViM
Sistemas Operativos	Windows Vista, Mac OS X, Debian Linux, Oracle Linux, VMWare ESXi, OpenSolaris
Herramienta seguimiento	Redmine Issue Tracker

mediante el uso de virtualización es notable, y ésta produce una independencia de los distintos servidores que facilita la integración del desarrollo en un entorno de Integración Continua, agilizando la automatización, de forma que si algún elemento servidor falla no escala afectando a otras partes de la programación del ERP. Por ejemplo, si el servidor de Integración Continua se para no es ningún problema para poder seguir programando por parte de los programadores, ya que gracias al servicio de repositorio de código el desarrollo no se ve afectado.

Además, el diseño distribuido de los repositorios, tanto de código por parte de Mercurial, como de artefactos Maven por parte de Nexus, también mantiene alejados de problemas de parada de estos servidores, aunque siempre será aconsejable solucionar sus paradas lo antes posible.

Según los recursos mencionados por parte de la empresa de desarrollo y sus metodologías, el diseño propuesto para este caso es el descrito en la figura 5.6, donde si se compara con la arquitectura propuesta en la sección 4.1 se puede encontrar perfectamente la relación entre todos los elementos.

Lo que se consigue con este diseño es que cualquier programador construya eficazmente su código mediante el IDE sin tener que preocuparse de las dependencias que pueda tener su parte de programación con la de otro programador gracias a la gestión de dependencias que realiza Maven y el código actualizado mediante Mercurial, ambos configurados en el propio IDE (en el caso NetBeans y Eclipse). También la gestión de versiones de los artefactos de desarrollo no es un problema, ya que Nexus consigue que todos descarguen los artefactos del servidor centralizado, que además está en la intranet.

En el diseño se observa también que en los entornos de *Testing* y *Pre-Producción* existen unos nodos Hudson. Esto lo único que hace es ejecutar las tareas de Hudson de la integración continua en el propio servidor, sin tener que descargar el código en el servidor

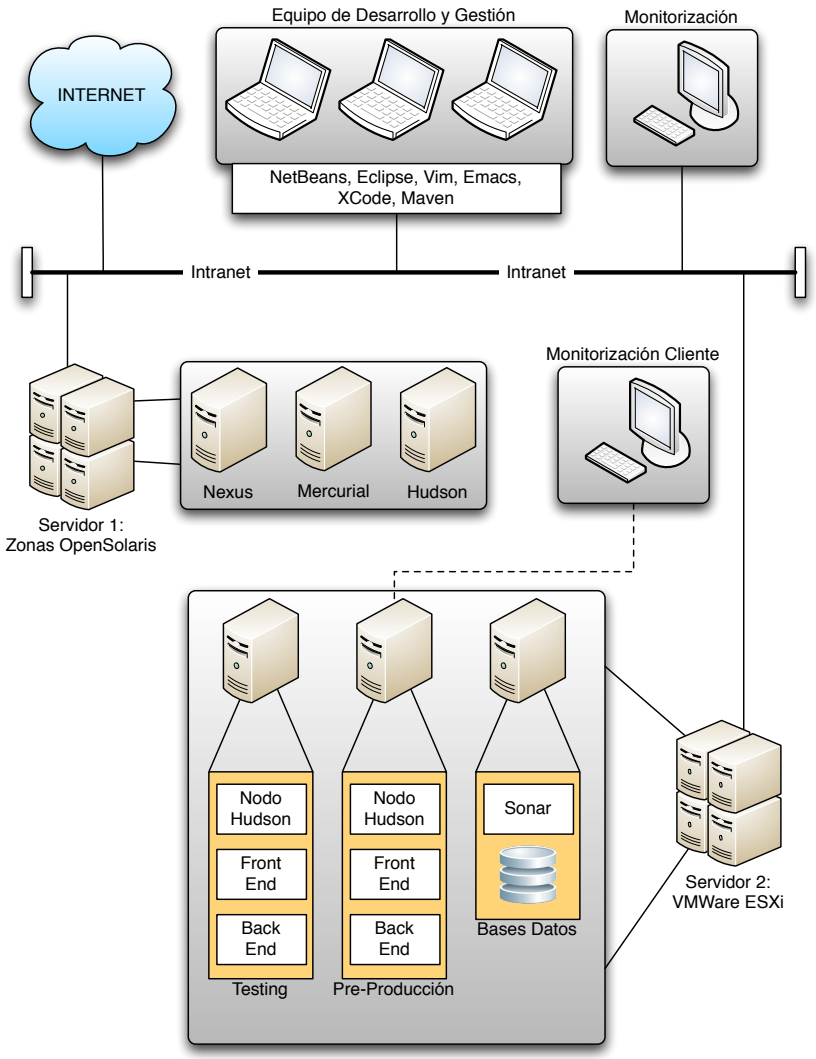

Figura 5.6: Diseño propuesto para el desarrollo del ERP.

central de Integración Continua. Esto es así por el propio funcionamiento de Hudson, permitiendo entonces que en casos como este, donde es necesario, los datos estadísticos de las tareas de automatización se encuentren en el mismo servidor donde se ejecuta el código construido.

En el servidor número 2, según este diseño, también se observa una máquina virtual donde están las bases de datos, y además también se ha instalado un servidor de análisis de código. La herramienta utilizada para este caso es Sonar, referenciada en el apéndice

B.5.

Durante este diseño se estudiaron distintos factores que influyen notablemente en la implantación:

- Preferencias de herramientas de programadores. Algún programador demostró rechazo al uso de IDEs gráficos por su metodología de trabajo, por lo que se automatizó la integración en su caso entre su herramienta empleada, Maven, Nexus y Mercurial.
- Dedicación completa a QA. La figura de Scrum Master es la misma persona que el reponsable de calidad del código, por lo que la integración, responsabilidad de QA, está continuamente vigilada.
- Flexibilidad del cliente. La calidad es la variable más importante para el cliente, demostrando flexibilidad confesa en las entregas de versiones, siempre y cuando las razones de cualquier retraso sean por temas cualitativos del software.
- Equipo multidisciplinar. Varios miembros del equipo de desarrollo poseen conocimiento de distintas disciplinas de tecnología software, por lo que hubo implicación directa en la adaptación y configuración de la arquitectura. Así hace mucho más ágil el posterior uso por parte del equipo.
- Distribución física del personal. La sala de programación donde se encuentran los usuarios de la arquitectura posee la ventaja de tener una comunicación directa y personal entre los miembros usuarios de la arquitectura, por tanto la resolución de dudas tecnológicas a veces se resuelve de inmediato.

Implementación y despliegue

Con el diseño establecido y las características del equipo de desarrollo y del proyecto a llevar a cabo de programación se puede aplicar la implementación especificada en la planificación del capítulo 4, según las tareas detalladas. Se encuentra entonces con tareas de implementación adaptadas a las iteraciones Scrum de dos semanas que se integra con

la propia metodología de las fases iniciales del proyecto de desarrollo del ERP.

Los lunes que inician un sprint de dos semanas, en la reunión *Planificación de Sprint*, se proponen las instalaciones a llevar a cabo al equipo y a los gestores. Con esta reunión la comunicación con el equipo para llevar las instalaciones es mucho más fluida, y se recibe información para establecer las tareas más inmediatas a llevar a cabo durante la iteración con ayuda del equipo de desarrollo y de los propios gestores del proyecto de programación.

Durante las *scrum meetings* diarias se van proponiendo cambios o configuraciones que los programadores van aportando según cambios que van surgiendo en la configuración de las herramientas. En este caso, según se ha avanzado anteriormente, ante la decisión del equipo de desarrollo de poder tener herramientas de programación más básicas, se realizan ciertas configuraciones que pueden automatizar esas herramientas básicas de programación, sin tener que preocuparse de configuraciones extra. Esto se fue viendo en estas reuniones diarias y se adaptó de forma que el programador tuviera más flexibilidad en la elección de la herramienta.

Las tareas que se realizaron durante la primera iteración fueron:

1. Instalación de Maven en todo el hardware cliente y servidor, físicos y virtuales
2. Configuración inicial de Maven mediante el archivo `settings.xml` (configuraciones en el apéndice B)
3. Instalación de NetBeans y Eclipse en cada portátil de programación, según preferencia, e integración con la instalación de Maven
4. Instalación de Mercurial en todas las máquinas exceptuando el servidor Mercurial
5. Instalación del servidor Mercurial
6. Instalación del servidor Hudson
7. Configuración de Hudson para adaptarlo a Maven y Mercurial

8. Instalación de Nexus y adaptación al desarrollo (configuración como proxy de los repositorios actuales)

9. Configuración de las máquinas con Maven para integrar con Nexus

10. Documentar las configuraciones e instalaciones

11. Configuración de los servidores Testing, Pre-Producción e Integración [5]

Algunas de las configuraciones de los IDEs se hacen por parte del propio programador, pero siempre con el conocimiento por parte de QA para saber qué es lo que se está utilizando y en caso de problemas ser el responsable de calidad quien dedique tiempo a su solución, siendo por norma general bastante más rápido y ágil, no desconcentrando al equipo de desarrollo de su tarea principal de programar.

Una vez realizadas estas tareas en la primera iteración, en la reuniones de retrospectiva y revisión del sprint se proponen nuevas acciones a tomar y cambios en las configuraciones, para poder realizar el despliegue en la siguiente iteración. Durante el despliegue los programadores ejecutan sus construcciones mediante Maven y descargando los repositorios de artefactos mediante Nexus, añadiendo también los necesarios durante el siguiente desarrollo del proyecto de programación. Entonces, durante el despliegue, se realiza:

1. Cambios en configuraciones en Maven, Nexus y Hudson

2. Configuraciones de tareas Hudson para pasar tests de calidad

3. Primeras ejecuciones de tests mediante Hudson

4. Adaptación del proyecto de desarrollo del ERP a Maven, mediante configuración del archivo `pom.xml`, como se puede comprobar en el apéndice B

5. Análisis de datos de los tests según Integración Continua

[5] El entorno Integración no está en el diseño como tal porque es la integración dedicada por el servidor Hudson en conjunción con Sonar y la base de datos.

6. Configuración de los nodos de Hudson en cada entorno

Una vez analizados los primeros datos y propuestos durante las reuniones de la metodología se proponen nuevas modificaciones si es necesario, pero a partir de aquí ya se puede realizar un seguimiento de cómo se adapta la arquitectura completamente al desarrollo del software y sacar conclusiones.

5.2.2. Pruebas realizadas sobre la arquitectura

Una vez implantada la arquitectura en la infraestructura de desarrollo software de la empresa se deben realizar pruebas y hacer un detallado seguimiento de cómo se está desarrollando y utilizando para concluir si efectivamente las herramientas empleadas consiguen que la metodología ágil de desarrollo sea efectiva, y que se consigan los objetivos deseados con la implantación.

Primero se debe saber que el análisis de la arquitectura es un análisis cualitativo del desarrollo, donde lo que se pretende demostrar es que efectivamente la arquitectura ayuda a la agilidad del desarrollo del proyecto que está entre manos en la empresa. Para ello, las herramientas que se han propuesto y el diseño de su integración deben probar que el enfoque hacia la calidad del código se está cumpliendo.

¿Significa esto que con la arquitectura lo esperado es que no haya defectos en la construcción del código? Todo lo contrario. Es decir, lo que permite esta arquitectura de desarrollo no es que los defectos o *bugs* en el código vayan disminuyendo a cero, sino que su detección mejore y, conforme aumenten los errores o fallos, su resolución sea mucho más dinámica, rápida y ágil. Lo que sí es cierto, es que una vez finalizada la primera iteración global del producto los fallos y defectos deberían empezar notablemente a decrecer. Esto es algo que no se podría ver hasta que llegase la fecha de entrega del producto, hito que está fuera del alcance del estudio.

La metodología ágil en la gestión del proyecto de desarrollo ayuda a mejorar la agilidad en la acción contra los problemas y las nuevas características en el código, pero para poder ejecutarlas en un proyecto donde la integración es crítica necesita que las herramientas se adapten perfectamente a esta metodología. Esto es parte también de lo que se intenta demostrar.

También un indicador muy significativo es la satisfacción del cliente durante el desarrollo del proyecto, mostrando que efectivamente la percepción de calidad durante las entregas también se está logrando. El problema es que este indicador es difícil de medir, aunque las reuniones de la metodología donde se proponen las mejoras en este aspecto ayudan a medir la temperatura sobre la satisfacción, induciendo también una sensación positiva en el ánimo del equipo de desarrollo.

Aún siendo un estudio cualitativo para determinar el éxito de la implantación de la arquitectura, se necesitan ciertos indicadores cuantitativos para poder analizar los resultados. Se pueden disponer de distintos datos y realizar pruebas que darán perspectiva al éxito de la implantación:

- **Tendencia de los tests de código en Integración Continua**. En el ciclo de construcción del código Maven se ejecuta los tests diseñados en el desarrollo, tanto los tests unitarios, de integración y los tests gráficos de la aplicación. Se verá cómo los tests van aumentando notablemente tras la implantación de la arquitectura y que además la tendencia de resolución es buena.

- **Ejecución de tests en entornos de pruebas**. Los tests se ejecutarán en distintos entornos para ver la tendencia del desarrollo. En el entorno Testing los cambios son constantes y los tests se deberían ver más inestables en el tiempo, pero con una tendencia positiva en su resolución.

- **Número de *bugs* reportados y tiempo de resolución**. Conforme se posea más información, fluya de forma más dinámica, a la vez que el desarrollo es más ágil, los

bugs se deben encontrar más eficazmente y con mayor frecuencia, a la vez que su tiempo de resolución debería ser menor conforme avanza el proyecto.

- **Estadísticas de tareas de automatización Hudson**. El servidor de Integración Continua Hudson establece una política de muestreo de información que determina cuándo una tarea de automatización ha sido exitosa o no. Se establecerán según la tarea, se determinará cuándo se debe considerar como tal y cuál es el resultado esperado.

- **Tiempo de ejecución en los tests**. Según los entornos se podrán sacar resultados de rendimiento según el tiempo de ejecución en los tests.

Pruebas de Integración

Se han establecido los tests en el entorno de Integración de forma que la construcción del código, automatizada por Hudson y ejecutada mediante Maven, se tengan las distintas partes lógicas de la aplicación - según la figura 5.5 - en distintos servidores para así mostrar las debilidades de la aplicación cuando existe una integración tecnológica del desarrollo en una red distribuida. La integración se puede ver en la figura 5.7.

Según los resultados en el histórico de Hudson se puede comprobar cómo han ido evolucionando los tests en el desarrollo. Se observará si aumenta el número de tests en el tiempo y además se podrá comprobar si efectivamente ante la aparición de fallos se resuelven de forma dinámica.

Se realizan dos tipos de tests: los del entorno gráfico, donde se ejecutan tests unitarios del código y tests de usabilidad del interfaz; y los del *Back End*, donde se prueba la estabilidad de la parte servidor. Por tanto la parte servidor es crítica para poder usar eficazmente el interfaz de uso de la aplicación, por lo que los tests de la parte servidor serán más estrictos para considerarse exitosos.

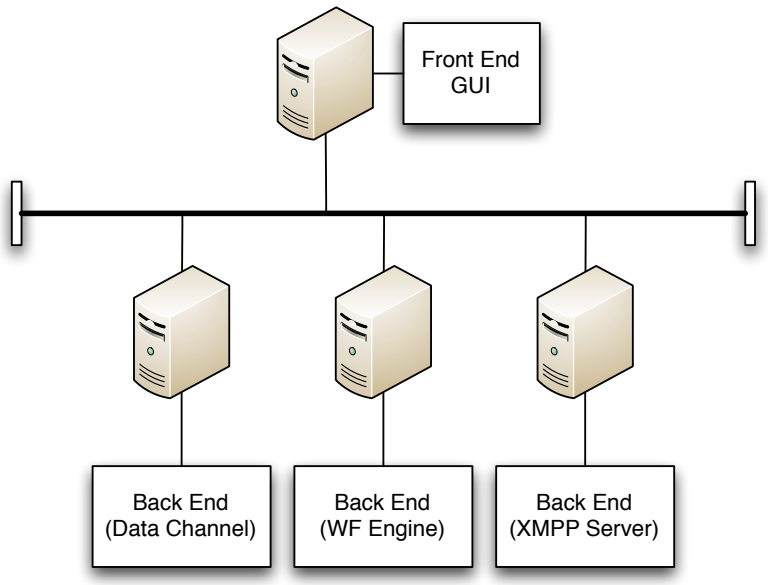

Figura 5.7: Servidores de pruebas en la integración.

La automatización de estos tests es diaria, pero el código que utiliza del repositorio lo hace según versiones y no con los cambios diarios. Así los errores de código unitario no deberán ser frecuentes pero sí los de la integración en sí misma de los servidores. Las tareas de tests de Hudson se ejecutan todas las noches, para no interferir con otro tráfico distinto en la red al generado por la aplicación.

Pruebas del entorno Testing

En el entorno de desarrollo ó Testing los tests deben ser más dinámicos y se esperan encontrar más errores y defectos, o por lo menos con más frecuencia, ya que es un entorno en continuo desarrollo y cambiante todos los días. Según la Integración Continua debería haber cambios todos los días y con frecuencia durante cada día, por lo que es muy importante encontrar los errores en el propio código además de cumplir con la funcionalidad requerida por la aplicación.

En este entorno se pasarán todos los tests, tanto de la parte servidor, como de la parte

cliente, y se deberán monitorizar continuamente. En este caso no es tan importante que los distintos servidores se encuentren distribuidos, por lo que todo se ejecuta en la misma máquina servidor, incluidas las pruebas gráficas del cliente de usuario del ERP.

En este entorno se ejecutan automáticamente los tests todas las noches, pero también manualmente durante el trabajo diario desde el responsable de QA se ejecutan las tareas de pruebas para poder ir monitorizando y haciendo un seguimiento de la calidad del código y la funcionalidad de la aplicación ERP. Es decir, este entorno de pruebas está disponible para ejecutar las tareas de pruebas propias de los programadores para controlar cómo discurre la integración de su código.

Estas pruebas son muy importantes ya que determinan la agilidad con que el equipo de desarrollo lleva a cabo el proyecto, y muestra de antemano la dirección que van a tomar las pruebas en el entorno de integración.

Pruebas del entorno Pre-Producción

Según el diseño de la arquitectura, se tiene un entorno que se ha denominado *Pre-Producción*. No es realmente un entorno como tal, ya que la aplicación no entrará en producción hasta que haya concluido el desarrollo del ERP en su primera versión, pero cada vez que se entrega al cliente una parte del desarrollo que él debe probar para comprobar que se están cumpliendo sus requisitos y que las funcionalidades requeridas están correctamente implementadas en la aplicación.

Este entorno debe ser el más estable de los tres, y por tanto sus tareas de automatización sólo consisten en generar la parte cliente y la parte servidor de forma correcta. Este entorno es en realidad un clon del entorno de desarrollo y con el código altamente probado en Integración y Testing. Las pruebas gráficas de usuario no son ejecutadas en este entorno, ya que su fin es proveer al cliente del *Back End* para poder ejecutar el clien-

te en sus máquinas la aplicación, siendo el usuario por tanto, quien realiza las pruebas manualmente.

Las tareas que el servidor de Integración Continua ejecutan en este entorno son:

1. Construcción del *Back End* según la última versión estable.
2. Construcción del *Front End*, según la última versión estable.
3. Despliegue del *Back End* para pruebas del cliente.
4. Empaquetado de la aplicación (*Front End*) para la instalación del usuario.

Con esto se le entrega al cliente el instalador de pruebas del ERP para que el usuario realice las pruebas manuales y se obtenga realimentación de la información (*feedback*) para la mejora continua del desarrollo del producto.

5.3. Resultados

Con las pruebas descritas en la sección anterior se realiza un análisis descriptivo y cualitativo del funcionamiento de la arquitectura del desarrollo y del propio desarrollo del proyecto del ERP.

En realidad se necesitan los datos de las pruebas del propio código para poder estudiar cualitativamente el funcionamiento de las herramientas de la implantación de arquitectura. Es decir, según se vean los datos y los cambios durante el tiempo en el desarrollo del producto, se puede analizar exhaustivamente si las herramientas y el diseño de su integración en el equipo de desarrollo están cumpliendo con las expectativas de una arquitectura como esta.

Tabla 5.4: Diferencias tras la implantación.

Antes de implantar	Después de implantar
6 meses en primera funcionalidad	Entregas cada 4 iteraciones (1 mes)
Sin datos de tests	Tests estructurados y monitorizados
Sólo tests unitarios	Tests unitarios, de integración y funcionales
Resolución fallos ad-hoc	Anticipación de fallos
Redundancia innecesaria en tests	Redundancia sólo cuando necesaria
Actitud reactiva del equipo	Actitud proactiva del equipo
Escasa visibilidad de la ejecución del código	Construcciones del código visibles por todos

Se verán los resultados obtenidos gracias al servidor de Integración Continua Hudson, y según los entornos se comprueba si efectivamente los resultados obtenidos ayudan a sacar conclusiones válidas para verificar que implantar una arquitectura de desarrollo como la propuesta.

5.3.1. Datos iniciales en la implantación

Durante la primera iteración en la implantación no había realmente ningún método para medir el estado de la calidad del desarrollo y había cierta incertidumbre en el verdadero estado en la programación por parte del equipo de desarrollo. En realidad no existían datos de tests excepto aquellos que cada programador había ejecutado unitariamente en su código. Esto provocó el retraso en la entrega del primer módulo funcional de la aplicación.

Las diferencias percibidas tras implantar la arquitectura de desarrollo se han descrito en la tabla 5.4, mostrando un claro ejemplo de que tras la instalación, configuración e integración de las herramientas del diseño, los objetivos de la metodología ágil se acercan más a la realidad. Lo que muestra la tabla es que efectivamente el dinamismo del equipo de desarrollo se agiliza y la actitud hacia la creación y ejecución de tests se convierte en un hábito sin hacer necesaria su definición en cada tarea del proyecto.

Lo que ocurría antes de la implantación es que el equipo de desarrollo no tenía una visibilidad de si la arquitectura de la aplicación que se había diseñado realmente funcionaba según la integración cliente-servidor. Antes de tener una plataforma de Integración Continua hacía más arduo la obtención de conclusiones del trabajo y pruebas realizadas por cada programador. Había que seguir una organización exhaustiva por parte de la gestión del proyecto para poder analizar los errores e incoherencias que se presentaban.

Tras la primera iteración de la implantación hubo un cambio natural en el equipo, donde la resolución de tests y programación de los nuevos era mucho más fluida, tendiéndose más hacia una mecánica de trabajo TDD (*Test Driven Development*). Ésta empezaba a ser una buena aproximación hacia la mejora de la calidad del código, buscada también por la gestión mediante metodología ágil del proyecto.

Otra interpretación de estos datos está en que cada programador del equipo está más centrado en su trabajo, pero con una visibilidad global del resto del código del equipo, con el que se tiene que integrar. Esto también es positivo.

También otro dato importante está en el hecho que antes de tener lista la arquitectura mediante el primer despliegue no se tenía constancia de cómo estaba funcionando la integración de la arquitectura interna del ERP, ya que se monitorizaban independientemente los servidores y registros de cada parte. Pero no había forma centralizada de saber cómo estaban interactuando entre ellos. Los *logs* de las tareas del servidor de Integración Con-

tinua con el código actualizado mediante Mercurial ayudan a esta propia tarea de QA, donde gracias al seguimiento del Scrum se comunicaba inmediatamente al equipo para que cada parte tomara nota de los fallos detectados y actuase rápidamente en consecuencia.

5.3.2. Datos de Mejora Continua

El entorno Testing o Desarrollo está construyendo código en continuo cambio. Estos datos son muy significativos y todos los datos que se obtienen son indicadores inmediatos de cómo se está desarrollando el código. Las tareas que el servidor de Integración tiene configuradas para esto son:

1. Construcción del modelo de datos
2. Construcción del Data Channel y ejecución de tests
3. Indexado de datos según la capa de Data Channel
4. Construcción del motor de *workflows* (flujos de proceso)
5. Construcción del cliente interfaz y ejecución de tests, de integración y funcionales (de usuario automatizados)

De las estadísticas mostradas por Hudson en las ejecuciones de pruebas se puede comprobar en la figura 5.8 que un 18 por ciento de todos los tests en este entorno están fallando, mientras que el éxito de los tests está en el 82 por ciento. En el estado final del seguimiento se estaban ejecutando 135 tests de integración, de los cuales 102 son tests que comprueban la funcionalidad de la aplicación mediante pruebas gráficas de usuario en el interfaz.

Lo que se observa es que en las primeras ejecuciones casi la mitad de los tests están dando errores, siendo significativo la cantidad de aspectos a mejorar en la integración. Dos iteraciones de desarrollo después se consigue bajar la tasa de errores notablemente,

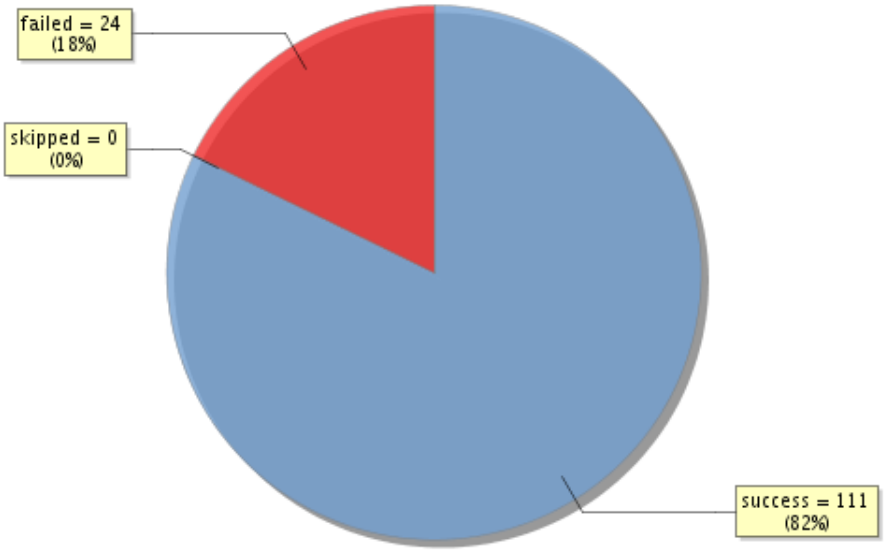

Figura 5.8: Estadísticas de tests entorno Testing.

y posteriormente se consigue obtener una tasa de errores en las funcionalidades parecida para un mayor número de tests funcionales. Es decir, aumentando en 25 las nuevas funcionalidades, la tasa de errores no sube, por lo que se consigue mantener un nivel alto de calidad en el que se ve una tendencia de mejora a lo largo del tiempo.

Desglosando en este entorno los datos de los tests se observa la representación de la tabla 5.5. Lo que se comprueba es que efectivamente la tasa de éxito en el Back End, donde es crítica, los tests de integración son exitosos al ciento por ciento, y gracias a la arquitectura se puede seguir en todo momento si todos los cambios que se están realizando a la infraestructura del ERP se realizan de forma adecuada o no.

Sin embargo, en la parte cliente, donde se realizan pruebas continuas de las funcionalidades de usuario en el interfaz, es normal que siempre exista algún error a la hora de

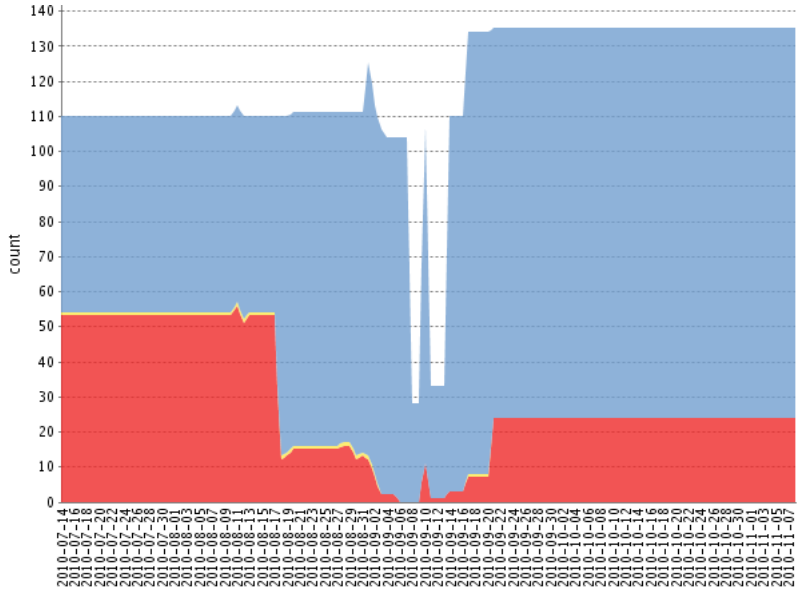

Figura 5.9: Tendencia de tests entorno Testing.

construir, ya que depende de múltiples factores que tienen que ir solucionándose según se va desarrollando. Es decir, es importante saber que la forma de desarrollo, al estar enfocada a los tests, se realiza tal que cada funcionalidad se diseña en función a un tests a pasar. Se continúa iterando su desarrollo tras cada test, por lo que por cada funcionalidad nueva que se va introduciendo en la primera iteración hay un porcentaje de error al ejecutar las pruebas. Este porcentaje, desde el departamento QA se establece en el **80 por ciento de éxito** siempre y cuando la tendencia de fallo sea decreciente. Así tras las iteraciones en las funcionalidades sólo se da por terminada cuando el éxito de dicha funcionalidad sea del cien por cien.

5.3.3. Datos de Integración

En el entorno Integración, donde los servidores están distribuidos es necesario saber si existen errores de rendimiento, de integración cuando los servidores están distribuidos, de cortes de conexión, etc. En definitiva, cuando los distintos elementos de la infraestructura del ERP están distribuidos es necesario ver cómo se comporta la aplicación y los errores

Tabla 5.5: Desglose de tests en Testing

Capa de aplicación	Tiempo ejecución	Tiempo de pruebas	Porcentaje éxito
Data Channel	2m 56s	0.84s	100 %
Front End	1h 14m	1h 9m	82 %
WF Engine	1m 15s	0s	—

que van surgiendo. lo que permitirá la arquitectura de desarrollo será poder reaccionar ágilmente ante los errores en este entorno, que son mucho más significativos en cuanto al aumento de factores existentes en los tests, ya que las conexiones entre servidores y peticiones de datos a través de la red mediante el software eleva exponencialmente la complejidad.

En este caso el histórico de datos es mucho mayor porque se deben guardar los datos en mayores intervalos, ya que es vital analizar la tendencia de los tests para compararlos con la implantación de la arquitectura.

En la figura 5.10 se observa que los errores en los tests son mucho mayores. Es lo esperado por parte del equipo de desarrollo, por lo comentado en cuanto a las variables de las que este entorno depende. La tasa de fallos aquí sube hasta el 33 %. La verdad es que es una tasa bastante alta teniendo en cuenta que la calidad es uno de los factores que más se tiene en cuenta en el desarrollo de este producto. Pero el equipo de desarrollo relaciona esta tasa a una menor dedicación a los servidores distribuidos, debido al cambio de requisitos en la infraestructura de la aplicación.

Es decir, a partir de cierto momento la ejecución de tests en integración se realiza para monitorizar los errores que tengan una implicación directa en el rendimiento del ERP al distribuir los servidores, pero los errores de conexión con los servidores no son tenidos en

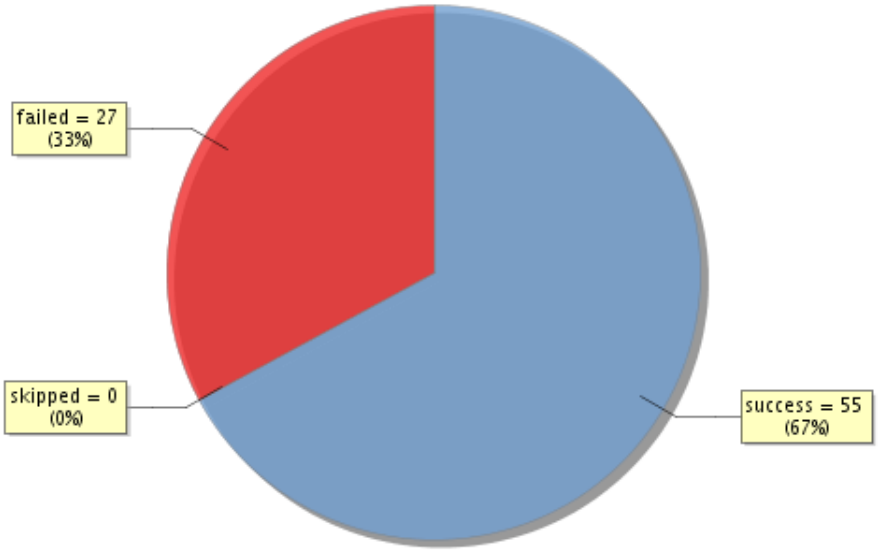

Figura 5.10: Estadísticas de tests entorno Integración.

cuenta. Aún así, los tests en este aspecto se siguen ejecutando para tener localizado en todo momento los problemas con las configuraciones de los servidores distribuidos en la red. Es la razón por la que en el gráfico de la figura 5.11 las últimas ejecuciones mantienen estables determinados tests.

Pero lo interesante de esta gráfica respecto a la arquitectura son las variabilidades del éxito de tests. Es decir, si se atiende al primer tercio de la gráfica se comprueba que no hay datos de tests ejecutados con éxito o no. Esto ocurre porque en el momento de instalar el servidor de Integración Continua en el entorno Integración, no existía aún pruebas funcionales del producto. Pero lo interesante es ver que nada más ejecutar las pruebas diseñadas se obtiene un porcentaje de éxito enorme.

Si se analiza la misma gráfica se observa cómo crece el número de tests en el tiempo

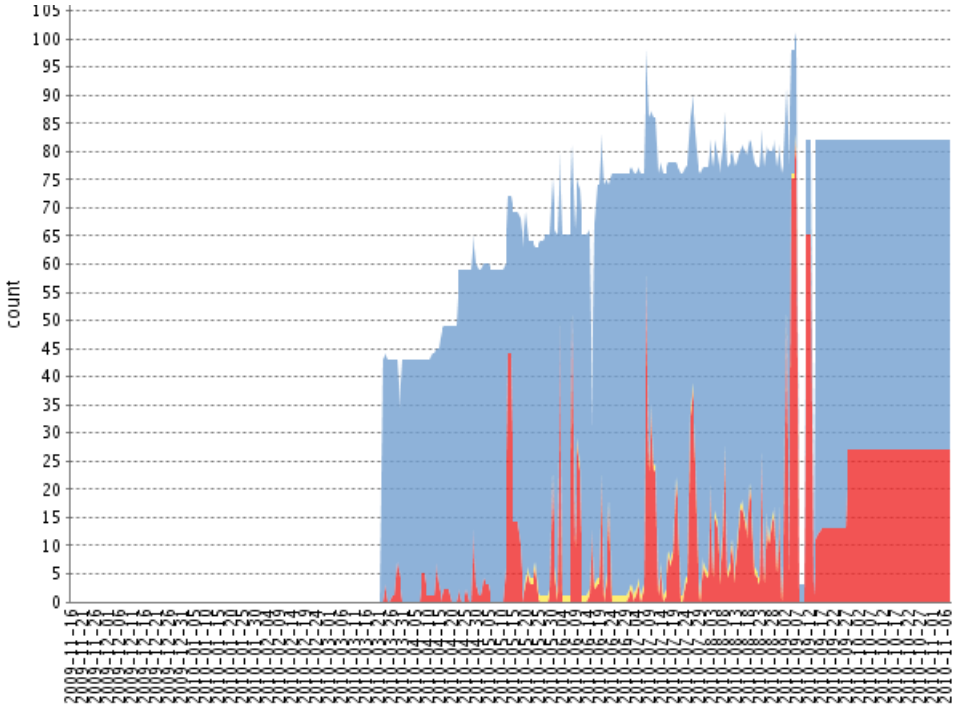

Figura 5.11: Estadísticas de tests entorno Integración.

y los picos de tests fallidos (en rojo) no se mantienen durante esa línea temporal, sino que en dos tareas de Hudson consecutivas ejecutadas se solucionan. Esto demuestra una rápida acción por parte del desarrollo en cuanto a la solución de los errores en el código, y por tanto la agilidad en la acción del equipo de desarrollo ante cambios en el código y su integración es bastante inmediata.

Las tareas de ejecución en Hudson que muestran los datos en este entorno son las mismas que en el entorno Testing, pero configuradas para arrancar remotamente en los entornos distribuidos. Esto provoca que los tiempos de ejecución aumenten. Según los datos de la tabla 5.6, se comprueba la diferencia de tiempos.

Lo que muestra la tabla es que en la parte del Back End los tiempos de ejecución aumentan, mientras que el tiempo de ejecución de tests es menor. La razón por la que los tests emplean menos tiempo es simplemente porque el número de tests en este entorno es menor que en Testing. El total de tests en este entorno es de 79, mientras que en Testing

Tabla 5.6: Desglose de tests en Integración

Capa de aplicación	Tiempo ejecución	Tiempo de pruebas	Porcentaje éxito
Data Channel	5m 50s	28ms	100 %
Front End	52min	42m	63 %
WF Engine	1m 15s	0s	—

son 135. ¿Por qué es así? Pues porque en el entorno Integración la versión que se integra está por detrás de la versión que está en desarrollo, y por tanto el número de tests será menor al tener menos funcionalidad.

Pero hay algo que en los resultados no encaja y es que si Integración siempre está usando el mismo código de una versión, mientras que Testing está continuamente cambiando, ¿por qué entonces la fluctuación en los errores es mayor en Integración? Este es un dato interesante a estudiar, aunque la respuesta está en que este último entorno hay errores que se deben a temas de conexión o reinicio de servidores distintos. En esos momentos, si la conexión falla algunos tests también lo hacen, y en el momento que la conexión se restablece en la siguiente ejecución el mismo test ya no falla.

Lo único que indica esto es que los errores que este entorno saca a la luz son más de la integración de los servidores que del propio código en sí, siendo bastante más compleja la interpretación de los datos. En este caso la continua monitorización por parte de QA es esencial para establecer nuevas tareas.

Volviendo los datos de la tabla 5.6, y observando la ejecución del cliente del ERP (Front End), podría sorprender que el tiempo total es menor que en desarrollo, tanto en la ejecución de la tarea como en los tests. Pero si se interpretan los datos, teniendo en cuenta que se están ejecutando 56 pruebas menos, el caso cambia. En el caso del entorno

Integración la diferencia entre la ejecución de tests y la ejecución total de la tarea (tests incluidos) son 10 minutos. Sin embargo, en el entorno Testing según la tabla 5.5 esta diferencia es de tan sólo 5 minutos. Comprobando efectivamente que la ejecución de la aplicación cuando los servidores están distribuidos es mayor, tal y como era de esperar.

5.3.4. Otros resultados

Además de todos estos resultados ligados a las tareas automatizadas del servidor de Integración Continua Hudson hay otros datos cualitativos que tal vez necesiten mención a la relación con la implantación de la arquitectura.

- **Comunicación con el cliente final**. Al inicio de la implantación el estado en la captura de nuevas funcionalidades y requisitos se realizaba estrictamente mediante la metodología ágil, Scrum en este caso. El resultado siempre era la captura de requisitos muy dispersos en numerosas reuniones y la desorientación del cliente al no tener un conocimiento real de la agilidad del desarrollo. Tras la implantación de la arquitectura y el desarrollo con ella durante un tiempo, las numerosas reuniones se han eliminado y sólo con el seguimiento de la metodología la comunicación es mucho más clara y concisa.

- **Satisfacción del cliente**. Desde la implantación de la arquitectura y las entregas más frecuentes el número de informes con errores en la aplicación por parte de las pruebas del usuario han disminuido y la petición de nuevas funcionalidades ha aumentado.

- **Resolución de bugs**. Las tareas a realizar del desarrollo se introducían en el sistema de seguimiento, al igual que los *bugs* detectados. Al inicio los bugs resueltos eran muy pocos, debidos a la falta de visibilidad antes de la implantación y a los escasos datos respecto a los tests resueltos. Ni siquiera se introducían *bugs*, al ser resueltos por cada programador antes incluso de ser visibles, pero empleando mucho tiempo

y resultando poco ágil. Al implantar la arquitectura de desarrollo, en la versión 0.1 (iteración 11 del desarrollo), la tendencia cambia y como se comprueba en la figura 5.12 el número de *bugs* reportados y resueltos aumenta considerablemente durante el desarrollo del proyecto con la arquitectura ya implantada y adaptada al equipo de desarrollo.

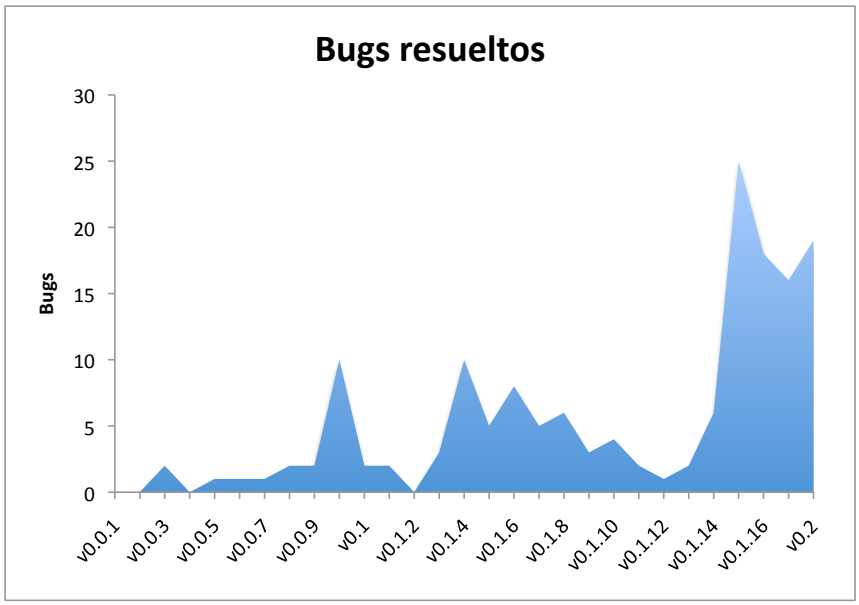

Figura 5.12: Número de resolución de bugs por iteraciones.

El tiempo de resolución de *bugs* no es significativo desde el punto de vista donde la medida temporal es la iteración, y lo que muestran los datos en cuanto a tiempo de resolución de errores es el número de ellos que se resuelven en cada iteración.

5.3.5. Análisis global

Lo que se puede determinar con los datos del caso de estudio durante el seguimiento es que lo que consigue la implantación de la arquitectura de desarrollo aplicada al caso del ERP es un mecanismo para detectar de forma más óptima los errores en el desarrollo y programación, y poder actuar rápidamente según las necesidades de la programación y

del equipo.

Conforme avanza el proyecto de desarrollo y aumentan las funcionalidades y se crean nuevos módulos de la aplicación las dependencias en la integración aumentan, lo que en principio se traduce en un aumento en los fallos de los tests, pero lo interesante es que la relación es exponencial. Es decir, las dependencias pueden aumentar enormemente, lo que hará que los fallos también aumenten, pero de forma que los fallos aumenten en menor medida que las dependencias. Además llega un punto en el máximo de fallos donde el aumento de dependencias no produce aumento de fallos. Este punto debería coincidir con la finalización de nuevas funcionalidades según los requisitos iniciales. Dicho punto no se alcanzó en el caso de estudio, ya que aún faltaban algunos meses para finalizar la versión 1.0 del ERP.

Para entender esta relación de dependencias en la integración y desarrollo del código sería descriptivamente lo mostrado en la figura 5.13

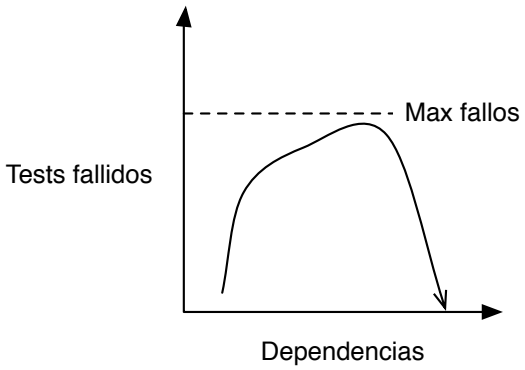

Figura 5.13: Fallos en función a dependencias de integración.

En definitiva, la implantación de una arquitectura como la propuesta parece ser una buena aproximación para el desarrollo de un producto bajo una metodología ágil como el caso estudiado, donde se consigue efectivamente una visibilidad ante los fallos mucho mayor, consiguiendo también una capacidad de reacción a la programación rápida, enfocándose de forma óptima hacia la calidad del desarrollo del código y la gestión del proyecto.

Algunos problemas con el caso de estudio que tal vez puedan llevar a tomar otras interpretaciones puede ser:

- Proyecto estudiado de demasiada envergadura para un análisis inicial
- Indicadores de satisfacción del usuario sensibles a la subjetividad
- Registros de tests no analizados en toda profundidad de detalle
- No haber participado desde la iteración cero del proyecto de desarrollo

Efectivamente estas razones pueden llevar a discutir más profundamente los resultados aquí expuestos, pero un estudio así estaría más allá de los objetivos de este proyecto fin de carrera, y por tanto para otro tipo de estudio. Aún así, efectivamente un caso de estudio más sencillo podría haber llevado a los mismos resultados de este capítulo, pero este caso ha aportado datos muy interesantes que hacen ver lo útil que puede ser la implantación de una arquitectura como la propuesta.

Capítulo 6

CONCLUSIONES

A continuación se exponen las distintas conclusiones que se pueden sacar tras el estudio de las herramientas y el diseño propuesto de la arquitectura de desarrollo para proyectos ágiles de software. Según los objetivos propuestos al inicio del proyecto y estudiados durante su desarrollo se pueden analizar distintas conclusiones, tanto del cumplimiento de los objetivos propuestos como de las distintas acciones que se han ido realizando durante la ejecución.

El caso de estudio analizado en el capítulo 5 ha ayudado a conocer las implicaciones que tiene el diseño propuesto y las herramientas seleccionadas. Además es una suerte poder trabajar profundamente en un caso real de un desarrollo específico y trabajar desde dentro del Departamento de Calidad, conjuntamente con la gestión ágil del proyecto. Esto ha proporcionado datos reales de un proyecto de "nacimiento" de un producto software y que, aún a fecha de finalización de este proyecto fin de carrera, está en fase de lanzamiento.

6.1. Conclusiones sobre los objetivos

Las arquitecturas de desarrollo en las empresas o en los departamentos de software se implantan normalmente según unas necesidades iniciales de la propia empresa en general, y que durante el desarrollo de los distintos proyectos se van adaptando según el trabajo

de programación de los equipos de desarrollo. El objetivo principal en este proyecto ha sido poder dar una visión distinta a este aspecto y plantear la implantación de una arquitectura de desarrollo adaptada a una metodología específica, utilizada para la mayoría de proyectos software por uno o varios equipos de trabajo.

Las metodologías ágiles al fin y al cabo se basan en mecánicas de trabajo, de gestión y de disciplina de los recursos de un equipo de desarrollo para poder realizar un trabajo de forma más dinámica, y adaptarse correctamente a los continuos cambios del mundo del software durante la ejecución de proyectos. Pero esta visión es común tenerla enfocada a las herramientas de uso de las propias personas que trabajan en aquellos proyectos. Por norma general la implantación de las herramientas que usan los programadores y personas implicadas en el desarrollo no están basadas en las características de metodología del equipo, sino más bien en especificaciones técnicas y de uso tecnológico, no humano.

Es decir, está claro que las metodologías ágiles tienen un enfoque más humano en la consecución de proyectos, pero esto no encaja con el uso que hacen de las herramientas para ello. Así, un planteamiento que seleccione e integre las herramientas de desarrollo adecuadas para poder ejecutar estos proyectos desde ese punto de vista más "humano", es lo que se pretendía mostrar aquí, y que teniendo en cuenta ciertos aspectos y conocimiento actual de las herramientas era posible.

Efectivamente en este caso de estudio se ha podido concluir que puede funcionar este tipo de planteamientos. Un equipo de desarrollo que no está acostumbrado en su vida diaria a llevar una "mecánica ágil" necesita de algo más que unos principios y unas metodologías de trabajo. Necesitan que las herramientas que usan todos los días les permitan centrarse en esas mecánicas de trabajo y que cuando tengan que adaptarse a una integración continua diaria no tengan que "pelearse con el destornillador para apretar el tornillo". Encontrar unas herramientas que también sean ágiles en su manejo y fáciles de integrar ayudan a esto. Volviendo al símil del destornillador, si éste es ergonómico y diseñado para

no hacer esfuerzo, no habrá ningún problema apretando tornillos.

Entonces según los objetivos se ha podido comprobar que:

- Se han seleccionado herramientas que consiguen abstraer al programador de la complejidad en las dependencias del software y que se integran perfectamente con la arquitectura global del desarrollo. Además, estas herramientas consiguen mantener una información en tiempo real que ayuda a saber en todo momento si el camino tomado es el adecuado.

- Se ha logrado diseñar y adaptar las distintas herramientas y trabajo del equipo de forma que la calidad del desarrollo es el centro del trabajo. La ejecución de tests que permite la arquitectura, la forma en que monitoriza los datos y la continua disponibilidad de resultados a todos los implicados en el desarrollo facilita enormemente la agilidad del proyecto y mantiene el foco del objetivo.

- El trabajo del equipo de desarrollo no ha sido un problema a la hora de implantar la arquitectura. Al igual que aprender a manejar ciertas herramientas requiere invertir cierto tiempo, la mayor complejidad en este aprendizaje consigue mantenerse transparente al programador, siendo más responsabilidad el conocimiento de la complejidad del departamento de QA.

- La arquitectura implantada consigue un hito en el desarrollo del proyecto, y es el enfoque hacia la calidad. Desde el momento que se implanta la arquitectura los fallos y *bugs* del software se hacen más visibles, y su resolución más dinámica, por lo que el producto final es más estable y de mayor calidad.

Pero es importante hacer hincapié en que lo que muestra el proyecto no es un diseño de arquitectura universal para cualquier proyecto de desarrollo mediante metodología ágil. Lo que muestra es cómo afrontar el diseño de una arquitectura de este tipo, los factores a tener en cuenta, lo meticuloso del trabajo para poder sacar rendimiento de un equipo de desarrollo software y la cada vez mayor aproximación del uso de herramientas adecuadas

para poder implementar metodologías adaptadas a la demanda obtenidas del conocimiento maduro de proyectos más clásicos de la Industria.

Tal vez un mayor estudio de todas las herramientas y las nuevas metodologías ágiles concluyan con otros enfoques aproximados o más completos. Además, la existencia de herramientas más cerradas y con licencias comerciales específicas para este tipo de usos podría dar nuevos datos a un estudio como este, pero conlleva estudiar herramientas que están muy dirigidas al uso de un determinado sector o empresa y sería mucho más costoso, implicando además la participación de ciertas empresas y la consiguiente dificultad.

6.2. Conclusiones generales

El sector de las tecnologías de la información está empezando a ser un sector mucho más maduro de lo que ha sido durante los últimos años. La problemática de la gestión de proyectos en este sector siempre ha estado en que las normas y metodologías de la gestión típica de proyectos no funciona con el desarrollo del software y es cuando nacieron las metodologías ágiles.

Conforme el desarrollo de software ha ido avanzado y el sector ha madurado, las disciplinas aplicadas en la gestión de proyectos de industria para optimizar tiempo y mejorar la calidad se han adaptado al mundo del software. Ahora que se conoce mucho más de las distintas problemáticas de la gestión de proyectos de software y existen muchas más herramientas y experiencia en el negocio, las metodologías más clásicas enfocadas a un entorno ágil están ganando terreno. Esto lo se ha vivido en este caso de estudio, donde a la metodología Scrum del inicio se ha añadido una dinámica de trabajo tipo Kanban, enfocada a la mejora continua y conocida precisamente del mundo del desarrollo industrial.

Esto afecta a la implantación y utilización de herramientas de desarrollo. Al igual que las filosofías de la gestión de proyectos en los sectores de producción material afectan tam-

bién a la forma en la que se usan sus herramientas, cómo se organizan, cómo se adaptan, cómo son utilizadas, e incluso qué enfoque tienen a la hora de crearse ellas mismas según distintos factores, en el software está ocurriendo prácticamente lo mismo. Muchas de las herramientas software utilizadas para desarrollos actuales han sido creadas hace tiempo y no han tenido en cuenta aspectos que ahora son distintos, debido tanto a la evolución del sector como a su propia madurez.

Gran parte de las herramientas de desarrollo han sido durante mucho tiempo de propósito general y se han utilizado en numerosos entornos de desarrollo, siendo bastante polivalentes y funcionales. Pero actualmente la utilización de herramientas específicas está ganando algo de terreno, ya que está demostrado que una herramienta bien diseñada según unos objetivos claros, específicos y bien acotados optimiza enormemente su uso.

Pero no se puede decir que una aproximación o la otra es la correcta para la "verdad absoluta", sino que los factores y características de un equipo de desarrollo tienen mucho que decir en una u otra. Así, existe otra aproximación, y es la utilización de herramientas más básicas de propósito general que cumplen perfectamente con los requisitos globales y acotados a todas las arquitecturas, pero que para uso en entornos específicos son extensibles mediante *plugins* o extensiones. El diseño e implantación propuestos demuestra que esto es posible, ya que la mayoría de herramientas empleadas son de esta clase.

De todas formas, para poder asegurar esto sin ninguna duda, se debería estudiar otro caso completamente distinto, en el que con las mismas herramientas pero con diferente configuración e integración, cumplan perfectamente el objetivo.

También hay que resaltar, que hace un tiempo, la visión de la programación como un arte ha provocado que el desarrollo de cierto software fuesen obras maestras de la tecnología informática, y que los enfoques de producción de software como ingeniería a veces han llevado a realizar aplicaciones muy funcionales pero tecnológicamente más precarias

y poco flexibles. Pero el giro que últimamente está dando el sector, donde su madurez permite un enfoque más ingenieril y menos artístico para la obtención de software funcional, "usable", tecnológicamente avanzado y flexible, consigue encontrar el punto medio entre arte e ingeniería.

Es en este estudio donde se ha encontrado este problema, y que la arquitectura de desarrollo ayuda a solucionarlo. Un programador acostumbrado a una herramienta obsoleta, considerándose él mismo un artista, y rechazando ciertas otras herramientas porque están enfocadas a un objetivo más empresarial, es un obstáculo para el desarrollo ágil. Lo que se ha propuesto entonces es simplemente un esfuerzo por parte del programador a dar una oportunidad a esa herramienta, ya que facilita enormemente el desarrollo global. Y existe una probabilidad muy alta de que termine resultando satisfactorio su uso por su parte.

Desde una perspectiva diferente, ante la inflexibilidad de ciertos programadores a la adaptación hacia este tipo de herramientas, hay que cambiar su punto de vista y a veces obligar inicialmente a que empleen parte de ellas. Se debe dirigir el desarrollo de un equipo hacia ese punto medio entre el arte y la ingeniería, agilizando así el proceso.

En definitiva, la conclusión general que se pretende exponer está en que hoy día el desarrollo de proyectos software tiende hacia una optimización total del tiempo a la vez que un aumento de calidad en los procesos ,que se refleja en el resultado final del producto. De la misma manera se ha desarrollado durante mucho tiempo en los últimos años en la gestión del desarrollo industrial y tecnológico. Igual que en este último, la importancia de las plataformas y herramientas de desarrollo y cómo se adaptan a la gestión dinámica es un proceso clave. En el desarrollo del software está ocurriendo lo mismo, y una arquitectura como la aquí planteada es un buen primer paso en el camino.

Capítulo 7

FUTURAS LÍNEAS DE TRABAJO

Este proyecto ha tratado un tema que es de actualidad en la gestión de proyectos software, que son las metodologías ágiles, y busca una forma de ayudar a gestionar este tipo de proyectos mediante el uso correcto de herramientas existentes y adecuada integración y configuración de las mismas.

No cabe duda de que en este tipo de estudios hay mucho camino que recorrer, tanto para tomar la dirección correcta en los análisis como para dar un nuevo punto de vista en la gestión del software. Siempre desde el buen uso y organización de los recursos existentes en el desarrollo. Por tanto, las líneas de trabajo que pueden abrir nuevos caminos para el exitoso estudio en la implantación de arquitecturas de desarrollo, enfocadas a dinamizar la ejecución del desarrollo software, pueden realizarse desde distintos puntos de vista.

Se pueden proponer entonces varias líneas que podrían extender nuestro proyecto:

- Integrar las herramientas de colaboración y comunicación de un equipo de tecnologías de la información con la plataforma de desarrollo implantada, para optimizar así la realización de un proyecto de producción o implantación software. La idea ya no está sólo en proveer de una arquitectura de desarrollo del software que sea ágil, polivalente, usable y estable, sino que además, se podría mejorar a través de correctas herramientas y metodologías la eficiencia del flujo de información.

- Otra línea de trabajo puede estar en utilizar el conocimiento de este estudio para implantar una arquitectura para entornos seguros, ya no sólo participando en un desarrollo ágil de proyecto, sino también empleando herramientas, protocolos y estándares de seguridad potentes con el fin de obtener también, en caso necesario, productos finales donde la seguridad sea una prioridad requerida. Es decir, partiendo de la base del desarrollo de procesos, código y procedimientos seguros, obtener también una aplicación software extremadamente segura para el uso de información sensible.

- También el trabajo realizado puede ser base de reutilización del conocimiento para nuevas metodologías de desarrollo, donde ya no sólo la persona que programa sea el centro de la gestión, sino que las herramientas que emplea para ello sean parte vital de la ecuación.

- A partir de estas plataformas, entendidas como base de un desarrollo de gran envergadura, se pueden crear herramientas nuevas donde el foco esté puesto precisamente en su integración con herramientas centradas en la integración continua del desarrollo software.

- Analizar nuevamente nuestro estudio para entornos mucho más empresariales o corporativos[1], donde las metodologías ágiles son mucho más complicadas de aplicar, e incluso en algunos casos no teniendo sentido hacerlo, pero buscando el mismo objetivo: respuestas dinámicas en el desarrollo del proyecto a los continuos cambios, optimizando en todo momento el proceso.

En definitiva, muchas de las líneas futuras de trabajo pueden entenderse encaminadas hacia lo ya mencionado sobre el acercamiento de la gestión del desarrollo de proyectos

[1]Mejor entendidos por su palabra inglesa: *Enterprise.*

software a disciplinas más maduras y más ingenieriles, teniendo en cuenta que esto puede ser así para productos donde los procesos de ingeniería son los válidos según unos requerimientos rígidos y empresariales. Es decir, para aplicaciones donde los procedimientos y la eficiencia en su funcionamiento son la misión crítica.

APÉNDICES

APÉNDICE A

PRESUPUESTOS

En este apéndice se presentan todos los presupuestos empleados para realizar el proyecto. Los costes se desglosan según las tablas A.1 y A.2.

En la tabla A.1 se especifican las horas que se han empleado en las distintas fases del proyecto empleadas, tanto la fases de ejecución y no ejecución, como las de seguimiento y redacción de la memoria. Sumando las distintas fases se obtienen las horas invertidas en el proyecto, llegando a un total de 1.290 horas. Si se tiene en cuenta que el precio del ingeniero de calidad de implantación de la empresa está alrededor de 40€/hora, el coste de personal de este proyecto asciende a 51.600€.

Tabla A.1: *Fases del Proyecto*

Estudio y diseño	320 *horas*
Planificación	80 *horas*
Implementación	80 *horas*
Despliegue	60 *horas*
Seguimiento y análisis	450 *horas*
Redacción de la memoria	300 *horas*

Además del coste de personal de este proyecto, realizado por una sola persona, se tiene un coste de material que es necesario tanto para la implantación de la arquitectura propuesta como para todo el estudio inherente al proceso de seguimiento y documentación de todo el proyecto. Aunque es un coste que la empresa del caso de estudio invirtió para su arquitectura de desarrollo, no se puede obviar que es un coste necesario para tener implantada una arquitectura en el equipo de desarrollo para llevar a cabo los proyectos.

Los distintos elementos materiales, tanto tangibles en hardware y recursos materiales, como intangibles en licencias software se enumeran a continuación, los cuales se estiman en la tabla A.2.

Material y licencias software

- 15 herramientas de construcción de código **Apache Maven v2.2.1**. Licencia Apache gratuita, descarga online, sin licenciamiento de soporte.
- 1 gestión de repositorios **Sonatype Nexus v1.8.0 Open Source**. Licencia gratuita, descarga online, sin licenciamiento de soporte.
- 1 servidor de repositorio de código **Mercurial v1.7** y 15 instalaciones cliente. Licencia libre GPLv2, descarga online, sin soporte.
- 3 servidores de Integración Continua **Hudson** (actualización de versión gratuita cada dos semanas). Licencia gratuita MIT, sin soporte contratado.
- 4 servidores de aplicaciones **Glassfish v3**. Licencia gratuita CDDL v1.0, sin soporte contratado.
- 1 sistema operativos **OpenSolaris SunOS 5.11**. Licencia gratuita CDDL v1.0, sin soporte comercial contratado.
- 1 sistemas operativo **VMWare ESXi.** Descarga gratuita, licenciamiento comercial de soporte. 229€.

- 7 sistemas operativos **Mac OS X Snow Leopard**. Licencia incluida con hardware, contratada actualización de 10.5 (Leopard incluído) a 10.6 (Snow Leopard). 29€/unidad y 2 años de vida estimada.

- 1 sistema operativo **Windows Vista Professional**. Licencia comercial con soporte, incluído en el material de la empresa. 249€y 3 años de vida estimada.

- 1 sistema operativo **Oracle Linux**. Descarga gratuita y licencia de soporte contratada. 100€y un año de renovación de licencias.

- 9 sistemas operativos **Debian Linux**. Licencias libres, sin soporte contratado.

- 1 **OmniGraffel 5**. Licencia comercial con soporte incluido. 144€y un año de vida estimada (actualización).

- 1 **OmniPlan**. Versión sin licencia limitada.

Material hardware

- 2 servidores **Dell PowerEdge SC1435** (1 reutilizado de la empresa)

  Memoria: 16GB

  Almacenamiento: 2 x 1 Terabyte (RAID-1 por software) + Controladora SAS* (LSI Logic SAS 1068)

  Precio unitario: 2.699 €

  Vida estimada: 4 años en renovación

- 7 portátiles **MacBook Pro 13"**

  Memoria: 4GB

  Almacenamiento: 250GB SATA

  Precio unitario: 1.399 €

  Vida estimada: 4 años

- 7 discos duros Fujitsu 500GB

  Precio unitario: 149 €

  Vida estimada: 4 años

A partir de estos datos de costes mostrados se obtiene el presupuesto global de todo el proyecto de la implantación y documentación, con los costes de personal asociado al proyecto y el material empleado para instalar y configurar todos los elementos. Se ha tenido en cuenta la vida estimada en los costes materiales, calculando así el coste de uso de los mismos en las 1290 de proyecto. Se obtiene entonces los presupuesto de la tabla A.3.

Tabla A.2: *Costes de material*

Material hardware	
Portátiles	9.793 €
Servidores	2.699 €
Almacenamiento	1.043 €
Material software	
Integración Continua	0 €
Herramientas de desarrollo	0 €
Sistemas operativos	331,21 €
IDEs	0 €
Herramientas de edición	92,88 €

Este coste está asociado al caso de estudio del proyecto, que se realizó sobre una empresa de desarrollo en su inicio de vida laboral, por lo que en otras implantaciones los costes de material deberían ser distintos en función del aprovechamiento de material existente y su adaptación, dependiendo por tanto, como es lógico, del diseño particular del caso.

Tabla A.3: *Presupuesto*

Concepto	**Importe**
Costes personal	51.600 €
Costes material	4.671,09 €
Base imponible	56.271,09 €
I.V.A. (18 %)	10.128,80 €
TOTAL	66,399,88 €

APÉNDICE B

INSTALACIONES Y CONFIGURACIONES

B.1. Repositorio Mercurial

Como gestor de repositorio de código distribuido las instalaciones de mercurial en las máquinas que la utilizan hacen el rol de servidor para poder gestionar el código de forma distribuida. Pero como en el diseño se ha optado por tener un servidor central para centralizar el uso del repositorio, se debe instalar un servidor web para poder visualizar el estado del código a través de un navegador.

B.1.1. Instalación

En el caso de los clientes lo único que hay que hacer es descargar el paquete de instalación de mercurial según el sistema operativo y directamente el comando `hg` ya está listo para estar en funcionamiento.

En el servidor de repositorio de código se instala un servidor web. En el caso de estudio se instala un *Apache HTTP Server*, siguiendo la documentación de su página web[1]. Una vez instalado el servidor hay que prepararla estructura de directorios.

[1] Apache web server: http://httpd.apache.org/

```
$ sudo mkdir -p /var/hg/repos
$ sudo chown -R webservd:webservd /var/hg
```

Lo primero que hay que hacer es generar los repositorios que sean necesarios mediante:

```
$ sudo -u webservd hg init /var/hg/<nombre-repositorio>
```

B.1.2. Configuración

El fichero `.hgrc`

Este fichero contiene información sobre el nombre del desarrollador que queda asociado al código gestionado mediante los comandos de Mercurial: commit, push, etc... Tiene este formato:

```
[ui] username = Usuario <usuario@abstra.cc>
```

También se puede personalizar el comportamiento de Mercurial para cada repositorio editando el fichero .hgrc del directorio de trabajo. Una opción global interesante, para que Mercurial sea un poco más locuaz en su operación, es -v, que puede seleccionarse por defecto en el archivo de configuración mediante el apartado verbose del bloque [ui]. Completando el ejemplo anterior:

```
[ui] verbose=true
username=''Nombre Apellido <usuario@abstra.cc>''

[paths] default=https://aedea.abstra/BlueMountain\
  /Common/Server/ruote-rest
```

Push con Mercurial y Apache

En principio cada directorio del repositorio debe contener un fichero .htaccess con las siguientes instrucciones:

```
AuthName "Mercurial repositories"
AuthType Basic
AuthBasicProvider ldap
AuthLDAPURL "ldap://europa.abstra:1389/ou=People,
  o=abstra?uid"
AuthLDAPBindDN cn=mercurial,ou=System,o=abstra
AuthLDAPBindPassword ********
AuthzLDAPAuthoritative on

<LimitExcept GET>
  Require ldap-user
</LimitExcept>
```

En este ejemplo sólo se indica que se necesita un usuario válido LDAP.

Nota importante: es necesario instalar el paquete `SUNWapu13-ldap` en el servidor OpenSolaris, ya que en caso contrario, el módulo de autenticación LDAP de Apache provoca fallos (*segmentation faults*).

El fichero `.htaccess` y `hgrc`

Con estos dos ficheros se restringe el acceso "push.ª un repositorio. Un ejemplo de fichero de configuración de repositorio sería el siguiente:

```
[web] contact = XXXXXX
description = XXXXXXXXX
name = Simple Test
```

```
style= gitweb
allow_push = *
push_ssl = false
```

donde se indican ciertos datos que describen el proyecto. En la cláusula `allow_push` se especifican los usuarios que pueden hacer "push" con Mercurial. Con el * se permite que todo usuario pueda hacer "push". que todo usuario pueda hacer push. Sin embargo, el fichero .htaccess controla qué usuarios pueden efectivamente pueden realizar"pus".

El fichero `.hgignore`

Es posible omitir ciertos ficheros/extensiones al hacer "add". Basta con crear el fichero `.hgignore` en el mismo nivel que el directorio `.hg`. Para los proyectos, basta con incluir:

```
syntax: glob
tmp/
foo.rb
bar/baz.java
work*/
*.swp
#.*#
.git.*
syntax: regexp
^.*.o$
```

B.2. Hudson CI

B.2.1. Instalación

En la instalación de Hudson se utilizaron distintas configuraciones, tanto de las tareas como del propio servidor de Integración Continua. Éste está configurado en el entorno In-

tegración y tiene el control de todas las tareas automatizadas de la Integración Continua de todo el código del proyecto, sea del entorno que sea.

Está corriendo sobre Glassfish y se ha creado un usuario llamado çis"(por Continuous Integration Server) el cual almacena toda la configuración en su directorio home (/var/ci/cis). El directorio por defecto .hudson está asignado en una variable de instancia en Glassfish. Para ello se hace lo siguiente en la consola de administración de Glassfish (io.abstra:4848) :

1. En el menú *Common Tasks*: `Configuration >System Properties`
2. `Instance Variable Name: HUDSON_HOME`
3. `Default Value: /var/ci/cis/.hudson`

Se despliega entonces Hudson sobre Glassfish (acción `deploy`) y ya usa la variable de entorno que determinada, así no almacena las configuraciones en `root`.

B.2.2. Configuración

Para configurar Hudson, según su manual, el cual puede obtenerse libremente en la página web http://hudson-ci.org/, se debe instalar primeramente los *plugins* necesarios para adaptarlo a la arquitectura determinada y según las necesidades de los distintos proyectos. Se accede a la configuración de estos navegando en el servidor `Manage Hudson >Plugin Manager`. Los plugins necesarios para la instalación estudiada son:

- *DashBoard View*
- *Deploy to container Plugin*
- *Disk Usage Plugin*
- *Downstream buildview plugin*

- *Locale plugin*
- *Hudson M2 Extra Steps Plugin*
- *Hudson Maven Info Plugin*
- *Maven 2 Project Plugin*
- *Mercurial Plugin*
- *Nested View Plugin*
- *Performance Plugin*
- *Rake plugin*
- *Redmine plugin*
- *Ruby metrics plugin*
- *slave-status*
- *Hudson Sonar Plugin*
- *SSH Slaves Plugin*
- *Hudson Status Monitor*

Todos ellos se emplean para funcionalidades necesarias tanto de integración con las herramientas de la arquitectura, como de monitorización como de configuración especial de algunas tareas de automatización en la Integración Continua.

Las configuraciones globales de Hudson son las que se pueden ver en las imágenes B.1, B.2, B.4, B.3 y B.5.

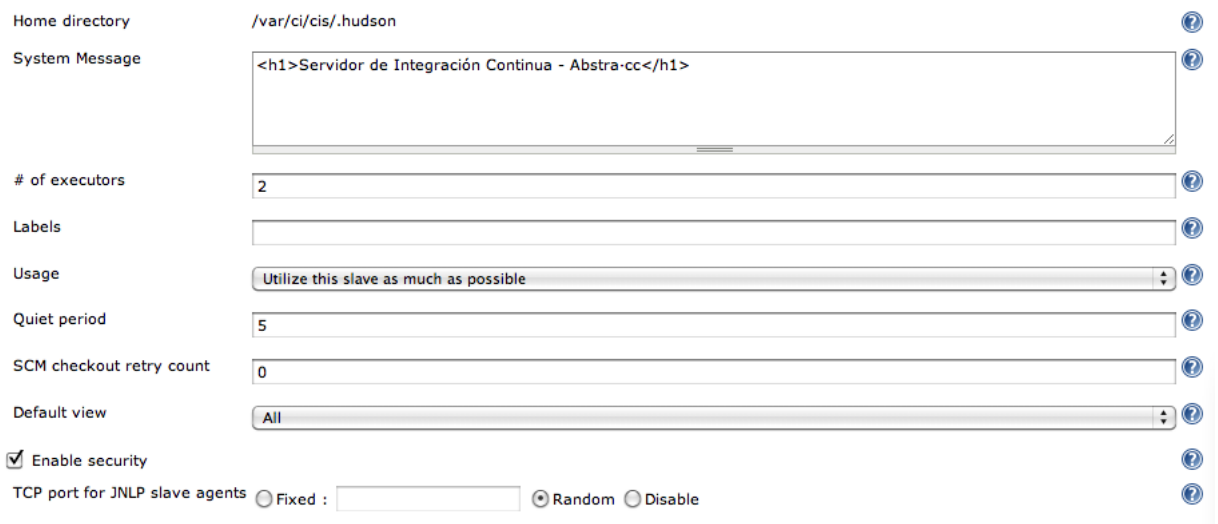

Figura B.1: Configuración Inicial Hudson.

B.3. Maven

En la instalación y configuración de Maven es esencial configurar de forma correcta la integración con Nexus y con la arquitectura de desarrollo según gestione los repositorios y artefactos necesarios para la programación.

B.3.1. Instalación Maven

Según los distintos sistemas operativos se debe tener en cuenta algunas consideraciones. En el caso de estudio las instalaciones de los equipos de programación y servidores se encontraban bajo Linux, Unix o Mac OS X.

Maven en Mac OS X

Normalmente Maven solía encontrarse en en el Mac en `/usr/share/maven`, pero desde la versión 10.5.8 se encuentra en el directorio `/usr/share/java/maven`. La versión de Maven instalada por defecto en Mac OS 10.5.8 es la 2.0.9, que es bastante anterior a las nuevas versiones. Entonces para tener la última versión, descargable desde Apache, sólo se

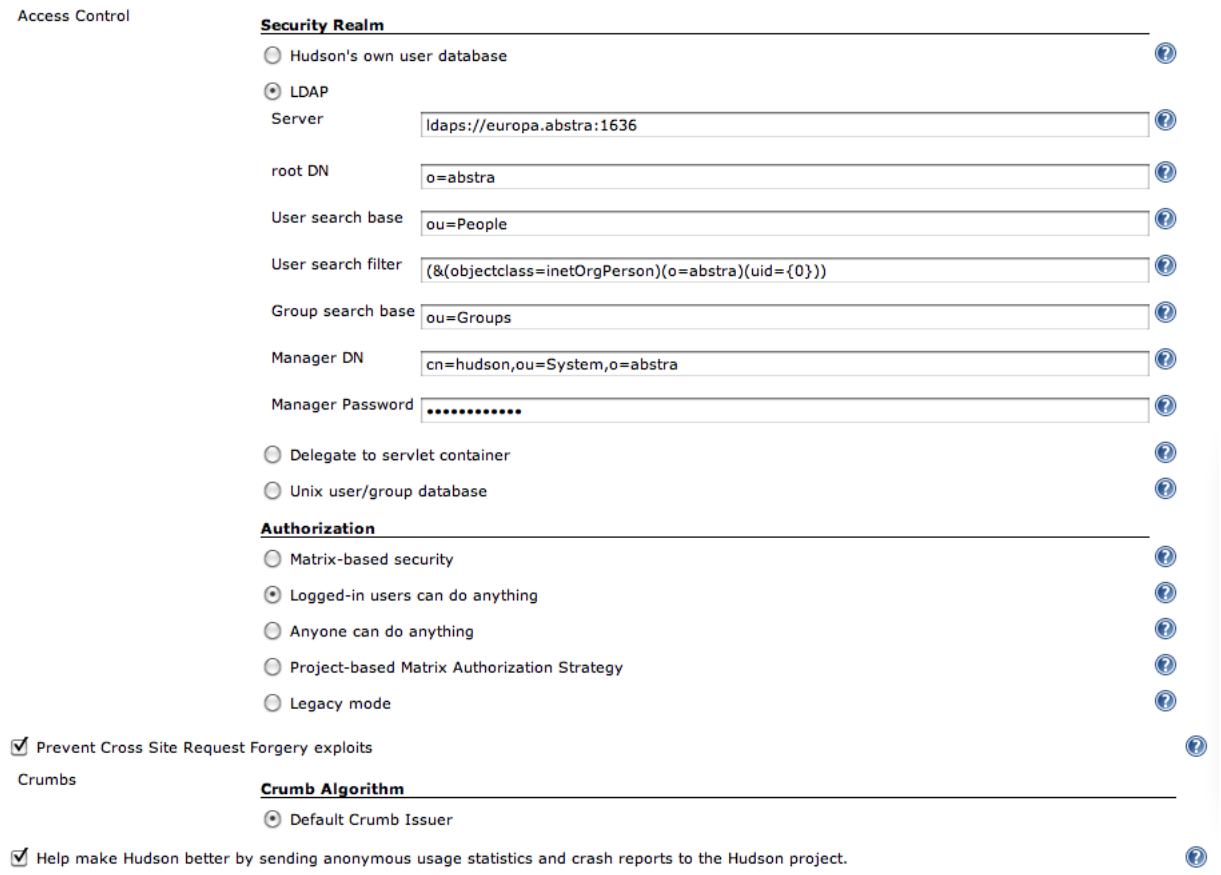

Figura B.2: Configuraciones seguridad Hudson.

tiene que crear un enlace simbólico en `/usr/share/java/maven` apuntando al directorio de la nueva versión descargada.

Es aconsejable que los distintos directorios de actualizaciones anteriores y nuevas se almacenen en el mismo directorio y a ser posible en `/usr/share/java` para así mantener la consistencia inicial y si hay que volver a una versión anterior sólo se necesite cambiar el enlace simbólico desde el mismo directorio donde se encuentran todas las versiones de Maven utilizadas.

Las variables de entorno `M2` y `M2_HOME` en el Mac no hay que configurarlas, pero sí hay que configurar `MAVEN_OPTS` para que Java no produzca excepciones de memoria en proyectos más o menos grandes. Un valor típico suele ser `MAVEN_OPTS=Xms256m -Xmx512m"`.

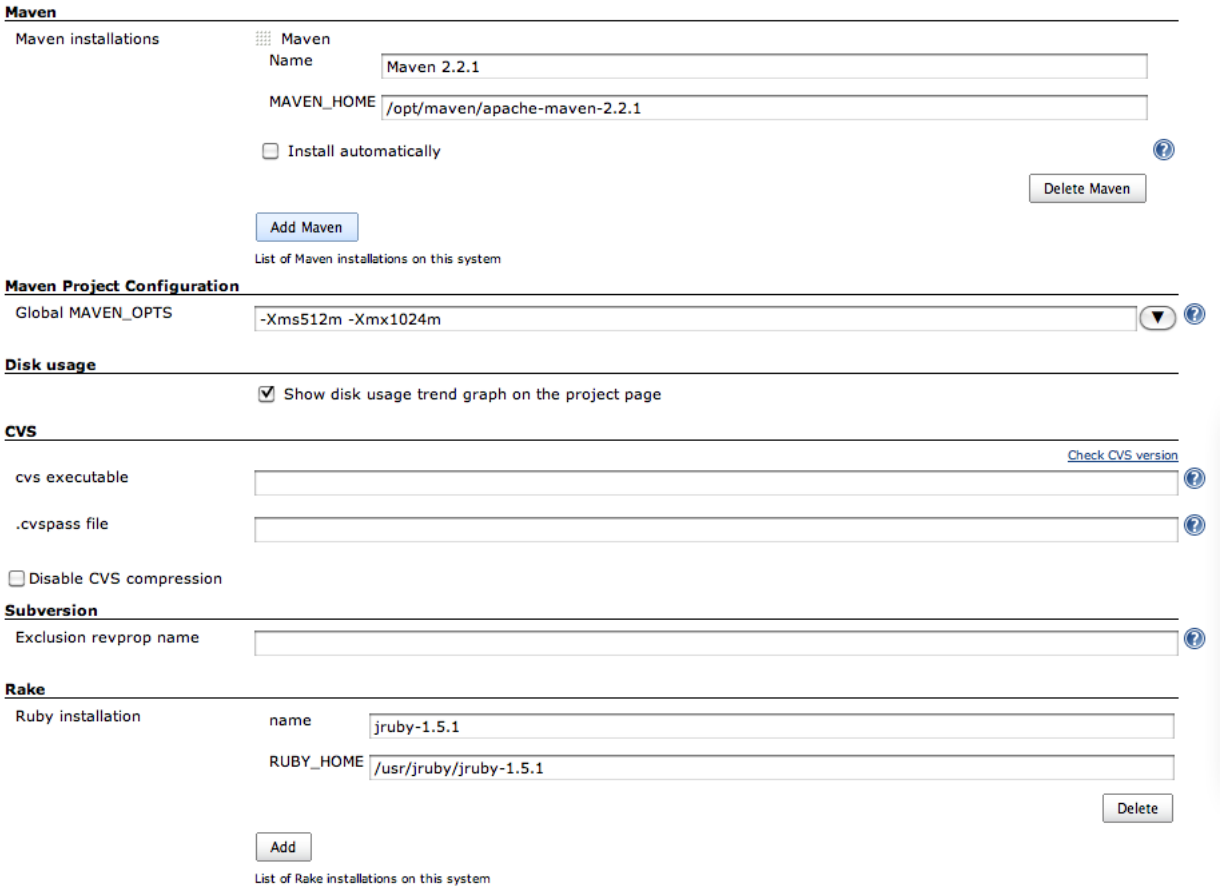

Figura B.3: Configuraciones Maven y Rake de Hudson.

Para que la variable de entorno en el Mac se utilice desde entorno gráfico también hay que configurarla en el archivo `environment.plist`, y situarla en el directorio oculto `.MacOSX` del directorio *home*.

Maven Linux/Unix

No se debe tener ninguna configuración extra en cuenta, ya que la documentación en Apache [3] es bastante buena y extensa para configurarlo.

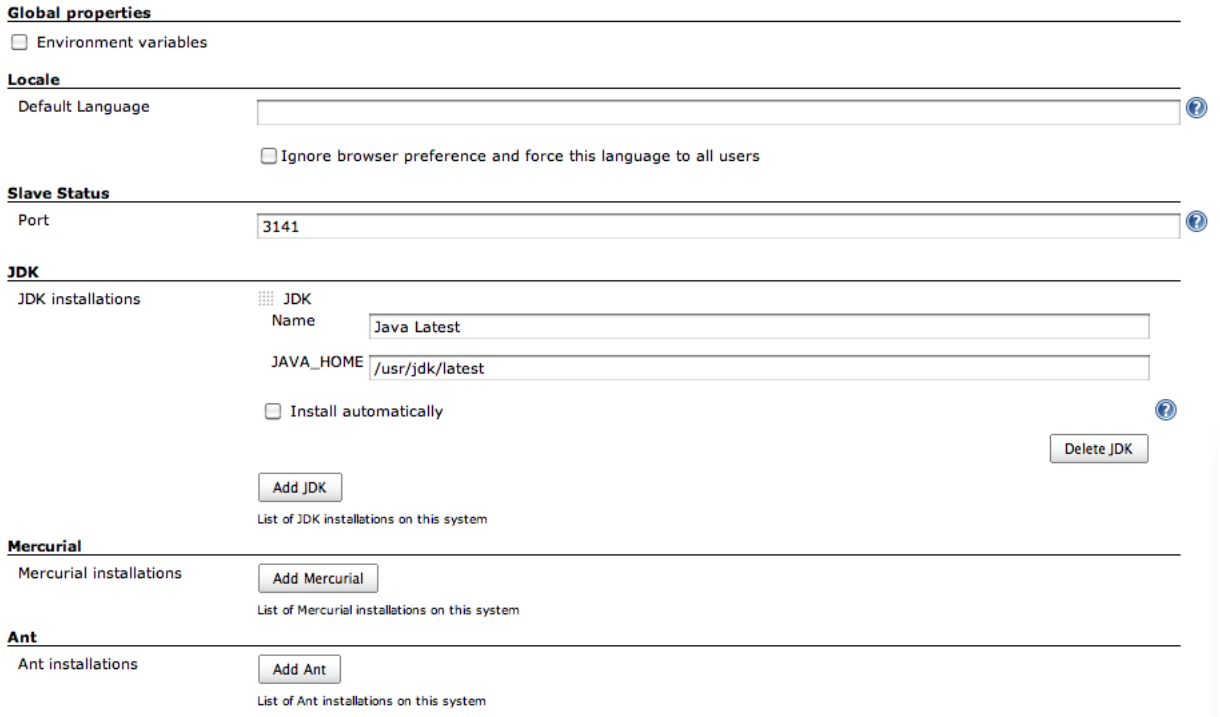

Figura B.4: Configuraciones herramientas desarrollo de Hudson.

B.3.2. Instalación Nexus

Para la configuración de la instalación de Nexus en la zona del servidor OpenSolaris se debe tener en cuenta que se ha realizado una instalación *standalone*, lo que consigue desplegarse ella misma sobre un servidor de aplicaciones *Jetty*. También existe un archivo tipo `WAR` para desplegar sobre otro servidor de aplicaciones, pero entra en conflicto si se usa con *Glassfish*, que es empleado como servidor de aplicaciones en la arquitectura global de la empresa. Para la instalación hay que tener en cuenta:

1. Nexus se encuentra en el directorio `/opt/nexus`
2. El directorio donde se encuentran las configuraciones es `/opt/nexus/sonatype-work`
3. El usuario *nexus* es quien arranca y para la aplicación
4. Se ha creado un servicio en OpenSolaris para arrancarlo, pararlo y reiniciar de forma automática

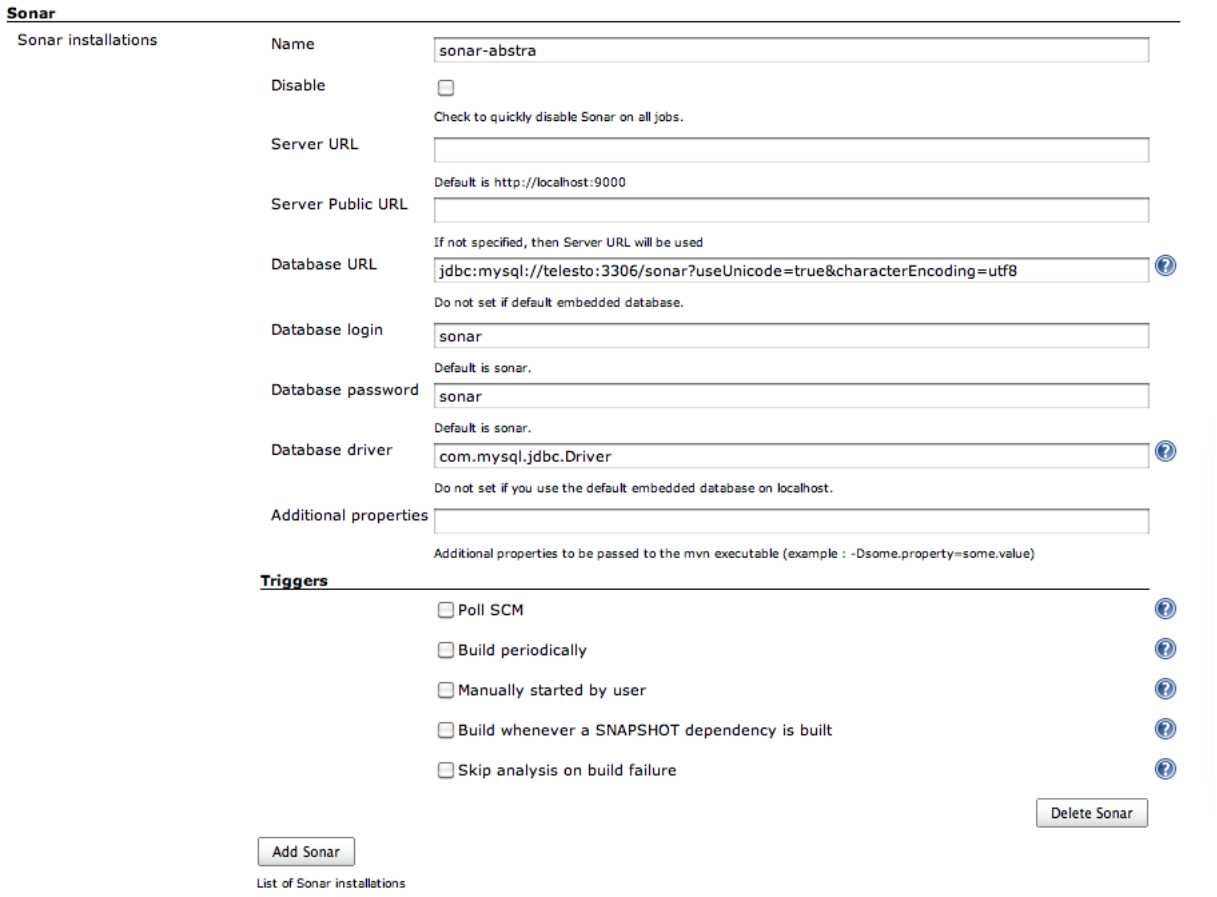

Figura B.5: Configuraciones integración Sonar y Hudson.

5. La configuración del servicio en OpenSolaris se encuentra en `/var/svc/manifest/network/nexus.xml`

6. Las configuraciones propias de Nexus se encuentran en el directorio `/opt/nexus/sonatype-work`

7. Para arrancar Nexus, se ejecuta en el servidor:
   `svcadm enable svc:/network/http:nexus`

8. Para deshabilitar Nexus, se ejecuta en el servidor:
   `svcadm disable svc:/network/http:nexus`

La configuración del servicio que se encuentra en `/var/svc/manifest/network/nexus.xml` es la siguiente:

```
<?xml version='1.0' ?>
<!DOCTYPE service_bundle SYSTEM '/usr/share/lib/xml/dtd/\
service_bundle.dtd.1'>
<!--
    Service Manifest for Nexus
-->

<service_bundle type='manifest' name='SUNWapchr:nexus' >
    <service name='network/http' type='service'
    version='1'>
        <instance name='nexus' enabled='false'>

            <dependency name='loopback'
            grouping='require_all' restart_on='error'
            type='service'>
                <service_fmri value='svc:/network/\
                loopback:default' />
            </dependency>
            <dependency name='physical'
            grouping='optional_all' restart_on='error'
            type='service'>
                <service_fmri value='svc:/network/\
                physical:default' />
            </dependency>

            <dependency name='localfs'
            grouping='require_all'
            restart_on='error' type='service'>
                <service_fmri value='svc:/system/\
                filesystem/local:default' />
```

```
        </dependency>

    <!--
        These privileges allow the service to run as
        user nexus/group nexus from the beginning.
        The net_privaddr privilege allows the start
        method to run with the ability bind to
        privileged ports (in this case, we only
         care about 80 and 443).  However,
        if one is logged in as 'www', one does
        not have this privilege.
    -->
        <exec_method name='start' type='method'
        exec='/opt/nexus/current/bin/jsw/\
        solaris-x86-32/nexus start'
        timeout_seconds='60'>
            <method_context>
                <method_credential user='nexus'
                group='nexus'
                privileges='basic,!proc_session,
                !proc_info,!file_link_any,
                net_privaddr' />
            </method_context>
        </exec_method>

        <exec_method name='stop' type='method'
        exec='/opt/nexus/current/bin/jsw/\
        solaris-x86-32/nexus stop'
        timeout_seconds='60'>
            <method_context />
        </exec_method>
```

```
            <exec_method name='refresh' type='method'
            exec='/opt/nexus/current/bin/jsw/\
            solaris-x86-32/nexus restart'
            timeout_seconds='60'>
                <method_context />
            </exec_method>

     </instance>
<stability value='Evolving' />
<template>
    <common_name>
        <loctext xml:lang='C'>Sonatype Nexus Maven
         Repository Manager </loctext>
    </common_name>
    <documentation>
        <doc_link name='sonatype.org'
        uri='http://nexus.sonatype.org' />
    </documentation>
</template>
</service>
</service_bundle>
```

B.3.3. Configuracion de integración Maven y Nexus

Cada vez que Maven tenga que resolver una dependencia y bajarse un artefacto lo hará a través de Nexus, ya sea del repositorio central de Maven http://repo1.maven.org/maven2 o de cualquier otro repositorio que se haya añadido en el POM del proyecto. Para ello se debe configurar Maven en la máquina local para que esto sea así.

Habrá que cambiar el archivo de configuración local de Maven, localizado en -/.m2/settings.xmlz

decirle que todas las peticiones de dependencias vayan al *mirror* principal, que es Nexus. Para ello el archivo `settings.xml` quedaría de la forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
    <!-- Configuración Nexus -->
    <mirrors>
      <mirror>
          <!--
              Este mirror se crea para que los snapshots
               se dirijan a través del
              grupo public-snapshots
          -->
          <id>nexus-public-snapshots</id>
          <mirrorOf>public-snapshots</mirrorOf>
          <url>http://aedea.abstra:8081/nexus/\
          content/groups/public-snapshots
          </url>
      </mirror>
      <mirror>
          <id>nexus</id>
          <!-- Por defecto todos los artefactos pasan a
           través del grupo public -->
          <mirrorOf>*</mirrorOf>
          <url>http://aedea.abstra:8081/nexus/\
          content/groups/public</url>
      </mirror>
    </mirrors>
    <profiles>
      <profile>
          <id>nexus</id>
          <!--
```

```
        Habilitamos los snapshots para compilaciones
         en el repo central para
        dirigir todas las peticiones mediante
        el mirror
    -->
    <repositories>
        <repository>
            <id>central</id>
            <url>http://central</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
                    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>central</id>
            <url>http://central</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>
</profile>
<profile>
```

```
        <!--Con este perfil permitimos que se busquen
         los snapshots en su repo-->
        <id>public-snapshots</id>
        <repositories>
            <repository>
                <id>public-snapshots</id>
                <url>http://public-snapshots</url>
                <releases>
                    <enabled>false</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                     <updatePolicy>always</updatePolicy>
                </snapshots>
            </repository>
          </repositories>
          <pluginRepositories>
            <pluginRepository>
                <id>public-snapshots</id>
                <url>http://public-snapshots</url>
                <releases>
                    <enabled>false</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                    <updatePolicy>always</updatePolicy>
                </snapshots>
            </pluginRepository>
        </pluginRepositories>
      </profile>
    </profiles>
```

```
    <activeProfiles>
      <!-- Hay que decirle que active los perfiles -->
      <activeProfile>nexus</activeProfile>
      <activeProfile>public-snapshots</activeProfile>
    </activeProfiles>
</settings>
```

Desplegar proyectos Maven propios a Nexus

Como los proyectos Maven generados serán parte o módulos de otros, los artefactos que se generan tendrán que publicarse en los repositorios locales alojados mediante Nexus. Para este fin existen dos repositorios distintos:

- Repositorio de *releases*[2], denominado en Nexus *Abstra Releases*
- Repositorio de *snapshots*[3], denominado en Nexus *Abstra Snapshots*

Por defecto Maven guarda los artefactos en el repositorio local de usuario en el directorio `.m2/repository` dentro del directorio propio del usuario, pero para compartirlo, según el proyecto, se deberá indicar en el `pom.xml` de dicho proyecto. Es importante tener en cuenta que depende de si la versión es una *release* o no se desplegará en un repositorio u otro. Para poder desplegarlo se necesita añadir en el `pom.xml` ciertas partes en función de si la versión es una release o un snapshot.

En el caso de una *release*:

```
<project>
  ...
  <distributionManagement>
    ...
```

[2]Versiones estables.

[3]Versiones inestables en desarrollo.

```
    <repository>
      <id>releases</id>
      <name>Abstra Releases</name>
      <url>http://aedea.abstra:8081/nexus/content/repositories/
      releases</url>
    </repository>
    ...
  </distributionManagement>
  ...
</project>
```

En el caso de un *snapshot*:

```
<project>
  ...
  <distributionManagement>
    ...
    <snapshotRepository>
      <id>snapshots</id>
      <name>Abstra Snapshots</name>
      <url>http://aedea.abstra:8081/nexus/content/repositories/
      snapshots</url>
    </snapshotRepository>
    ...
  </distributionManagement>
  ...
</project>
```

Una vez configurado el despliegue del archivo `pom.xml` hay que indicarle a Maven en el archivo de configuración `settings.xml` (`.m2/settings.xml`) que despliegue con el usuario de despliegue de Abstra en Nexus. Se añade entonces en dicho archivo:

```
<servers>
  <server>
    <id>releases</id>
    <username>abstra</username>
    <password>*******</password>
  </server>
  <server>
    <id>snapshots</id>
    <username>abstra</username>
    <password>*******</password>
  </server>
</servers>
```

Una vez configurado esto sólo hay que desplegar mediante Maven con el comando `deploy`, ya sea a través del IDE o mediante línea de comandos (`mvn deploy`).

B.3.4. Configuración de Nexus

La configuración de Nexus se realiza para:

1. Añadir repositorios usados en el desarrollo para utilizar los artefactos Maven necesarios
2. Integrarse con el sistema de acceso de usuarios para configuración y consulta de repositorios
3. Consultar los artefactos necesarios dentro de la arquitectura de la empresa
4. Desplegar artefactos o proyectos propios creados por el equipo de desarrollo

En la configuración del caso de estudio se determinan los roles de los usuarios en el proxy de Nexus según el caso de uso de cada integrante del equipo de desarrollo, creando

tres administradores, con acceso y permisos para poder realizar cualquier cambio y el resto de programadores con permisos para poder añadir o editar repositorios, configurar despliegues propios y configurar sus personalizaciones de usuario. Los administradores poseen todos los permisos, como edición de usuarios, de roles, de repositorios, del servidor Nexus, etc...

Los tres administradores de Nexus son el responsable de QA, el arquitecto de la aplicación software y el director técnico del equipo. Para configurar Nexus se hace de forma sencilla mediante un servidor web propio en su instalación, cuya configuración se puede ver en la figura B.6.

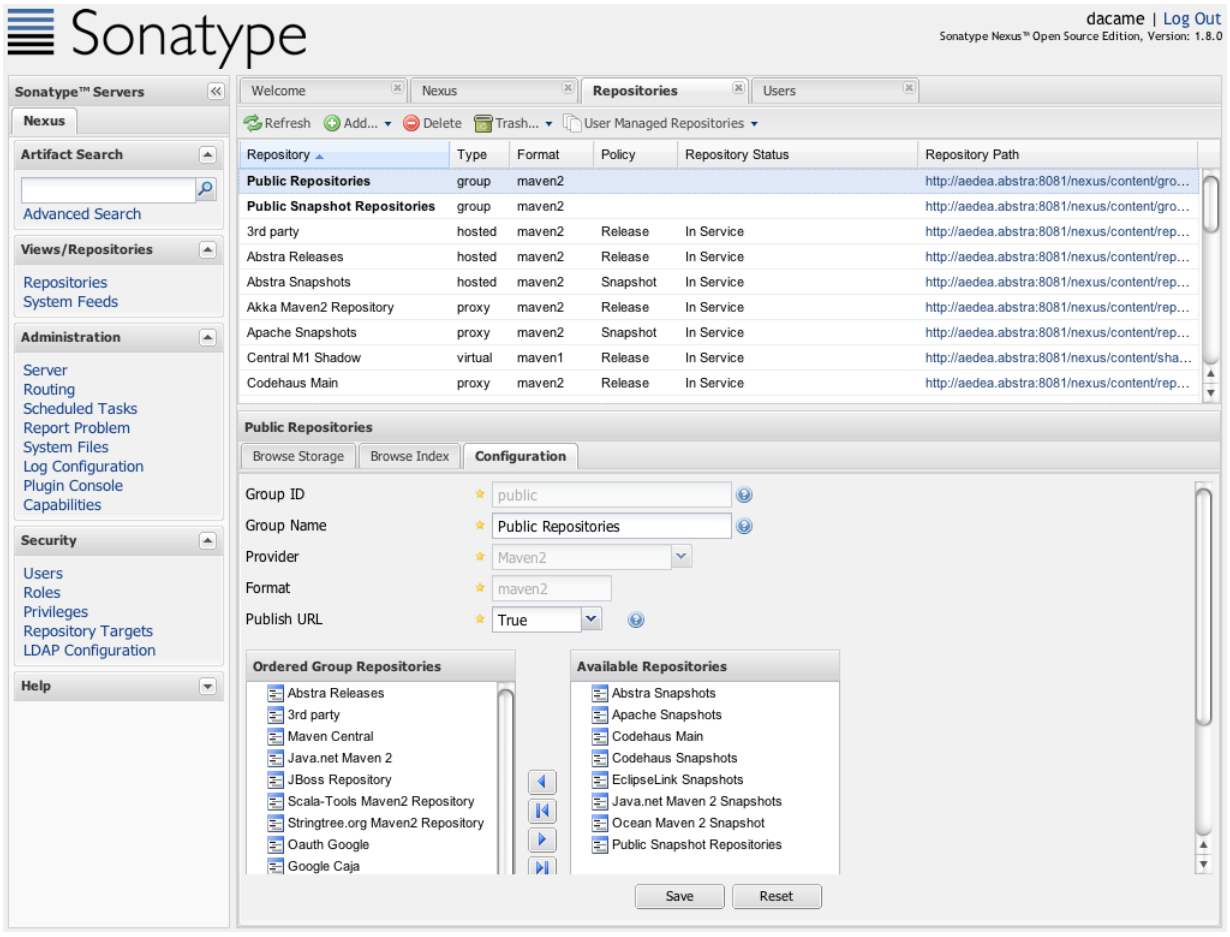

Figura B.6: Configuración de repositorios Nexus.

La forma de configurar Nexus y adaptarlo está muy bien documentada en la web de Sonatype Nexus http://nexus.sonatype.org/documentation.html, donde se puede descargar gratuitamente con suscripción el libro completo de la herramienta.

B.4. Entornos de Desarrollo Integrado

Los Entornos de Desarrollo Integrado, IDEs, del caso de estudio son fáciles de configurar para la adaptación de la arquitectura. Sólo hay que tener en cuenta que la integración se realiza con Mercurial, Maven y los lenguajes de programación empleados para cada caso.

Las configuraciones de lenguajes de programación se realizan normalmente por cuenta del programador, ya que el él el que normalmente tiene un profundo conocimiento del lenguaje y tiene claras las necesidades y requisitos para programar con éxito mediante dicho lenguaje.

En el caso de estudio se adaptaron a la arquitectura los IDEs NetBeans y Eclipse sólo teniendo en cuenta las instalaciones de Maven como herramienta del ciclo de vida y Mercurial como gestor de repositorio de código. Así cada vez que se necesita actualizar el código central en Mercurial no es necesario acudir al cliente de éste, sino que el propio IDE lo ejecuta.

B.4.1. Eclipse

En el caso de Eclipse Maven se instala mediante el *plugin* **m2eclipse**, desarrollado por la propia empresa Sonatype, la cual es creadora del propio Maven. Para realizarlo la documentación [21] tiene un capítulo muy bien documentado y dedicado a integrar Maven en Eclipse.

Lo que se consigue con este plugin es:

- Editar el archivo de configuración de proyectos Maven (POM) mediante un editor gráfico bastante intuitivo.
- Posibilidad de visualizar las dependencias globales del proyecto de desarrollo que gestiona Maven, tal y como se puede ver en la figura B.7.
- Personalizar ejecuciones de Maven según necesidades.

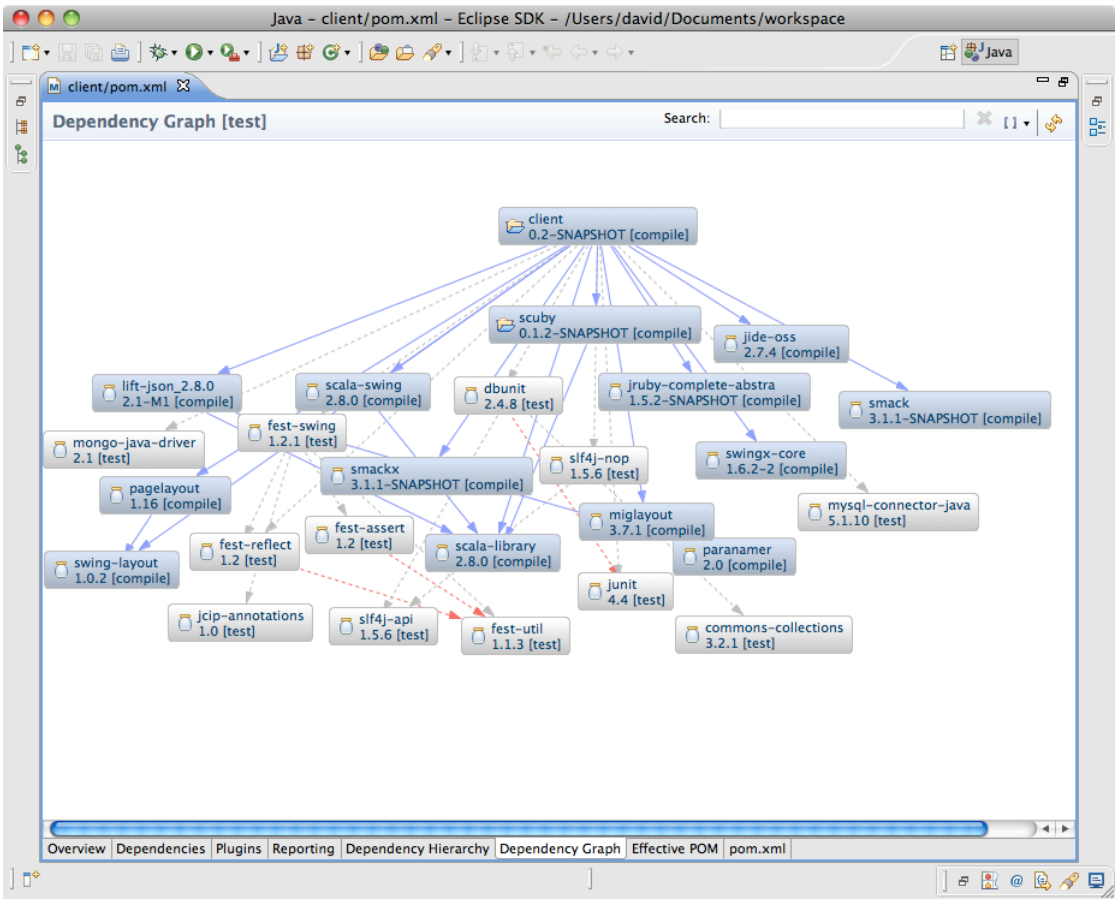

Figura B.7: Visualización de dependencias de artefactos en Eclipse.

El otro plugin importante a instalar es el de la herramienta Mercurial, que nos dará la opción de poder realizar las tareas básicas de actualización del código según el gestor de repositorio. El instalado en la arquitectura se denomina **MercurialEclipse**, y se puede

hacer la instalación mediante el *Eclipse Marketplace*, opción del IDE instalada también mediante una extensión, que permite hacer búsquedas e instalaciones de distintas extensiones en los repositorios de *plugins* de Eclipse por defecto.

Se puede encontrar amplia documentación para realizarlo en http://www.eclipse.org/documentation/.

B.4.2. NetBeans

NetBeans, como entorno de desarrollo, es muy similar a Eclipse, sólo con algunas diferencias de aspecto y de funcionalidad integrada. Así por ejemplo, para el caso que ocupa, Maven está incluido en la instalación por defecto de NetBeans y no es necesario instalarlo para integrarlo. Sólo hay que configurar las personalizaciones necesarias:

- Ejecución de *goals* específicos
- Parámetros de construcción dependientes del proyecto
- Perfiles utilizados por el equipo de desarrollo
- Variables o plugins de Maven específicos

La configuración de Maven en NetBeans es muy fácil e intuitiva de configurar, como se observa en la figura B.8. Está muy bien documentado en la página http://wiki.netbeans.org/MavenBestPractices, perteneciente al proyecto NetBeans.

Por otra parte Mercurial también está integrado en la instalación básica de NetBeans, sólo teniendo que saber las distintas opciones básicas del repositorio de código. Lo que hace NetBeans es identificar si el sistema tiene configuraciones tipo Mercurial, según se especifican en la sección B.1, y automatiza las acciones para sincronizar el código.

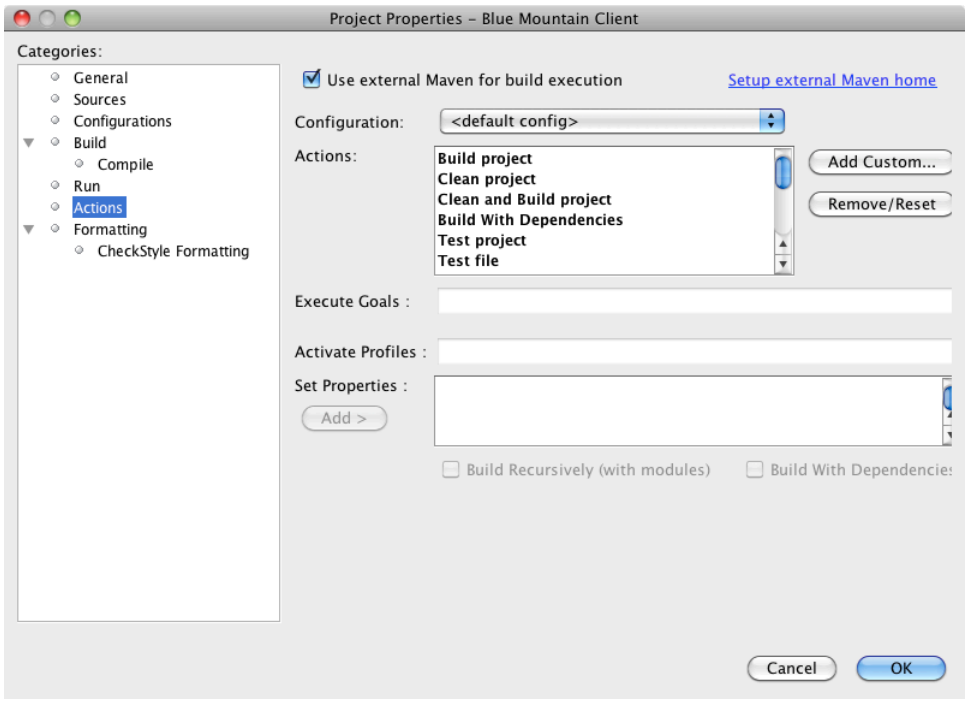

Figura B.8: Configuración de *goals* en NetBeans.

B.5. Otras Instalaciones

Por demanda del caso de estudio se integró una herramienta de análisis del código específica para la gestión de calidad del proyecto a desarrollar. Esta herramienta es Sonar (http://www.sonarsource.org/), de licencia *open source* gratuita.

B.5.1. Sonar

La misión de Sonar es ofrecer estadísticas del código de proyectos para poder hacer un seguimiento en su calidad mucho más exhaustivo. En este caso el seguimiento se realiza sobre el código Java utilizado en los proyectos involucrados en el desarrollo del proyecto global.

Para la instalación de Sonar se ha seguido instrucciones de su página web (http://docs.codehaus.org/display/SONAR/Install+Sonar), excepto en el paso cargar el

plugin de Maven, el cual no se utilizará el argumento sonar:sonar, sino la llamada completa al plugin, según la estructura de comandos de Maven.

Para instalar Sonar se siguen los pasos a continuación:

1. Descargar y descomprimir en el directorio de la zona de Integración Continua del servidor OpenSolaris `/opt/sonar`, configurando archivos para conectar con la base de datos.

2. Configurar la base de datos en Telesto (servidor virtual de base de datos en VMWare ESXi). Este caso es una base de datos MySQL de distribución libre.

3. Crear un servicio en OpenSolaris para arrancar Sonar.

4. Configurar las tareas Maven de Hudson para que use Sonar como herramienta de gestión de la calidad del código.

Configuración de Maven y Sonar

En el archivo de configuración `settings.xml` de Maven para el equipo de Integración Continua donde se ejecutan las tareas automatizadas se debe añadir un perfil de ejecución de la forma:

```
...
<profiles>
  <profile>
    <id>sonar</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <!-- Database at Telesto for Sonar use-->
```

```
      <sonar.jdbc.url>
        jdbc:mysql://telesto:3306/sonar?useUnicode=true&amp;
        characterEncoding=utf8
      </sonar.jdbc.url>
      <sonar.jdbc.driver>com.mysql.jdbc.Driver</sonar.jdbc.driver>
      <sonar.jdbc.username>sonar</sonar.jdbc.username>
      <sonar.jdbc.password>sonar</sonar.jdbc.password>

      <!-- SERVER ON A REMOTE HOST -->
      <sonar.host.url>http://localhost:9000</sonar.host.url>
    </properties>
  </profile>
</profiles>
...
```

En el servidor Sonar, donde se despliega la instalación, se debe configurar el archivo `/opt/sonar/sonar-2.1.2/conf/sonar.properties`, de forma que se desconfigura la parte de la base de datos Derby y se configura la parte MySQL de la forma siguiente:

```
...
# MySql # uncomment the 3 following lines to use MySQL
sonar.jdbc.url:  jdbc:mysql://telesto:3306/sonar?
useUnicode=true&characterEncoding=utf8
sonar.jdbc.driverClassName: com.mysql.jdbc.Driver
sonar.jdbc.validationQuery: select 1
...
# generic settings
sonar.jdbc.username: sonar
sonar.jdbc.password: sonar
sonar.jdbc.maxActive: 10
sonar.jdbc.maxIdle: 5
```

```
sonar.jdbc.minIdle: 2
sonar.jdbc.maxWait: 5000
sonar.jdbc.minEvictableIdleTimeMillis: 600000
sonar.jdbc.timeBetweenEvictionRunsMillis: 30000
...
```

Una vez configurada esta parte, en el gestor de repositorio Nexus hay que añadir el repositorio con la configuración siguiente:

- *Format* : Maven2
- *Repository Policy* : Release
- *Remote Storage Location* : http://io.abstra:9000/deploy/maven
- *Download Remote Indexes* : False

Estadísticas Sonar

Un ejemplo de las estadísticas principales que nos ofrece Sonar se ven en la figura B.9, que presenta los datos estadísticos de código del Data Channel del proyecto del ERP.

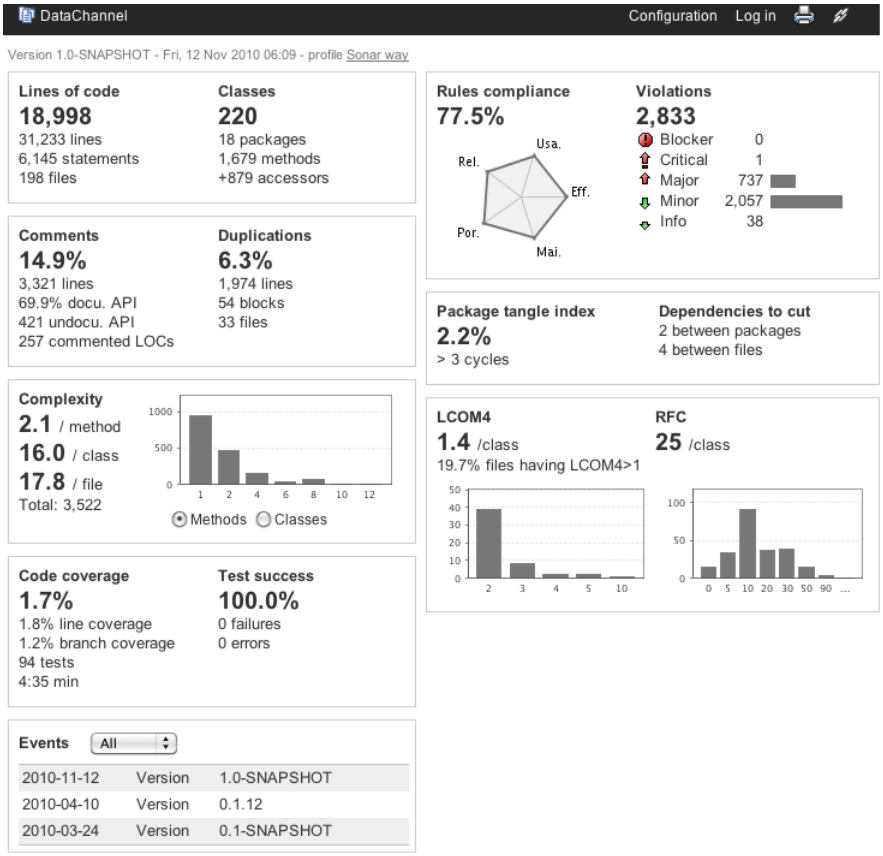

Figura B.9: Estadísticas del proyecto Data Channel.

GLOSARIO

SMED (Single-Minute Exchange of Die) Método de producción dirigido a eliminar tiempos muertos en los procesos.

5S Metodología de producción industrial enfocada a la optimización mediante cinco fases: colocación, organización, estandarización, limpieza y autodisciplina. En Japonés son cinco eses (Seiri, Seiton, Seiso, Seiketsu, Shitsuke)

Six Sigma Gestión de procesos y negocio para la mejora de la calidad mediante la eliminación de las causas de los defectos.

CONWIP (CONstant Work In Progress) Sistema de producción híbrido entre un tipo *pull* y un tipo *push*, mezcla de ajuste de demanda y de oferta.

CRM (Customer Relationship Management) Herramienta software para gestionar los clientes de una empresa y todos los recursos relacionador a ellos.

ERP (Enterprise Resource Planning) Herramienta software dedicada a gestionar y planificar todos los recursos de una empresa. Son aplicaciones modulares y muy costosas de implantar.

Just In Time Filosofía de producción "Justo a Tiempo", enfocada a producir en un entorno industrial donde la producción se ajusta a la demanda.

Kaizen Traducción japonesa de "Mejora Continua" y es una filosofía y metodología de producción industrial enfocada a la continua mejora de los procesos.

ASD (Adaptative Software Development) Desarrollo de Software Adaptativo es un tipo de metodología ágil de tipo iterativa.

FDD (Feature Driven Development) Desarrollo Dirigido a Funcionalidades es un tipo de metodología ágil basada en las funcionalidades a implementar.

XP (Extreme Programming) Metodología de programación denominada Programación Extrema, enfocada al desarrollo de software en equipo.

Scrum Metodología ágil de gestión de proyectos centrada en iteraciones según funcionalidad.

Agile Unified Process Metodología ágil que emplea el Desarrollo Dirigido a Tests.

DSDM (Dinamic Systems Development Method) Método de Desarrollo de Sistemas Dinámicos es una metodología ágil dirigida a entornos altamente cambiantes.

EssUP (Essential Unified Process) Metodología ágil para el desarrollo software de forma iterativa.

Lean Software Development Metodología ágil de gestión basada en métodos de ajuste de procesos a la demanda, que provienen de la metodología industrial y de gestión Lean Manufacturing.

The Crystal Methodologies Tipo de metodología ágil para el desarrollo software centrada en la gestión y el aspecto humano.

IID (Iterative and Incremental Method) Desarrollo Iterativo Incremental es un tipo de desarrollo software basado en las iteraciones como medio temporal y secuencial de desarrollo.

Commit Acción de publicar oficialmente en el repositorio de código cambios realizados al mismo.

CI (Continuous Integration) Integración Continua es la práctica de integración del código que propone unos procesos para automatizar de forma continua la integración del código.

IDE (Integrated Development Environment) Entorno de Desarrollo Integrado son herramientas informáticas para desarrollar software, de forma que se integra otras herramientas de desarrollo en la misma aplicación, de fácil manejo y con ayudas a la programación.

QA (Quality Assurance) Aseguramiento de la Calidad, referido al término de mantener y controlar un nivel de calidad determinado en cualquier proceso o procedimiento.

TDD (Test Driven Development) Desarrollo Dirigido a Tests, donde se programa de forma que los tests se desarrollan antes que el código, siendo usado como el fin de la funcionalidad.

API (Application Programming Interface) Interfaz de programación que se utiliza para interactuar con determinado software o aplicaciones.

Debugger Herramienta o aplicación con el fin de detectar errores y fallos en el código.

Lifecycle Build Tools Herramientas de construcción del ciclo de vida proporcionan las fases y ejecuciones necesarias para construir todo el ciclo de vida de una programación determinada.

XML (Extensible Markup Language) Lenguaje de programación para describir reglas mediante etiquetado.

Plugin Extensión instalable en cualquier software modularizado o extensible.

POM (Project Object Model) Elemento que modela las construcciones de código para la herramienta Maven.

WAR (Web Application Archive) Archivo de Aplicación Web, que tiene empaquetadas las funcionalidades de una aplicación que se despliega sobre un servidor de tipo web.

EAR (Enterprise Archive) Fichero que empaqueta los archivos necesarios de forma modularizada y que ejecuta aplicaciones web.

JAR (Java Archive) Fichero que empaqueta código Java para su ejecución.

Snapshot Versiones de código inestables que muestran funcionalidad rápidamente.

Release Versión de código estable, probada y lista para el lanzamiento.

DSL (Domain Specific Language) Lenguaje de programación enfocado a un problema específico de dominio.

Cloud Término referido a Internet como una nube dispersa con todo tipo de información disponible.

Open source Libre distribución y de código abierto.

LDAP (Lightweight Directory Access Protocol) Protocolo de acceso para directorio de usuarios.

Script Pequeño programas que automatiza secuencias de comandos o acciones software.

Transitario Empresa, persona que operadel sector de transporte internacional multimodal.

ECM (Enterprise Content Management) Gestión de Contenido Empresarial es un tipo de herramienta software que gestiona todos los contenidos documentales, digitales y de información de una empresa o corporación.

Bug Defectos en el software, ya sean de código o de funcionalidad.

Front End Capa del software que recolecta información del usuario y ofrece la funcionalidad visual. Es la capa de alto nivel en una aplicación software.

Back End Capa del software que procesa las especificaciones y funcionalidades que muestra el Front End.

Workflow Flujo de procesos, entendido como la consecución de acciones que producen una operación más global.

Enterprise Referido en inglés al término "empresarial", pero que abarca al mundo corporativo de las grandes empresas.

Bibliografía

[1] Agile manifesto. http://agilemanifesto.org.

[2] Implementing scrum web. http://www.implementingscrum.com.

[3] The Apache Software Foundation. Apache Projects. Website, 2010. http://apache.org.

[4] The Apache Software Foundation. Subversion. Website, 2010. http://subversion.apache.org.

[5] David Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, Upper Saddle River, NJ, 2005.

[6] Atlassian. Jira Repositorio Software. Website, 2010. http://www.atlassian.com/software/jira/.

[7] Mike Cohn. *Agile Estimating and Planning*. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, 2009.

[8] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, Boston, MA, 2010.

[9] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, Boston, MA, 2008.

[10] Carlos Dávila Piñeiro. Las TIC Aplicadas en el Transporte Internacional Multimodal. Proyecto Master, 2005-2006.

[11] Martin Fowler. Continuous integration. *MartinFowler.com Articles*, 2006. http://www.martinfowler.com/articles/continuousIntegration.html.

[12] James A. Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley, Boston, MA, 2002.

[13] Craig Larman. *Agile & Iterative Development: A Manager's Guide*. Addison-Wesley, Boston, MA, 2003.

[14] Martining and Associates. Continuous integration tools directory. http://www.continuousintegrationtools.com/.

[15] Vincent Massol; Timothy M. O'Brien. *Maven: A Developer's Notebook*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2005.

[16] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2009.

[17] Redmine. Redmine Issue Tracker. Website, 2006-2010. http://www.redmine.org/.

[18] Peter Schuh. *Integrating Agile Development in the Real World*. Charles River Media, Hingham, MA, 2001.

[19] Ken Schwaber and Jeff Sutherland. Scrum. http://www.scrum.org/scrumguides/.

[20] John Ferguson Smart. *Java Power Tools*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2008.

[21] Sonatype. *Maven, The Definitive Guide*. O'Reilly, Sebastopol, CA, USA, 2008.

[22] Grupo Taric. Taric. Website, 2010. http://www.taric.es/.