

Construcción de software: una mirada ágil

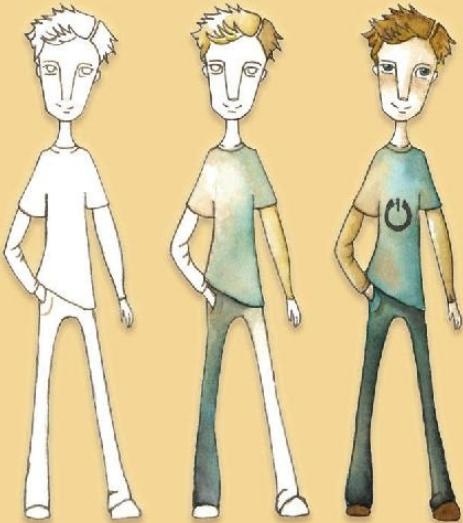
Nicolás Paez • Diego Fontdevila • Pablo Suárez
Carlos Fontela • Marcio Degiovannini • Alejandro Molinari



EDUNTREF

Construcción de software: una mirada ágil

Nicolás Paez • Diego Fontdevila • Pablo Suárez
Carlos Fontela • Marcio Degiovanni • Alejandro Molinari



EDUNTREF

Construcción de software: una mirada ágil

Nicolás Paez

Diego Fontdevila Pablo Suárez Carlos Fontela

Marcio Degiovannini

Alejandro Molinari

EDUNTREF

Editorial de la Universidad Nacional de Tres de Febrero

Coordinación editorial

Néstor Ferioli

Corrección

Licia López de Casenave

Ilustraciones

María Compalati

Directora de diseño editorial y gráfico

Marina Rainis

Diseño y diagramación

Ediciones Horizontales

Coordinación gráfica

Marcelo Tealdi

© Nicolás Paez... [et.al.]

© de esta edición UNTREF (Universidad Nacional de Tres de Febrero) para EDUNTREF (Editorial de la Universidad Nacional de Tres de Febrero). Reservados todos los derechos de esta edición para Eduntref (UNTREF), Mosconi 2736, Sáenz Peña, Provincia de Buenos Aires. www.untref.edu.ar

Directora editorial: María Inés Linares

Maquetación: Ediciones Horizontales

Primera edición septiembre de 2014.

Hecho el depósito que marca la ley 11.723.

Queda rigurosamente prohibida cualquier forma de reproducción total o parcial de esta obra sin el permiso escrito de los titulares de los derechos de explotación.

ISBN: 978-987-4151-00-1

Prólogo

—¿Y vos qué hacés, Juan? ¿Seguís trabajando con computadoras?

—Mmm... ahora trabajo más con personas que con computadoras.

Esta conversación la tuve una semana antes de escribir estas líneas, y me llama la atención el cambio personal y profesional que implica mi respuesta. Los que desarrollamos software y soluciones basadas en tecnología de la información nos dimos cuenta que los problemas que solucionamos son eminentemente humanos, y los modelos matemáticos a los que queremos llevarlos son simplificaciones tranquilizadoras para nosotros, nos llevan a nuestra zona de confort.

Y a esta complejidad, debe sumarse que la forma en que se resuelve el problema es una construcción social, en la que evolucionan simultáneamente el producto y el equipo que lo produce.

Esto se contrapone con el paradigma utilizado desde fines de la década de 1960, que ve al desarrollo de software como “una construcción hecha en fábricas de software por recursos”, equiparando diseño con construcción, a equipos con líneas de montaje y personas con máquinas.

Ese cambio paradigmático en el desarrollo de software, al que suele llamarse Desarrollo ágil, tiene impacto doble. Por un lado, nos lleva a tener un acercamiento más humanista y holístico. Por otro, nos exige profundizar nuestro conocimiento en las herramientas y prácticas. Los artesanos se dedican continuamente a aprender y mejorar su arte, los músicos a dominar sus instrumentos. Eso les permite crear e improvisar. Así también, a los desarrolladores de software a dominar nuestras herramientas y prácticas nos permite imaginar soluciones innovadoras y realizar cambios a bajo costo en productos existentes, en un proceso creativo grupal con el que generamos resultados asombrosos, personas felices y equipos altamente productivos.

Un punto de inflexión en los paradigmas es cuando estos empiezan a enseñarse como “la mejor manera actual” de realizar la práctica profesional. Los autores de este libro son todos profesionales de la industria y docentes universitarios que están llevando la experiencia en la práctica profesional a sus clases y materias, enseñando desde el nuevo paradigma.

Y se enfrentaron a una dificultad, como muchos de nosotros en esa situación. ¿Con qué material de soporte enseñamos? El mismo problema tienen los profesionales que quieren actualizar su conocimiento.

Hay pocos libros escritos originalmente en castellano sobre Desarrollo ágil, las traducciones sufren de las inconsistencias en los términos claves, al punto que para asegurarnos que hablamos de lo mismo, debemos recurrir al inglés.

Además, quizás por el origen del Desarrollo ágil (desde la industria), los libros existentes se enfocan en nichos del conocimiento y, por la rápida dinámica de creación del conocimiento, aún entre autores en inglés se generan diferencias. Por lo tanto, libros de pocos años ya están desactualizados en cuanto a terminología o incluso conceptos.

Este balance fue un gran desafío para los autores, unificar los conceptos mínimos, con contenido suficiente para que sea una explicación del tema en cuestión y no solo un glosario, y sin caer en la tentación de hacer una enciclopedia del conocimiento, que estaría desactualizada aún antes de finalizar su escritura.

Y si ese desafío era poco, los autores decidieron realizarlo en un grupo de seis personas, apostando a lograr consensos en criterios y estilos.

El resultado es un libro que ayudará a profesionales, docentes y alumnos a incorporar ideas, prácticas y herramientas de Desarrollo ágil de Software, ya que:

- Tiene los contenidos mínimos de todas las facetas del Desarrollo ágil de Software.
- Está organizado según las categorías del nuevo paradigma, cada una de ellas tratada con profundidad, estilo y terminología consistente (no una adaptación forzada de los conceptos nuevos en esquemas mentales previos).
- Contiene material pensado, escrito y usado en situaciones reales de enseñanza, tanto universitaria como profesional.
- Aporta numerosas referencias que permiten profundizar cuando el lector lo deseé.
- Logra riqueza conceptual y profundidad gracias al aporte de puntos de vista y experiencias distintas.

Creo que este libro es la continuación y aceleración de la adopción del Desarrollo ágil de Software, que empezamos de a poco entre 2001 y 2005 en distintas materias de la Facultad de Ingeniería de la Universidad de Buenos Aires, y que tuvo luego un salto cualitativo con el evento ágiles 2008 en Buenos Aires, que dio impulso a la comunidad latinoamericana. Actualmente se está enseñando o empezando a enseñar Desarrollo ágil de Software en universidades de toda América Latina, y este libro facilitará mucho este proceso.

Este libro ayudará al docente a preparar su materia; al alumno, a complementar lo que aprenda en clase; como profesional, a tener un libro de autoaprendizaje y referencia. Pero no es suficiente para aprender a hacer Desarrollo ágil de Software. Ningún libro lo será.

Gerald Weinberg comenta en su libro “Secrets of Consulting” que el impacto que tienen nuestras enseñanzas es inversamente proporcional al tamaño de la audiencia. Generamos un cambio mayor en una relación de mentor mentoreado, que en cada lector de un libro. Pero, por otro lado, al libro pueden acceder muchas personas más, solo podemos ser mentores de algunas pocas personas.

Entonces, tomá lo aprendido en este libro, y llevalo a la práctica, una práctica consciente, evaluando en cada momento qué te falta aprender. Busca mentores en tu universidad, en tu empresa, en la comunidad. Sumate a equipos que quieran ir más allá de su zona de confort.

Y recordá, cuando hayas avanzado en ese camino, que este libro es sólo la base sobre la cual innovar.

Juan Gabardini

Agradecimientos

A mi madre y a mi hermano, por estar siempre presentes; a Veily, por ayudarme a ver las cosas con otro matiz; a mis colegas de Snoop Consulting, Southworks y Kleer, por las ricas experiencias compartidas y a las comunidades de ágiles de Argentina y América Latina, por ese espacio de intercambio y experimentación del cual todos podemos ser parte.

Nicolás Paez

A mi esposa, por el apoyo. A mis hijos, por ayudarme a poner las cosas en perspectiva. A Carlos, por invitarme a participar de esto.

Pablo Suárez

A Fátima, mis hijos, mis padres y al resto de mi familia, por acompañarme en el camino que llevó a este libro; a mis socios y compañeros de Grupo Esfera, protagonistas de tantos proyectos compartidos, a ágiles, nuestra comunidad latinoamericana en la que tanto seguimos aprendiendo y creciendo; a Juan Gabardini, por abrirme la puerta a ella, y a los coautores de este libro, por compartir esta aventura.

Diego Fontdevila

A mi familia, por soportarme –en el sentido más amplio– en la escritura de este libro y de tantas otras iniciativas; a aquel alumno hoy desconocido que, allá por 1998, me preguntó por Extreme Programming, y me indujo a mis primeras lecturas sobre lo que terminaría siendo el agilismo.

Carlos Fontela

A Flor y a mi familia, porque siempre están ahí; a los coautores de este libro y en especial a Carlos, porque gracias a él entré en el mundo de la docencia y la escritura. A mis ex compañeros de C&S, con los que intentamos todo tipo de prácticas ágiles y otras locuras y a mis ex jefes, por la confianza que me tuvieron. A todo el equipo de Atix, con quienes aprendo cada día algo nuevo.

Marcio Degiovannini

A Gabriela, a mi familia y especialmente a mi madre, por acompañarme siempre. A mis compañeros de trabajo de C&S, con quienes ponemos en práctica y entusiasmo estos temas. A los alumnos y colegas de Taller de Desarrollo de Proyectos 2 de la FIUBA, de quienes aprendo en cada clase. A Carlos, por ser un referente y confiar en nosotros, y a los coautores por todo el esfuerzo, colaboración y conocimiento puesto en este proyecto.

Alejandro Molinari

A Juan Gabardini por su gran colaboración en la revisión del contenido.

A María Compalati por aceptar embarcarse como ilustradora de esta obra.

A la Universidad Nacional de Tres de Febrero por su apoyo para la publicación de esta obra.

Los autores

Prefacio

El por qué de este libro

Este libro es una introducción al desarrollo ágil, pasando por todos los temas importantes, y sin profundizar en ninguno. Los autores entendemos que, en el caso de que el lector interesado desee profundizar alguno, puede recurrir a otros libros más específicos o jornadas, cursos, blogs y otros recursos, en los cuales la comunidad ágil, tanto en América Latina como en España, es muy pródiga.

Otra cuestión que se da en paralelo es que mucha gente todavía considera que el desarrollo ágil no es serio, o se lo usa como atajo para no planificar, no documentar o no hacer nada que no sea programar. Por eso, nos proponemos desmitificar estas cuestiones y mostrar que el objetivo del desarrollo ágil es ser la mejor manera de realizar proyectos de desarrollo de software, de manera efectiva, en la inmensa mayoría de los casos.

Asimismo, el libro viene a cubrir una brecha en la literatura en lengua castellana. En efecto, hay excelentes libros sobre desarrollo ágil de software, desde hace ya más de quince años, pero en su mayor parte están escritos en inglés. Unos pocos se han traducido al castellano, pero incluso estos mismos están basados en experiencias en proyectos de desarrollo de software de países de habla inglesa, con una idiosincrasia distinta a la de los países latinoamericanos o a España.

Lo escrito está basado en la experiencia, profesional y académica, de seis autores con varios años de trabajo en el uso práctico de métodos ágiles. Todos ellos, sin excepción, han trabajado en organizaciones desarrollando software, son docentes universitarios y han dictado capacitaciones en forma particular.

Audiencia

El libro está dirigido a todas las personas que trabajan en desarrollo de software, pero que aún no han experimentado profundamente con el desarrollo ágil, tal vez porque no han sabido cómo hacerlo, tal vez por falta de conocimientos y hasta tal vez porque lo miran con cierta prevención.

Los gerentes en general pueden encontrar en este libro nuevas ideas sobre gestión, los comerciales pueden incorporar otra mirada sobre el tema, los clientes de equipos de desarrollo pueden usarlo para mejorar su interacción con los mismos, los analistas de negocio y funcionales van a hallar formas novedosas y muy efectivas para interactuar con usuarios y desarrolladores. Los testers se encontrarán con filosofías y técnicas de trabajo que le otorgan mucho valor a sus tareas, a la vez que les proponen nuevos puntos de vista. Los desarrolladores, en fin, podrán encontrar soluciones a muchas cuestiones que hoy soportan con resignación.

No es este un libro para expertos en métodos ágiles. Los que ya han experimentado abundantemente con el agilismo pueden encontrar consejos más precisos en la comunidad ágil, que en América Latina y en España está muy desarrollada. En particular, las Jornadas ágiles, que desde 2008 se vienen haciendo en distintas ciudades de América Latina, son un espacio muy rico para compartir experiencias, aprender de los éxitos y errores ajenos, y escuchar a oradores venidos de todo el mundo.

No obstante, quienes hayan incursionado en algunas prácticas ágiles y deseen continuar leyendo sobre sus fundamentos, complementación con otras prácticas y métodos, tienen en este libro un importante aliado para comprender cómo trabajar más efectivamente.

Estructura del libro

Los primeros dos capítulos son una introducción al tema del desarrollo de software ágil y construyen un puente entre las premisas básicas tradicionales de la comunidad de desarrollo de software y los pilares de la filosofía ágil. El resto de los capítulos están organizados de acuerdo a nuestra experiencia en proyectos reales de desarrollo de software, describiendo las actividades típicas más o menos en el orden en que cobran importancia en el proyecto. Estos capítulos, salvo el apéndice, son de dos tipos: o bien se enfocan en una técnica o área de práctica particular (por ejemplo, el capítulo Arquitectura y diseño en emergencia) o bien se enfocan en algún aspecto de la agilidad que nos parece central (por ejemplo, el capítulo “La fantasía de evitar los cambios”). El apéndice, como cierre, hace una recorrida por los métodos ágiles más establecidos. A lo largo del texto, los recuadros resaltan algún contenido o definición, y los que tienen las iniciales de un autor presentan una anécdota o comentario personal.

Traducciones

Muchos términos utilizados en este libro tuvieron su origen en inglés. En nuestro afán de hacer una obra en castellano hemos intentado proveer traducciones para todos ellos.

En los casos de términos con una traducción ampliamente establecida, la hemos utilizado. En los casos de términos con diversas traducciones o directamente si no la tienen, nos aventuramos en su gran mayoría a proponer la nuestra, indicando siempre el término original en una nota al pie.

Finalmente, en algunos pocos casos, hemos decidido mantener los términos originales por no encontrar traducciones apropiadas que no alteren la esencia del término (user story, backlog).

Qué viene después

Más allá de leer este y otros libros, y de experimentar en proyectos propios, creemos que para hacer desarrollo ágil es necesario aprender de otros, de los que tienen caminos parecidos y de los que ya han recorrido un largo trecho, de sus errores y también de sus aciertos. Para eso no hay un lugar mejor que la comunidad ágil, un espacio de pertenencia en el que practicantes de distintas extracciones comparten conocimientos y organizan eventos para aprender en conjunto. El sitio central de la comunidad ágil de América Latina es www.agiles.org, y el foro es foroagiles@yahooroups.com. Allí se pueden encontrar referencias a comunidades y eventos locales y regionales. Esperamos verlos.

¿Por qué cambiar?

Leyendas fundacionales

Casi todas las actividades económicas y las disciplinas científicas tienen leyendas fundacionales. Por ejemplo, dicen que la física moderna nació cuando a Newton, que dormía bajo un manzano, le cayó una fruta en la cabeza. Los primeros filósofos, según nos cuenta la tradición, compartían sus conocimientos en el ágora de Atenas, un lugar idílico donde discípulos y maestros podían debatir abiertamente sus preocupaciones.

Y así pasa con todas las ciencias y actividades humanas. Poco importa si son ciertas o no esas leyendas. Probablemente algo tengan de verdad, y otra parte haya sido inventada. Pero lo cierto es que influyen en nuestra percepción actual. Y el desarrollo de software no podía ser menos.

Los nostálgicos aseguran que hubo un tiempo en que los clientes nos decían que aplicación deseaban, nos dejaban trabajando por un largo tiempo, y cuando volvíamos con el producto desarrollado, se alegraban porque cumplía plenamente con sus expectativas. Es más, no sólo los clientes no necesitaban ver a los desarrolladores durante la ejecución del proyecto, sino que tampoco los desarrolladores necesitábamos comunicarnos con los clientes, ya que contábamos con una documentación tan exhaustiva y lograda sobre el producto a construir, que cualquier comunicación podía arruinar nuestro trabajo.

Se nos ha contado que en aquellos tiempos, los clientes desconocían que el software era un producto modificable, aunque ya algunos se estaban preguntando qué significaba el prefijo “soft” en la palabra en cuestión. Por otro lado, ese no era un problema, pues ¿quién querría modificar algo tan perfecto?

Los profesionales de desarrollo de software, debido a estas condiciones, podían planificar sus proyectos con seguridad, sabiendo que, una vez iniciado el proyecto, no había ningún peligro de que el plan cambiase. La tranquilidad que brindaba la previsión de todo futuro posible era muy reconfortante.

Además, existía –y existe– el dogma del mayor costo del cambio conforme el proyecto avanza, lo cual favorecía que cualquier modificación fuera vista como una amenaza. No es que el dogma sea absolutamente falso [Boehm1981], pero más que inducirnos a rechazar pedidos del cliente, debería animarnos a buscar formas más sencillas de implementarlos.

En realidad, ninguno de los autores de este libro conoció tiempos tan felices. Lo cierto es que, tanto si hubieran existido esos tiempos, como si se tratase de mitos de la historia del desarrollo de software, hoy vemos que sin duda las cosas no son así. Sin embargo, se trabajaba –y muchos todavía lo hacen– pensando que las cosas funcionaban de esa manera.

Errores de inmadurez

El desarrollo de software es una disciplina joven. Hace poco más de 60 años que se programan computadoras, hace 40 se empezó a intentar sistematizar un proceso, y hace unos 30 se empezaron a introducir paradigmas que permitieran el manejo de la complejidad. Si comparamos con las ingenierías tradicionales, que en general tienen varios siglos de antigüedad, no deberíamos asombrarnos de que la nuestra sea considerada una actividad humana inmadura.

Prácticamente no existe ciencia que se haya aplicado en sus inicios sin cometer grandes errores. Los médicos de la antigüedad pretendían curar a sus enfermos con purgas y sangrías, que en ocasiones aceleraban la llegada de la muerte.

Algunos errores no han sido tan graves, sino que solo fueron simplificaciones fácilmente refutables. Pero aun estas simplificaciones, muchas veces han puesto en entredicho la disciplina en sus momentos fundacionales. La economía, dicen los especialistas, nació con el libro de Adam Smith, que explica su funcionamiento en un mundo ideal de competencia perfecta. Cuando se vio que esa simplificación era excesiva, muchos pretendieron predicar la invalidez de toda la ciencia económica, por lo que fue necesario investigar en el marco de hipótesis más realistas para devolverle el prestigio que se estaba esfumando.

Algo parecido pasó con la astronomía, al punto que el sistema de universo definido por Ptolomeo dos siglos antes de Cristo, seguía siendo utilizado en el siglo XV, hasta que Copérnico lo puso en entredicho, provocando gran contrariedad en el mundo académico de la época, que no estaba dispuesto a que se modificasen las nociones esenciales de su ciencia.

Con el desarrollo de software también han pasado estas cosas. Se han cometido gruesos errores iniciales que han provocado el fracaso de muchos proyectos y el abandono de iniciativas de informatización. Más tarde, se han establecido algunos fundamentos basados en supuestos simplistas, o simplemente erróneos, que también han hecho perder prestigio a la disciplina. Además, el ambiente académico es reacio a pensar en términos nuevos, a pesar de la corta historia del software.

Por ejemplo:

- La idea de la división del trabajo en roles, basada en una visión tayloriana [Aitken 1960], llevó a que se crearan varias especialidades, y que cada una tuviera una labor totalmente diferenciada. Un analista funcional jamás se comparaba con un programador (¡Dios nos libre!), dado que estos últimos eran el equivalente de los obreros industriales en la producción de software, gente a la que no se le pedía pensar, sino solo trabajar. Para pensar estaban los analistas, que sabían lo que querían los clientes, cuales eran sus necesidades y como lograr satisfacerlas desde lo técnico. En definitiva, cada persona, según su rol, buscaba un objetivo diferente. A nadie se le ocurría pensar que en realidad se necesitaba un equipo multidisciplinario trabajando en pos de entregar valor al cliente.

- También se dio importancia excesiva a los documentos. Partiendo de la idea, tal vez razonable, de dejar registro de las decisiones que se toman en un proyecto mediante documentos, y siguiendo con la noción de que cada perfil se debía ceñir a sus tareas específicas, se llegó al extremo de que la única forma de comunicación en los proyectos eran los documentos. Los analistas especificaban requerimientos, que era lo que podían ver los diseñadores, programadores y testers. Al fin y al cabo, ¿qué otra cosa debían conocer los programadores y testers de su proyecto? Los diseñadores también especificaban el diseño con documentos, probablemente acompañados de complejos y detallados diagramas. Si esos diagramas luego debían ser mantenidos conforme cambiaba el software, no era un problema que se tuviese en cuenta.

- Los diseñadores se podían lucir con complicados diseños que mostraran su genialidad. Los programadores, toda vez que podían, introducían código que nadie sino ellos podrían entender. Y eso los hacía felices a todos, sin preguntarse si era bueno para el trabajo final.

Porque el producto final también había pasado a ser secundario. Lo único importante eran los documentos, que definían qué se iba a desarrollar y cómo se iba a construir la aplicación. El software en sí mismo no era lo que importaba, y hasta se fantaseaba con la

generación automática de código para enfatizar esta idea, pensando en un futuro feliz en el que la codificación humana fuera innecesaria.

Pero llegó el día en que todo eso ya no se pudo sostener más.

La evidencia nos condena

Todos conocemos casos de proyectos de software que han fracasado. Ha habido problemas en pequeños proyectos, más o menos intrascendentes (salvo para quienes los han sufrido), y hay casos de grandes fracasos, de esos que mencionan los libros. A veces los particulares parecen ser solo eso, y los desecharmos pensando que son una pequeña proporción de los proyectos.

De Marco y Lister mostraban¹, hace ya 25 años, la gran proporción de proyectos de software que fracaaban. Tal vez pensemos que es un dato anacrónico.

Ya no es tiempo de seguir equivocándonos

En los tiempos que corren, los clientes quieren todo lo antes posible. Ya no se puede, como en otros períodos, decirle a un cliente que vamos a trabajar durante un año y le mostraremos todo cuando esté terminado. Ellos quieren ver el software funcionando, lo antes posible, hacer observaciones sobre el mismo, saber cómo vamos en el proyecto y cuánto falta para el feliz día en que determinada funcionalidad esté lista.

Como además los clientes se han dado cuenta de que el software es maleable, que admite cambios aún durante su desarrollo, están constantemente buscando mejorar el producto. Si bien en un punto eso puede irritarnos, es parte ineludible de la naturaleza del software y, por lo tanto, debería serlo también de nuestro trabajo.

Precisamente, la necesidad de realizar cambios en los proyectos lleva a que los planes no puedan ser rígidos. Hay que entender que la planificación debe ser algo vivo, algo que se puede adaptar en tiempos, en alcance, e incluso por cuestiones derivadas de la incertidumbre sobre el diseño de la solución.

Otro inconveniente es el costo del cambio, que no por ser posible es barato. Adicionalmente, ese costo empeora si el diseño es complicado y el código poco legible.

Además, el software que desarrollamos hoy es tremadamente más complejo que el de décadas atrás. Esa complejidad no se puede encarar con procesos de desarrollo que solo funcionan razonablemente bien con pequeños programas. Lo mismo aplica a la calidad: un producto complejo necesita ser construido con la calidad en mente. Esta no puede ser algo que agregamos al producto a posteriori.

Por otro lado, los roles con objetivos contrapuestos pueden hacer perder el objetivo principal, que es entregar valor al cliente. Este valor no puede estar en los documentos, que son artefactos que nos sirven a los desarrolladores durante la construcción. ¿O alguno de nosotros, cuando quiere que le construyan una casa, está dispuesto a recibir planos a cambio?

Los clientes exigentes nos obligan a que tengamos criterios de aceptación, sin ambigüedad, que nos indiquen cuando hemos cumplido lo que ellos requieren y cuando podemos dar por concluido el desarrollo de una funcionalidad. Y esos criterios nos van a servir también a nosotros para medir el avance, aun cuando los clientes no sean tan exigentes en ese punto.

Como consecuencia de todo lo dicho, el tiempo en que nuestro optimismo nos hacía pensar que los errores y descuidos podían ser tolerados, o incluso no ser percibidos, ha quedado atrás. Y eso exige pensar en otras maneras de desarrollar.

Buscando encauzar al desarrollo de software

Siguiendo con la analogía de las demás ciencias y técnicas, no siempre es fácil romper con lo establecido para cambiar métodos y procedimientos. Hace falta ver los errores, luego evidenciarlos y finalmente proponer cambios mostrando sus ventajas. Muchos pretenderían mantener sus anteojeras, o a lo sumo realizar pequeños retoques a las visiones del pasado. No les fue fácil a Heisenberg y a Einstein poner en entredicho a la física clásica con proposiciones tan poco intuitivas que parecían descabelladas. Algo parecido debe haberle ocurrido a Lord Keynes cuando se atrevió a proponer cambios en la lógica de la macroeconomía. O a los médicos medievales que, de a poco, introducían cambios aprendidos en la España islámica.

Pero parece obvio que si hay problemas y se están cometiendo errores, habría que tratar de enfrentarlos y corregirlos. En el desarrollo de software seguimos purgando pacientes hasta matarlos por deshidratación. Esto no es serio en una disciplina que, si bien es joven, ya cumplió 60 años. Por eso es que a fines del siglo pasado se empezaron a sugerir las mejoras que llevaron a los métodos ágiles.

En efecto, resulta totalmente insólito plantear que los proyectos de software se alarguen por la necesidad de generar documentación innecesaria, que cuando un cliente pida un cambio estemos con la guardia en alto para que no arruine nuestro diseño, que los equipos sean en realidad compartimentos estancos de perfiles que solo se comunican mediante documentos, que los clientes deban conformarse con ver documentos de requerimientos y de diseño en vez de software funcionando, que los planes estén tallados en piedra.

Y si a fines del siglo pasado no podíamos seguir pretendiendo todo eso, mucho menos una vez que surgieron personas que, como hicieran en otros tiempos y desde otras disciplinas Copérnico, Heisenberg, Einstein, Colón o Keynes, han puesto en entredicho unas cuantas verdades instaladas y nos han demostrado que el desarrollo de software puede ser más eficiente, más cooperativo, más transparente y menos rígido de lo que había sido hasta ese momento. Hoy, un poco más de una década más tarde, es todavía más inverosímil que haya gente que no perciba que los principios en los que nos habíamos basado tenían serios errores de concepción, que hacían que los métodos derivados de los mismos fueran inadecuados en determinados contextos.

La naturaleza del software al rescate

Ahora bien, varias de las respuestas a los problemas planteados están en el propio software y en las peculiaridades de los proyectos de desarrollo del mismo.

Expliquémonos:

- A menudo se dice que el software es maleable, es decir, que se puede adaptar durante su construcción, y aun una vez terminado. ¿Qué mejor, entonces, que usar esa maleabilidad para poder ofrecer alternativas a nuestros clientes? ¿Por qué resistirnos tanto a los cambios que sabemos posibles?
- Por otro lado, el software es particionable y, por lo tanto, se puede construir por etapas, de modo tal que cada una implique que a la salida de la misma tengamos un producto, parcial, pero con valor para el cliente. ¿Por qué no usar esta característica para darle visibilidad durante su construcción?
- Otra cuestión a la cual se le presta poca importancia es que el desarrollo de software implica la materialización de conocimiento en

programas de computadora. ¿Por qué no usar las reuniones de captura de conocimiento para interactuar más con nuestros clientes y dentro del mismo equipo? ¿Por qué no aprovechar para socializar?

- Contrariamente a lo que sucede en otras disciplinas, el diseño y la construcción del producto no son procesos separados, uno único y el otro repetitivo, sino que se hacen en conjunto, con un único producto para cada diseño. Entonces, ¿por qué nos resistimos de manera tan apasionada a los cambios de diseño durante el desarrollo?

- El software es también susceptible de ser copiado en forma íntegra. ¿Por qué no aprovechamos esta característica para entregar productos de calidad?

- Si decimos que el software es extensible, ¿por qué nos resistimos a hablar de la evolución permanente de cada producto?

Es cierto que, otras veces, el propio software conspira para solucionar los problemas, como ocurre con la visibilidad del producto. En efecto:

- Al ser un producto invisible por definición, es complicado saber, durante el desarrollo, cuanto se ha construido y cuanto queda por construir. Por esto se ha vuelto necesario definir técnicas y métricas específicas para el software. Esta falta de visibilidad del producto también hace difícil que el cliente sepa rápidamente si lo que construimos es lo que él esperaba. Por eso es que conviene definir criterios de aceptación, que también permiten que el propio equipo de desarrollo pueda conocer cuando puede dar por terminada la construcción de determinada funcionalidad.

- Otro problema del software es su complejidad, muchas veces mencionada, pero poco comprendida cabalmente. En efecto, se habla con ligereza de sistemas medianos de, digamos cincuenta mil clases. ¿Somos conscientes de que no estamos hablando del equivalente de una máquina de cincuenta mil piezas, sino de una con cincuenta mil tipos de piezas distintas?

El resto lo cubren las personas

Otro aspecto característico del software es que es un producto construido en su totalidad por personas. Por eso es que la mayor parte de los problemas y de las soluciones debemos verlas más desde la sociología que desde la tecnología. Esto puede ser duro para la mayoría de nosotros, e incluso para quienes se desempeñan en roles gerenciales, que hemos sido formados más en tecnología que en disciplinas humanísticas. Sin embargo, deberíamos pensar que las habilidades sociológicas, aunque carezcamos de nociones formales, están en nuestro ADN y que las venimos practicando desde que nacimos. Por ejemplo:

- Durante miles de años, hemos evolucionado comunicándonos cara a cara para poder entendernos y superar los malos entendidos, los equívocos, y todo lo que solo podemos expresar mirando a nuestro interlocutor. Por eso es importante no olvidar que esa es la manera más natural de comunicación entre humanos.

- Nuestra naturaleza social también hace que seamos propensos a formar equipos exitosos, que se sienten orgullosos de su creatividad y sus éxitos y, sobre todo, si los problemas resueltos fueron muy complejos. Estos equipos tienden a autoorganizarse, sin necesidad de impulsos externos, y a trabajar mejor cuanta más sinergia logren.

- Por último, contrariamente a lo que dicen algunos refranes, aprendemos de nuestros errores, y somos los únicos animales capaces de reflexionar sobre los mismos, para no volverlos a cometer. Y como contrapartida, tendemos a analizar y a repetir conductas que hayan llevado a resultados exitosos.

Ahora bien, dificultades también hay. Aunque en este caso las dificultades no suelen venir de las personas que trabajan, sino de prejuicios típicos del mundo corporativo, que atentan contra la naturaleza humana:

- Una de las nociones que más conspira contra la autoestima de las personas, y que afecta la calidad de lo producido por esta misma razón, es el considerar a las personas como “recursos”, pretendiendo que, como ocurre con otros recursos que afectan nuestro proyecto, se trata de piezas intercambiables. Eso lo pueden provocar tanto los gerentes de mentalidad industrial, como las metodologías que ponen el foco en que no importan tanto las personas como el seguimiento de un método supuestamente infalible.

- Asimismo, cuando por cuestiones de costos o cronograma, se impulsa a las personas a construir un producto de baja calidad, también se afecta la autoestima de todo el equipo de trabajo. En realidad, todo cronograma ajustado de manera irreal es percibido como una falta de respeto por los equipos experimentados. Si bien en algunas ocasiones se le puede pedir un compromiso a un equipo ante una necesidad puntual, esto funciona en la medida en que el equipo haga suyo este compromiso, y que realmente sea una excepción definida y acordada, no la regla. Las presiones ejercidas sobre las personas para que produzcan más no funcionan, simplemente porque el hecho de que se pueda obligar a alguien a estar más tiempo sentado en una silla, no implica que de esa manera se logre creatividad o productividad.

- Otro aspecto negativo es un entorno laboral que conspire contra la producción. Sillas y mesas incómodas, falta de luz, infraestructura inadecuada, no contar con el software necesario para desarrollar, restricciones de acceso a Internet que impiden investigar, son todos problemas que acarrean frustración y cuestan bastante más de lo que parece. Por eso, un buen gerente debe enfocarse en permitir que las personas trabajen cómodas y eficientemente más que en “hacer que trabajen”.

- Y por último, hay que confiar en las personas y en los equipos que las personas conforman. Si un gerente vive obsesionado por desarmar equipos que disfrutan del trabajo en conjunto, por temor a que formen grupos elitistas, terminará con trabajadores poco motivados y menos productivos. Si solo considera que están trabajando cuando los ven escribiendo código, y no cuando piensan, conversan con sus colegas, investigan soluciones en la web o se reúnen espontáneamente, solo va a lograr una actitud defensiva y poco productiva.

El manifiesto ágil

Así como Copérnico cuestionó los principios de la astronomía de su tiempo, o como Colón impugnó la teoría de la Tierra plana, hubo un conjunto de personas que se dieron cuenta de que las premisas en las que se basaba el desarrollo de software tenían fallas, y propusieron una serie de ideas para remediarlas.

Los cuestionadores de las premisas antiguas del desarrollo de software fueron un conjunto de 17 críticos² de los procesos tradicionales, que se reunieron para redactar lo que denominaron el “manifiesto ágil” en febrero de 2001³.

El manifiesto dice⁴:

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

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Respuesta ante el cambio sobre seguir un plan.

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

Los últimos 10 años

Ahora bien, si el manifiesto ágil tiene ya más de 10 años, ¿qué pasó desde entonces? ¿Hubo una adopción importante de los principios ágiles? ¿Hubo una mejora en los proyectos de desarrollo de software? ¿Se puede ver una correlación entre la mayor adopción de los métodos ágiles y la mejora en los proyectos? La encuesta [VersionOne2012], realizada a fines de 2012, parece indicar que las tres preguntas se pueden responder en forma positiva.

Respecto de la adopción, la encuesta nos muestra una mayor proporción de organizaciones usando métodos ágiles, a la vez que una mayor proporción de los proyectos en cada organización⁵. Incluso ha aumentado la adopción de métodos ágiles en equipos distribuidos.

Las otras dos preguntas no se encuentran respondidas en forma directa por la encuesta, pero los encuestados declaran haber obtenido mejoras gracias a su adopción del desarrollo ágil. Un 90% dice haber mejorado su habilidad para manejar cambios de prioridades. Otras ventajas observadas han sido bajar el tiempo en salir al mercado y la mejor alineación con el negocio.

La encuesta se sale del molde de aquellas que buscan mostrar mejoras en tiempos, costos, o apego a los requerimientos del cliente. En efecto, el movimiento ágil no considera al alcance de un proyecto como algo fijo, y si los tiempos o los costos mejoran, esto es más bien un efecto secundario. Para cualquier agilista, la calidad del producto, la visibilidad del proceso, la facilidad de adecuación a los cambios en las necesidades del cliente y el menor tiempo de retorno de la inversión son más importantes que cumplir con unos requisitos fijos y establecidos en el momento del lanzamiento del proyecto.

En resumen

Si las cosas no están funcionando bien, ¿Por qué no cambiar? ¿Por qué fingir que el desarrollo de software debe ser lo que no es? Al fin y al cabo, como vimos, la propia naturaleza del software y la actividad inherentemente humana que permite construirlo, nos ayuda en nuestra tarea de trabajar mejor. Entonces, ¿qué nos impide cambiar? ¿El miedo a lo desconocido? ¿No somos parte, acaso, de una disciplina joven, que no debería tenerle miedo al cambio?

Insistir en los errores, en el “no puede ser”, nos lleva al conservadurismo de Poincaré que, por aferrarse a antiguas concepciones físicas, no se atrevió a dar el salto que luego Einstein postularía: por eso hoy nos acordamos mucho más del segundo que del primero.

En eso estamos, pues. Avancemos.

Iterativo por naturaleza

En los últimos cuarenta años, no ha habido ningún modelo de proceso de desarrollo de software exitoso que no sea iterativo. Desde el proceso en espiral hasta el unificado, todos los procesos recomendados tanto desde la academia como la industria asumen una estructura de aproximaciones sucesivas para su desarrollo.

Sin embargo, como vimos en el capítulo anterior, son comunes múltiples visiones (el modelo en cascada) y prácticas (la planificación de largo plazo basada en tareas) que asumen un proceso en el que, en mayor o menor medida, se sabe todo lo importante al principio.

Muchas veces hemos escuchado enunciados con la clásica estructura de recomendación seguida de claudicación:

“Hay que hacer las pruebas en paralelo con el diseño, pero no podemos enseñarlo así porque es confuso”; o la otra de:

“Claro que el análisis puede ser influenciado por el diseño, pero de todas maneras siempre hay que hacer todo lo que pide el cliente, por eso se llaman requerimientos”.

En estos ejemplos, la vocación parece clara pero falta el coraje o la percepción profunda o la técnica para llevarla a buen término. En cada uno, se deja para después algo que debería hacerse antes y, por lo tanto, se asume en lugar de aprender o discutir. En fin, así nos va.

Parte del éxito de los métodos ágiles está en abrazar esa perspectiva de persona mirando al abismo, y proponer técnicas específicas para lidiar con la complejidad.

Ejemplos concretos son:

- El time-boxing, consistente en limitar el tiempo a utilizar antes que el alcance de las tareas, forzando un límite que podemos controlar siempre (el tiempo transcurrido) y usándolo como medida para controlar otros aspectos (alcance, avance, calidad, valor entregado, etc.).
- El desarrollo guiado por pruebas (Test Driven Development o TDD), consistente en escribir las pruebas antes de escribir el código y hacerlo iterativamente de manera tal que las mismas puedan ser ejecutadas una y otra vez para garantizar que el código evoluciona correctamente.
- Las prácticas de planificación estratégica y táctica⁶ que se ejecutan iterativamente durante el proyecto.

En este capítulo nos proponemos revisar las razones por las cuales creemos que los procesos iterativos y por incrementos son la evolución natural del desarrollo de software.

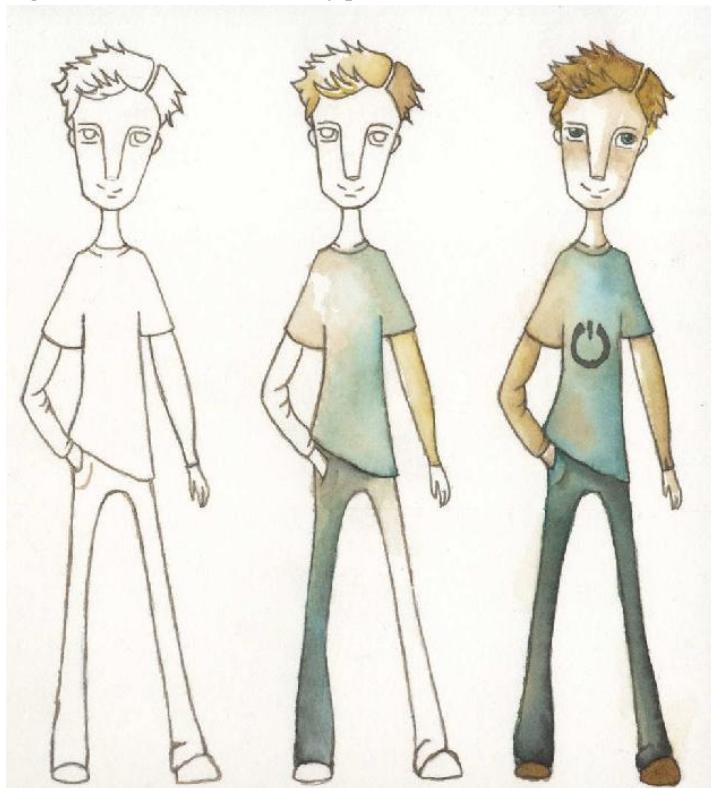
El desarrollo de software como proceso iterativo

La principal confusión reinante alrededor del proceso de desarrollo consiste, como vimos en el capítulo anterior, en una mirada basada en actividades disjuntas (análisis, diseño, implementación, pruebas, etc.) que son llevadas a cabo por gente que se comunica poco y mal, principalmente a través de documentos. Esta confusión hace fácil identificar implícitamente el proceso con el modelo en cascada, es decir, con una secuencia de actividades (por ejemplo, primero implementación y después pruebas). Aún cuando no sea explícita, esta preconcepción tiende a dificultar la aplicación de un proceso iterativo.

Ahora bien ¿Qué es un proceso iterativo? Es un proceso de aproximaciones sucesivas al resultado final, que nos permite ir ajustando tanto el producto como el proceso para maximizar el valor del resultado. Un ejemplo es la escritura de este libro: produjimos varios borradores hasta llegar al resultado final⁷. En concreto, es un proceso donde las actividades se repiten en cada iteración, permitiendo obtener y analizar los resultados de esas múltiples actividades y utilizarlos para nutrir a las demás (por ejemplo, las pruebas se ejecutan en cada iteración sobre lo construido).

Lo anterior no quiere decir que todas las actividades tengan que realizarse en todas las iteraciones, el foco puede variar a medida que transcurre el proyecto, por ejemplo estar en un momento en la puesta en producción y en otro en la construcción.

Figura 2.1. Una obra iterativa y por incrementos.



Un proceso iterativo nos ayuda a:

- *Innovar*: podemos usar una o varias iteraciones para explorar alternativas. Si las alternativas son descartadas, el proceso nos permite limitar el esfuerzo dedicado a ellas mediante la asignación de un número acotado de iteraciones a esa exploración. Si las alternativas son seleccionadas, pueden refinarse en futuras iteraciones.

- *Minimizar el costo de nuestros errores*: como cada iteración aborda solo una parte del problema completo, si nos equivocamos, no estamos arrriesgando todo nuestro proyecto.

- *Maximizar las oportunidades de mejora*: un proceso iterativo provee múltiples oportunidades concretas (por ejemplo, retrospectiva⁸ al final de cada iteración) para la reflexión tendiente a la mejora.

- *Imprimir un ritmo*: con iteraciones de duración fija, los equipos realizan sus actividades a intervalos regulares, facilitando la asimilación de las prácticas (por su repetición disciplinada). El ritmo también fomenta el mantener un nivel de esfuerzo sustentable y evita el *burnout*⁹.

- *Maximizar las oportunidades de control*: al final de cada iteración podemos evaluar métricas y validar el producto para determinar el progreso y la alineación en relación a los objetivos del proyecto.

Un proceso iterativo por incrementos

Para ser efectivo, un proceso iterativo depende de que cada iteración produzca resultados concretos que nos den sensación de avance. Sin esa sensación es difícil revisar la planificación y garantizar que el proyecto en su conjunto vaya en la dirección correcta (es decir, que vaya a cumplir con sus objetivos).

Para ayudar en esa tarea, es común dividir el producto en incrementos, particiones que pueden desarrollarse en distintas iteraciones y permiten planificar el trabajo para cumplir con el alcance completo mediante un conjunto de iteraciones. En este modelo, cada iteración produce un nuevo incremento o refina uno anterior, de manera tal que al final todos los incrementos han sido construidos mediante sucesivos refinamientos.

No estamos solos

Este modelo de proceso iterativo y por incrementos no es una característica exclusiva del desarrollo de software. Como muestran Rob Austin y Lee Devin en su libro *Artful Making* [Austin 2003], tanto en el teatro como en otras formas de arte, en el diseño de estrategia, y otros trabajos creativos, los procesos exitosos son iterativos por la naturaleza del trabajo. Los autores describen las siguientes condiciones de aplicabilidad de esta forma de trabajo:

- *Necesidad de innovación*: cuando el producto es original o por lo menos lo es en el contexto actual.

- *Repetición confiable*: el proceso de trabajo es repetible, es decir, contamos con la capacidad y disciplina para trabajar iterativamente.

- *Bajo costo de iteración*: el costo de iteración se define como la suma de:

- Costo de reconfiguración: el costo de modificar lo que hemos realizado en iteraciones anteriores, o de cambiar el proceso.
- Costo de exploración: el costo de explorar alternativas que no son incluidas finalmente en el producto final, porque se encuentra una alternativa mejor o simplemente porque no forman un todo armónico con el resto.

Estas condiciones son típicas de situaciones en las que el producto que se obtiene es intangible (por lo menos parcialmente), y requiere el trabajo colaborativo de un grupo de personas.

El modelo de *Artful Making* nos permite caracterizar a nuestros procesos de desarrollo para decidir si corresponde o no un proceso iterativo. La filosofía ágil y muchos otros conciben al desarrollo de software como un proceso eminentemente innovador. Por innovador entendemos:

- Que los problemas a los que nos enfrentamos son radicalmente nuevos cada vez. Aunque parezca que muchos proyectos tienen cosas en común, lo que tienen de particular e interesante tiende a ser siempre más de lo esperado.
- Que la forma de trabajo apropiada para lidiar con esos problemas no puede definirse en detalle de antemano. Dicho de otra forma, que nuestros procesos y prácticas de trabajo deben adaptarse a la realidad específica de cada proyecto.

Si no hubiera necesidad de innovación, bastaría reusar software existente. Lo que ocurre con el reuso es que normalmente los requerimientos suenan a “quiero algo parecido pero distinto”. Las expectativas sobre el reuso tienden a ser altas, ya que implica no tener que desarrollar software, pero en la mayoría de los casos requiere extensión y adaptación. Además, el reuso en sí implica desafíos análogos o mayores en complejidad a los del desarrollo de software a medida¹⁰.

En cuanto a la repetición confiable, es uno de los desafíos metodológicos fundamentales de la ingeniería de software, como tema fundamental de la madurez que discutimos en el capítulo anterior. En ese aspecto, los métodos ágiles promueven una práctica y disciplina cotidiana (por ejemplo, reuniones diarias, validación y revisiones periódicas a intervalos regulares, etc.) que soportan tanto la repetición sustentable como la mejora continua.

Finalmente, dada la propia naturaleza intangible del software (como opuesto al hardware, que tiene costos de reconfiguración mayores), y de la mano de ciertas prácticas específicas (por ejemplo, integración continua), en el desarrollo es posible mantener bajo el costo de iteración.

La mejora como un proceso empírico

Dadas las características esenciales del software (complejo, intangible, ajustado al uso y cambiante [Brooks 1975]), y entendiéndolo como información empaquetada, el proceso de desarrollo es un proceso de aprendizaje continuo, tanto sobre los requerimientos y el contexto del sistema, como sobre el diseño y el proceso de desarrollo. Desde esa perspectiva, todo proceso de desarrollo debe implicar la mejora continua para garantizar mínimamente que se logren los resultados esperados, porque si no aprendemos lo suficiente sobre el contexto y el proceso, es poco probable que logremos pasar la prueba final de que el software sirva para lo que lo construimos.

Dicho de otra forma, si no nos esforzamos por mejorar, es muy poco probable que acertemos desde el principio o, como vimos en la sección anterior, que sepamos lo suficiente como para lograr nuestros objetivos.

La filosofía ágil asume esta mejora como un proceso empírico, es decir, basado en la exploración y la experimentación. En un equipo ágil, todos los individuos son solidariamente responsables por experimentar, evaluar y adaptar el proceso y las prácticas de desarrollo en forma iterativa, para lograr los objetivos del proyecto.

Los métodos ágiles promueven la mejora mediante algunas prácticas propias de un proceso iterativo:

- *Retrospectivas*: al final de cada iteración se evalúa el proceso y se establece un compromiso de mejora.
- *Revisiones*: al final de cada iteración se evalúa el producto y se determinan los refinamientos apropiados.
- *Incrementos*: el producto se realiza en partes pequeñas que pueden ser validadas tempranamente, reduciendo el impacto de los errores.
- *Planificación estratégica*: se revisan las prioridades, avance y resultados del proyecto. Basada en hitos como iteraciones completadas y entregas de subconjuntos de funcionalidad.
- *Planificación táctica*: se organizan las tareas de la iteración inmediata subsiguiente.

En resumen

Lidiar con problemas complejos requiere abordarlos progresivamente, tanto para aprender sobre el problema como sobre la solución. Enfrentar con éxito una gran incertidumbre requiere la sabiduría de dividir el problema en partes pequeñas, y la humildad para enfrentarlas como si cada una fuera tan importante como el todo. La filosofía y los métodos ágiles proponen prácticas específicas para mejorar la efectividad del proceso iterativo. Depende de nosotros aplicarlas con criterio para maximizar los resultados. En los próximos capítulos esperamos poder ayudar a lograrlo.

¹ Es famoso el primer capítulo de la primera edición de *Peopleware*, cuyo título es “Somewhere Today, A Project Is Failing” (en castellano: “Hoy, en algún lugar, un proyecto está fracasando”).

² Se trata de Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas. Todos venían del mundo industrial y no académico y, en forma separada, estaban trabajando en propuestas para remediar esta problemática desarrollando métodos propios.

³ Mientras estaba explotando la burbuja de Internet.

⁴ Tomamos la traducción oficial al castellano, que se encuentra en <http://agilemanifesto.org/iso/es/>. El manifiesto original en inglés se encuentra en <http://agilemanifesto.org/>

⁵ Tengamos en cuenta, no obstante, que el 77% de los encuestados son empresas europeas o estadounidenses, lo que no dice mucho de la adopción en los países de habla castellana.

⁶ Véase el capítulo “Planificación constante”.

⁷ En este momento, estamos haciendo un refinamiento de este capítulo.

⁸ Véase el capítulo “En retrospectiva”.

⁹ Entendemos por sustentable un nivel de esfuerzo que puede mantenerse durante todo el proyecto, como opuesto a los esfuerzos excesivos como trabajar cuarenta horas extra por semana, que si se mantienen en el tiempo causan serios problemas, incluyendo la rotación de personal (véase [DeMarco 1987], capítulo 3).

¹⁰ Watts Humphrey, al diseñar las métricas del PSP (Personal Software Process), propuso contar el tamaño de un sistema como lo escrito + lo reusado [Humphrey 2005b]. Véase [Garlan 1995] para una discusión de las dificultades en reuso de componentes de software.

Delineando el alcance

Todo proyecto, independientemente de la disciplina a la que pertenezca, comienza por una visión. La misma constituye el punto de partida y la motivación para realizar el proyecto. Puede que dicha visión esté formalmente documentada o no, pero más allá de esto es fundamental que todos los integrantes del equipo la conozcan, pues es el instrumento que debería guiarnos a la hora de tomar decisiones. Cuando vamos al ámbito de los métodos ágiles, estas afirmaciones son igual de válidas, pero aquí lo distintivo es la estrategia para convertir esa visión en un entregable de valor para el cliente. El primer paso para transformar la visión en producto es la definición del alcance. Este es el foco del presente capítulo.

Alcance, análisis y planificación

La definición de alcance es parte de la actividad de análisis del proyecto. En este proceso de análisis lo fundamental es entender la necesidad del cliente y su negocio. Al mismo tiempo, en ciertas ocasiones el alcance de un proyecto puede ser variable y, en ese caso, la definición de alcance se ve naturalmente “mezclada” con planificación. Es por esto que algunas de las técnicas tratadas en este capítulo serán también referenciadas en el capítulo “Planificación constante”.

Backlog de producto

El primer paso en el proceso de materialización de la visión es la definición del alcance. En términos ágiles, está representado por el backlog de producto¹¹. Este backlog de producto es una lista de ítems que representa todo el trabajo necesario para concretar la visión. En términos tradicionales de gestión, el backlog podría ser análogo a una WBS¹² orientada a producto. Sin embargo, hay una diferencia entre las WBS de los enfoques tradicionales, generalmente compuestas por todas las tareas a realizar, y el backlog del producto de los métodos ágiles, el cual debiera contener solo ítems que tengan valor real para el cliente.

Dependiendo del método particular que se utilice, estos ítems podrán tomar distinta forma, pero como ya hemos mencionado, en todos los casos deberán ser los que el cliente valore.

Algo fundamental para los métodos ágiles es que el cliente priorice los ítems de backlog. Luego de un trabajo de análisis, dichos ítems deberán ser estimados por el equipo de desarrollo. De esta forma tendremos la información necesaria para planificar la construcción de nuestro producto.

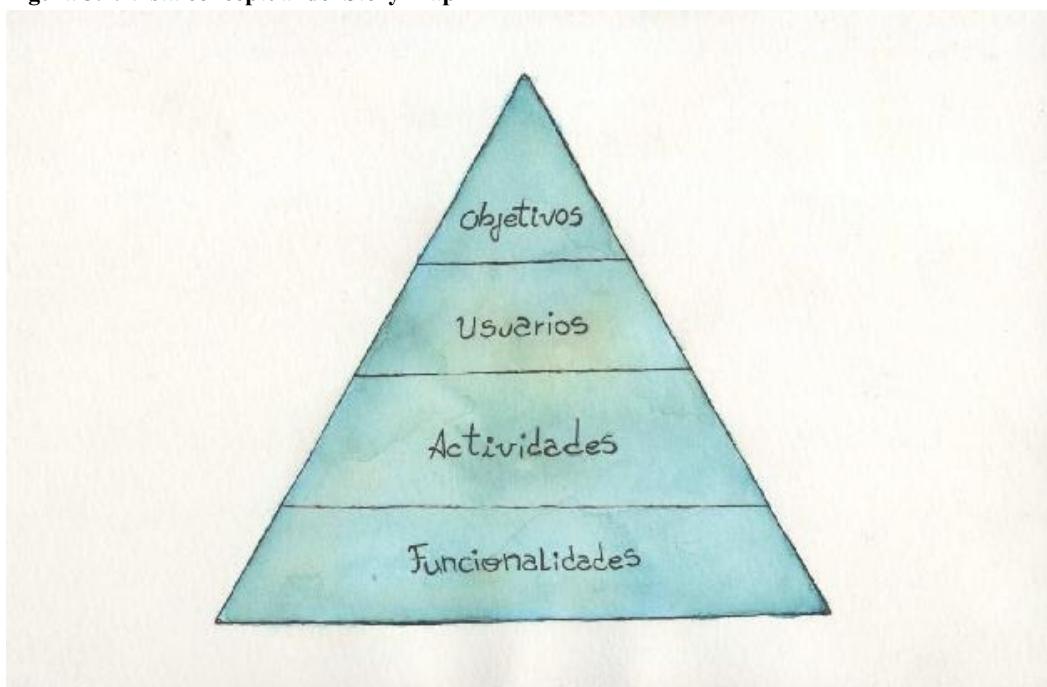
Para generar el backlog de producto a partir de la visión, existe una técnica muy popular en los ambientes ágiles denominada Visual Story Mapping.

Visual story Mapping

Esta técnica de análisis, propuesta por Jeff Patton [Patton 2005], parte de la idea de que a nivel organizacional trabajamos en un contexto de negocio con determinados objetivos que requieren de ciertos procesos de negocio, que son llevados a cabo por personas que realizan ciertas actividades. Para la ejecución de estas actividades, se realizan ciertas tareas utilizando herramientas y es ahí donde entra en juego nuestro producto/software. A partir de esto tenemos una jerarquía: proceso de negocio (objetivo) > actividades > funcionalidades de nuestro software.

Esta técnica se aplica en sesiones de trabajo con los usuarios y los miembros del equipo de desarrollo. En líneas generales podemos decir que la técnica consta de cuatro pasos:

Figura 3.1. Vista conceptual del Story Map

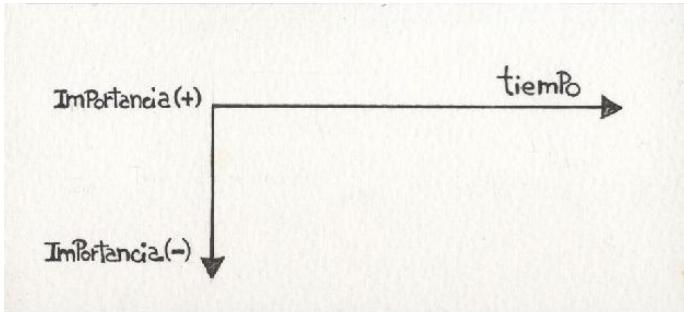


1. Identificar los procesos de negocio (objetivo).
2. Identificar los usuarios.
3. Identificar las actividades de los usuarios.
4. Identificar las funcionalidades del software.

Para aplicar la técnica es necesario contar con marcadores, cinta de papel, tarjetas (o notas autoadhesivas) de al menos tres colores distintos y una superficie de trabajo (una pizarra o mesa amplia). Sobre la superficie de trabajo trazaremos dos ejes utilizando cinta de papel: el eje horizontal representará el tiempo, mientras que el eje vertical representará importancia para el negocio (cuanto más arriba,

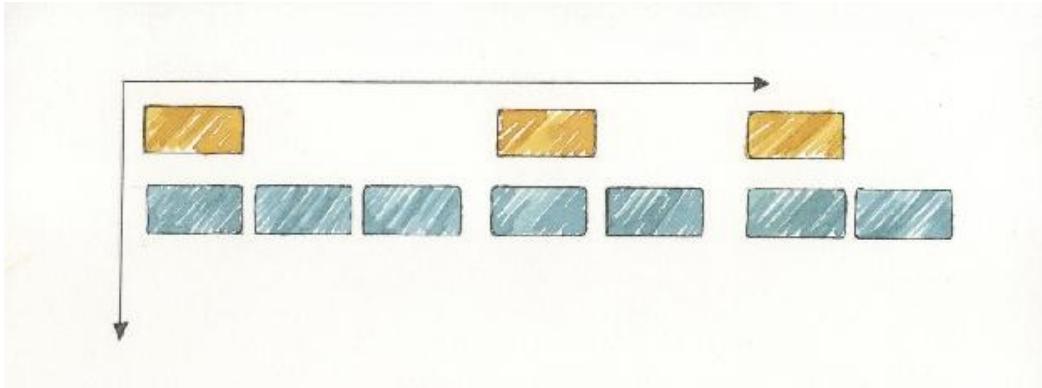
más importante).

Figura 3.2. Superficie de trabajo para el VSM



Entonces comenzaremos ubicando en la parte superior las tarjetas que constituyen los distintos procesos de negocio. A continuación, en un segundo nivel (utilizando tarjetas de otro color), ubicaremos las tarjetas con las distintas actividades que conforman el proceso de negocio, prestando atención a mantener la secuencia en que dichas actividades deben ejecutarse.

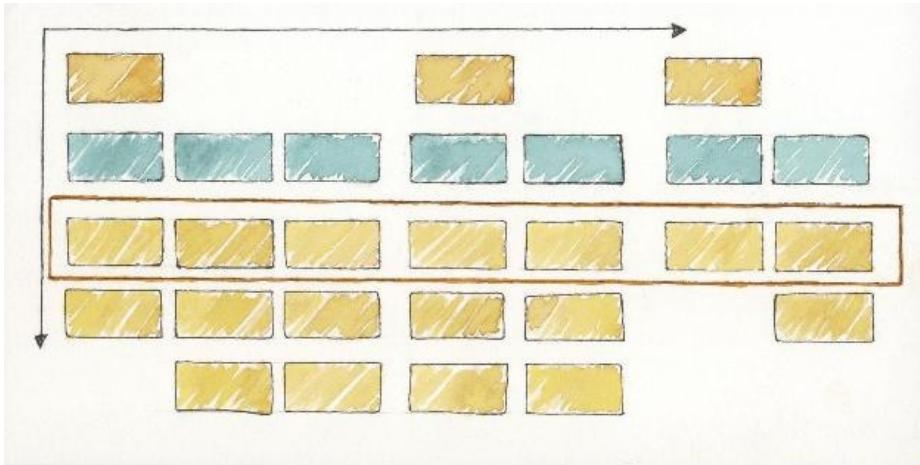
Figura 3.3. Identificación de procesos de negocio y actividades



Por último, en un tercer nivel, ubicaremos las funcionalidades que nuestra aplicación deberá proveer para que puedan completarse las actividades identificadas previamente. Es común que para completar una actividad dada, el sistema deba proveer más de una funcionalidad. Es por eso que nos encontraremos con varios niveles de tarjetas de funcionalidades, siendo los de más arriba, los más importantes. Del mismo modo puede suceder que algunas de las funcionalidades resulten indispensables para el negocio mientras que otras sean complementarias. Es importante atender a estas cuestiones al ubicar las tarjetas con las funcionalidades del sistema.

El primer nivel de tarjetas debería incluir aquellas que constituyen el conjunto mínimo de funcionalidades necesario para completar el flujo de negocio.

Figura 3.4. Visión final del Visual Story Map



Al finalizar la actividad de Visual Story Mapping tendremos:

- El mapa de funcionalidades que nuestra aplicación deberá implementar y no solo eso sino que tendremos una visión global de como cada funcionalidad encaja en el contexto de negocio que pretendemos resolver.
- El conjunto de funcionalidades que constituyen nuestro backlog de proyecto, siendo las funcionalidades del primer nivel las de mayor prioridad.

El resultado del Visual Story Mapping es un mapa “físico” de las stories que constituyen el sistema. Generalmente este mapa no se descarta, sino que es colgado en algún lugar visible dentro del espacio de trabajo de equipo, para tener como referencia de contexto a lo largo de todo el proyecto.

Con esto ya estamos en condiciones de estimar y planificar el desarrollo de la aplicación, pero esas son cuestiones que trataremos en el siguiente capítulo.

User stories

Al describir la técnica de Visual Story Mapping hemos hablado de funcionalidades de la aplicación. En los contextos ágiles dichas funcionalidades suelen representarse con user stories. Alguien podría pensar que las user stories son especificaciones de requerimientos, lo cual no es correcto. La definición purista indica que una user story es un recordatorio de algo relevante que debe hablarse con el usuario, lo cual hace que a lo sumo las user stories puedan considerarse el título de un requerimiento.

Las user stories tienen una forma muy simple, consiste en una oración escrita en el lenguaje del negocio que generalmente se expresa utilizando el siguiente patrón:

Como <rol> quiero <funcionalidad> para <beneficio> Algunos ejemplos comunes de user stories para un portal de comercio electrónico podrían ser:

- Como comprador quiero buscar productos por rango de precio para ver solo aquellos que estén a mi alcance.
- Como vendedor quiero poder duplicar una publicación pasada para evitar cargar toda la información otra vez.
- Como moderador quiero editar las publicaciones para asegurarme que respeten las políticas de la empresa.

Las user stories son un artefacto propuesto por Extreme Programming, cuyo uso y popularidad se ha extendido mucho más allá de este método particular. Posiblemente parte de su gran popularidad se deba al libro de Mike Cohn [Cohn 2004], el cual es una referencia obligada para quienes deseen ahondar en el tema.

User Stories vs. Casos de uso

Es común que en una primera aproximación tienda a verse a las user stories como análogas a los casos de uso del Proceso Unificado, en el sentido que ambos artefactos describen en cierto modo una funcionalidad del sistema. Esta analogía no parece apropiada, ya que más allá de la forma de estos artefactos, hay una diferencia radical en el propósito de cada uno, dado por el contexto metodológico en el cual se usan.

Habitualmente, al trabajar con casos de uso se tiende a generar documentos que especifiquen con bastante nivel de detalle la funcionalidad que el programador debe implementar. Por su parte, las user stories son intencionalmente vagas, pues lo que buscan es promover el diálogo entre quien debe implementar la funcionalidad y quien la ha requerido. Es justo esta diferencia de enfoque la que lleva a que en general no se utilicen casos de uso al trabajar con métodos ágiles.

N.P.

Propiedades INVEST

Al hablar de user stories se suele hacer referencia a ciertas propiedades deseables que estas debieran cumplir. Dichas propiedades enunciadas en inglés forman la sigla INVEST:

- *Independent* (independiente): en el sentido de independiente de las demás; esto nos brindará más libertad a la hora de planificar y al mismo tiempo debería ayudarnos a evitar ambigüedades a la hora de estimar.
- *Negotiable* (negociable): una user story no es un contrato de funcionalidad, pues sus detalles van evolucionando y definiéndose conjuntamente entre el cliente y el desarrollador a medida que se desarrolla.
- *Valuable* (valiosa): si una story no tiene valor para el cliente entonces no tiene razón de ser.
- *Estimable* (estimable): si una story no puede ser estimada por el equipo entonces no es posible que este pueda asumir un compromiso para su construcción.
- *Small* (pequeña): al ser pequeñas serán más fáciles de estimar, tendrán menos ambigüedades y darán una mayor flexibilidad a la hora de planificar.
- *Testable* (que puede probarse): tenemos que poder probarla para que podamos definir una condición de aceptación, sin esto ¿cómo saber cuando está terminada?

Más allá que estas propiedades se han popularizado con las user stories, la realidad es que son propiedades deseables para todo requerimiento, estemos o no trabajando con un enfoque ágil.

Story Cards

Durante el proceso de análisis, ya sea que se use la técnica de Visual Story Mapping o no, las user stories suelen escribirse en fichas bibliográficas que se denominan Story Cards. Cada story card tiene el enunciado de la user story (Como <rol> quiero <necesidad> para <beneficio>), el valor de negocio que tiene la story y la cantidad de story points que el equipo le asignó en la estimación.

Las user stories brindan muy poca información respecto de la funcionalidad, por eso es que se suele decir que son simplemente un recordatorio de algo que debe hablarse con el cliente. En consecuencia, lo importante no son las user stories en sí, sino la discusión que se da en torno a ellas.

Por último, para dejar bien en claro las expectativas respecto de la funcionalidad provista por cada user story, al dorso de la story card se suelen escribir las condiciones de aceptación de la story, las cuales funcionan como una especificación para quien tenga que implementarla.

En resumen, estas tres particularidades suelen ser referidas en inglés como CCC:

- Card: la tarjeta física donde se escriben las stories.
- Conversation: la discusión que debe darse entre el cliente y el equipo de desarrollo en torno a cada user story.
- Confirmation: los criterios de aceptación que verifican el cumplimiento de las user stories.

Figura 3.5. Story Card



Épicas y Temas

En una etapa temprana del proyecto es común que no se tenga demasiado detalle sobre algunas funcionalidades. Más allá de esto, puede que ya se sepa que algunas funcionalidades serán muy grandes y seguramente requieran más de una iteración para ser implementadas. A estas user stories “grandes”, que por serlo no cumplen con las propiedades INVEST, se las suele llamar épicas.

Por otro lado, en ocasiones resulta útil, ya sea por cuestiones de negocio o de planificación, agrupar conjuntos de user stories solo para facilitar su identificación. Estos conjuntos de user stories suelen denominarse temas.

Visual Story Mapping y User Stories

Al hacer Visual Story Mapping es común enunciar las funcionalidades de la aplicación como user stories, pero sin entrar en mayor detalle que su enunciado y su prioridad. O sea, tendremos una story card que ubicaremos en el mapa que solo contendrá el enunciado de la story y su valor de negocio. A esta altura la story card no tendrá condiciones de aceptación, pues estamos en un etapa muy temprana donde recién estamos definiendo qué debe hacer nuestra aplicación. Incluso es posible que algunas user stories no cumplan con el criterio INVEST o directamente sean épicas.

Otras técnicas

Existen algunas otras técnicas de uso común en contextos ágiles para la identificación del alcance del proyecto.

Una de estas técnicas es la conocida como Impact Mapping, desarrollada por Gojko Adzic [Adzic2012]. Esta técnica va más allá de la identificación del alcance, trabaja sobre la planificación estratégica, con un foco importante en la comunicación y colaboración entre los involucrados técnicos y de negocio.

Otra técnica de análisis ágil es la denominada Product Canvas [Pichler 2013]. Esta, al igual que el Visual Story Mapping propone una alternativa al clásico Product Backlog lineal. La misma hace un importante foco en los destinatarios del software en construcción y está basada en varios elementos visuales que se ubican sobre un canvas físico.

Por último, nos parece relevante destacar que más allá de las técnicas aquí mencionadas, hay otras herramientas tradicionales como los diagramas UML de clases, actividades y estados, que suelen resultar muy útiles para comprender el dominio de un negocio. Asimismo, es importante destacar que cuando se utilizan diagramas UML, los mismos no se suelen generar utilizando software, sino que se dibujan de manera informal en pizarras o papel, pues el fin es facilitar el entendimiento y no la documentación.

En resumen

En este capítulo hemos analizado un conjunto de técnicas y artefactos de uso común al trabajar con métodos ágiles. Como puede notarse en la descripción de cada técnica, se propone trabajar cara a cara con el cliente, valiéndose de herramientas físicas. Esto no impide que luego de cada actividad todo sea volcado en algún software, pero es importante que durante las sesiones de trabajo se utilicen las herramientas físicas (tarjetas, notas autoadhesivas y otras), puesto que permiten reacomodar elementos con facilidad, experimentando distintas alternativas.

Con lo visto en este capítulo hemos presentado el enfoque ágil para entender las necesidades de nuestro cliente y su negocio. Asimismo, como parte de este proceso de entendimiento, hemos definido un conjunto de artefactos que nos servirán como entrada para las siguientes actividades del proceso de construcción.

¹¹ El término backlog de producto, en inglés product backlog, fue acuñado en Scrum, pero en la actualidad se lo usa más allá de Scrum.

¹² La sigla WBS hace referencia a Work Breakdown Structure, un artefacto de uso muy común en la gestión tradicional de proyectos.

Estimar no es predecir

La estimación en los proyectos de software siempre ha sido una cuestión de debate. Incluso existe un libro llamado *Estimación de Software: desmitificando el arte negro*¹³ [Mc Connell 2006]. Han sido propuestos diversos métodos de estimación con distintos niveles de formalidad, pero ninguno ha logrado imponerse con claridad.

En este capítulo veremos la forma en que los métodos ágiles encaran esta temática junto con algunas de las técnicas más populares. No se pretende hacer una revisión de las teorías y técnicas de estimación, sino hacer un breve repaso de algunos puntos y sí destacar y ahondar en los puntos en los que los métodos ágiles aportan una visión distinta.

Cuestiones generales de estimación

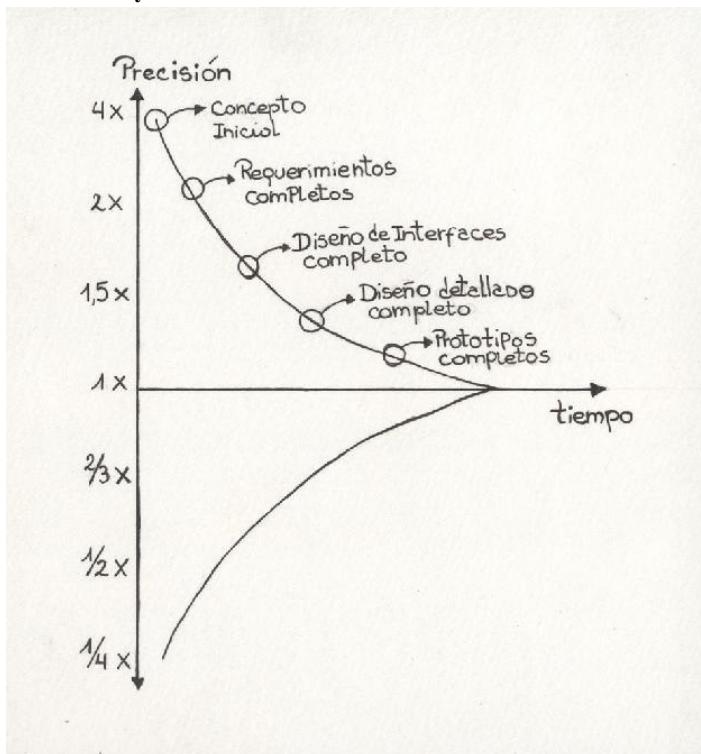
Para qué estimar

Es necesario para poder planificar. En primer lugar, el cliente necesita saber cuándo tendrá el producto y cuanto costará. El área de infraestructura necesita saber para cuando preparar el ambiente necesario para su puesta en producción. El equipo necesita saber cuando terminará el proyecto para comenzar el siguiente. Y así, cada interesado en el proyecto necesita cierta visibilidad para organizar sus tareas.

Cuándo estimar

Uno de los factores que más influye al estimar proyectos de software es el conocimiento que uno tiene del dominio del negocio y de las herramientas que utilizará para la construcción. Esto se ve claramente en el famoso cono de incertidumbre de Barry Boehm (figura 4.1), el cual nos indica que la incertidumbre disminuye a medida que avanzamos en el proyecto, dado que cada vez es mayor el conocimiento que el equipo tiene del negocio y de su propio funcionamiento. Es por esto que resulta razonable estimar a medida que se avanza en el proyecto.

Figura 4.1. Cono de incertidumbre: en el eje vertical se muestra el nivel de incertidumbre del equipo de proyecto mientras que el eje horizontal representa la dimensión temporal. A medida que transcurre el proyecto, el equipo aumenta su conocimiento y el nivel de incertidumbre decrece.



Puntualmente, tendremos dos momentos distintos de estimación con dos objetivos distintos¹⁴.

En un primer momento realizaremos una *estimación inicial o de orden de magnitud*. Esta estimación la haremos antes de empezar el proyecto, durante la etapa de preventa o inicio de proyecto si quisieramos ponerlo en términos del Proceso Unificado. A esta altura, el cliente tiene una visión del proyecto, pero no todos los detalles. Para seguir adelante necesita cierta información de nuestra parte: tiempo que tomará el desarrollo y los costos. Es en este contexto que realizaremos nuestra primera estimación para, entre otras cosas, saber si estamos hablando de un proyecto de 3 meses o 2 años.

El segundo momento será al comienzo de cada iteración. Estas estimaciones periódicas, se realizan en un contexto diferente al anterior: ya tendremos más información, estaremos cada vez más familiarizados con el dominio del problema y con la tecnología y, tal como indica el cono de incertidumbre, nuestra probabilidad de error será cada vez menor. El objetivo de esta estimación es distinto: la primera fue para determinar un orden de magnitud del proyecto, mientras que esta es para que el equipo pueda asumir un compromiso de los ítems que entregará al final de la iteración actual.

Quién estima

Dependiendo de la técnica de estimación que se utilice, el encargado puede variar. En el caso de las utilizadas en entornos ágiles, la propuesta es que la realicen los miembros del equipo que realizarán el desarrollo. Esto se basa en la simple idea que resulta poco lógico que alguien tome un compromiso en base a una estimación realizada por un tercero.

Quién estima

Dependiendo de la técnica de estimación que se utilice, el encargado puede variar. En el caso de las utilizadas en entornos ágiles, la

propuesta es que la realicen los miembros del equipo que realizarán el desarrollo. Esto se basa en la simple idea que resulta poco lógico que alguien tome un compromiso en base a una estimación realizada por un tercero.

Estimaciones en el enfoque ágil

Siendo conscientes de la situación evidenciada por el cono de incertidumbre previamente mencionado, resulta evidente que no tiene sentido utilizar escalas de estimación muy precisas. Es por esto que al trabajar con métodos ágiles suelen utilizarse ciertas escalas de estimación predeterminadas. Las dos escalas más comunes son los “talles de camiseta” y los valores de la serie de Fibonacci [Cohn 2006].

- Talles de camiseta: chico, medio, grande, extragrande.
- Fibonacci¹⁵: 1, 2, 3, 5, 8, 13, 21, ...

El uso de una u otra depende del nivel de abstracción de la estimación. En una estimación en etapa de pre-proyecto (pre-venta) resulta perfectamente lógico estimar con talles de camiseta, mientras que en una de comienzo de iteración, puede resultar más conveniente utilizar la serie de Fibonacci. Al mismo tiempo, como veremos más adelante, suele recomendarse el uso de unidades de estimación abstractas [Shore 2007].

Estimación != planificación

Como ya dijimos, uno de los objetivos de la estimación es la planificación. Lamentablemente esto suele ocasionar infinidad de problemas: el equipo da una determinada estimación para un conjunto de actividades y automáticamente algún gerente convierte esa estimación en un compromiso y arma todo un plan de proyecto en base ello.

No olvidemos que la estimación solo nos da una idea de lo que nos puede insumir cierta actividad, pero bajo ningún punto de vista representa un compromiso de parte del equipo. Volveremos sobre este tema en el capítulo siguiente dedicado a cuestiones de planificación.

Estimación y diseño

El proceso de estimación implica establecer ciertos supuestos al respecto de cómo se construirá el software. Dichos supuestos muchas veces involucran cuestiones de diseño de alto nivel que podrían tener una gran influencia en la estimación. Un ejemplo de esto, muy común en estos días, son los reportes: es radicalmente distinto tener que programar todos los reportes manualmente frente a usar librerías como Crystal¹⁶ o Jasper¹⁷ o incluso utilizar un servicio como el de Google Analytics¹⁸. Por esto es que a la hora de estimar, puede que el equipo lo haga asumiendo una estrategia en particular, ya que el uso de una u otra estrategia podría implicar variaciones en el orden de magnitud de la estimación.

Story points

Para medir el tamaño del software en el desarrollo ágil se suelen usar story points. Son una medida de tamaño relativo, ya que al estimar con story points lo importante no son los valores puntuales de cada ítem, sino los relativos de los mismos. Cuando decimos que un ítem dado tiene tamaño de un story point y otro de dos, estamos diciendo que el segundo es el doble de grande que el primero. Dada esta importancia del valor relativo, muchas veces a la hora de estimar se comienza por identificar el ítem de menor tamaño y tomarlo como unidad de estimación, para luego estimar el resto en base a la unidad establecida.

El uso de estas unidades no convencionales está en cierto modo sustentada por la intención de utilizar unidades que no sean traducibles a tiempo, pues nunca faltan los “entrometidos” que pretenden llevar la estimación del equipo a un compromiso de calendario. La estimación es un proceso de entendimiento, previo y necesario para la planificación. Insistimos, bajo ningún punto de vista la estimación es un compromiso. De lo uno a lo otro hay varias cuestiones intermedias.

Días ideales

Para entender lo que significa la medida en días ideales tomemos el ejemplo de un viaje en la ciudad. Pensemos que tenemos que manejar desde un punto A a un punto B, ¿cuánto tiempo nos llevaría esa travesía? Es algo incierto, pues hay una cantidad importante de factores (un accidente o un desvío) que podrían influir. Pero si pudiéramos desestimar todos esos factores y conocíramos la distancia entre los dos puntos, podríamos suponer una velocidad promedio y así calcular el tiempo que nos insumiría la travesía. El tiempo insumido por esta travesía “perfecta”, sin interrupciones, es tiempo ideal.

Muchas veces al comenzar a trabajar con el enfoque ágil, los equipos suelen preferir utilizar días ideales, pues les resultan mucho menos abstractos que los story points.

Spikes

Al trabajar en desarrollo de software puede que nos encontremos con tareas que nunca antes hemos realizado como, por ejemplo, utilizar un determinado componente, interactuar con una determinado servicio, etc. Esta falta de experiencia/conocimiento nos hace imposible que estimemos el tiempo de realización. En estos casos, el enfoque ágil es hacer un spike.

Básicamente, es una tarea que se hace para despejar incertidumbre sobre otra tarea. En concreto: si una user story requiere interactuar con un componente de hardware vía interface bluetooth, siendo esto algo que nunca hemos hecho, entonces no podremos estimar la user story y faremos un spike.

En sí, el spike es una tarea de investigación o experimentación. Es importante no perder de vista el objetivo: obtener el conocimiento necesario para poder estimar la tarea/user story que dio origen al spike.

Una característica importante de todo spike, es que es time-boxed, pues si no lo fuera, uno podría estar “infinitamente” investigando un tema. Otra cuestión recomendada es definir un entregable como resultado del spike, que podría ser un fragmento de código que muestre cómo utilizar el componente que se investigó o una respuesta concreta a la pregunta que planteó el spike. Más allá de esto, el principal resultado debe ser la capacidad de poder estimar la tarea que le dio origen.

Técnicas de estimación

Existen diversas clasificaciones de los métodos de estimación, pero de forma simplificada podemos clasificar las técnicas de estimación en dos grandes grupos: las paramétricas y las basadas en opiniones.

En líneas generales, podemos decir que las técnicas paramétricas proponen algoritmos que, tomando como entrada distintos parámetros del proyecto (complejidad, conteo de entradas, conteo de salidas, experiencia de los miembros del equipo, prácticas de programación utilizadas, etc.), proveen una estimación del esfuerzo requerido para completar el proyecto.

Por su parte, las técnicas basadas en opiniones, como su nombre lo indica, dependen de la opinión de personas. En la bibliografía, estas técnicas suelen denominarse “basadas en expertos”, pues la idea es que las personas que estiman sean los expertos. Pero como en la práctica esto no siempre se cumple, hemos decidido denominarlas técnicas basadas en opiniones. Una de las más populares de este grupo es la conocida con el nombre de Delphi, la cual ha dado origen a una serie de variantes entre las que se destaca la denominada Wideband Delphi, desarrollada por Boehm y Farquhar en la década de 1970 y popularizada luego en el libro *Software Engineering Economics* [Boehm 1981].

Las técnicas de estimación ágil pertenecen al grupo de técnicas basadas en opiniones.

Planning poker

Possiblemente es la técnica de estimación más difundida en ambientes ágiles. Lo primero que debemos notar es que si bien su nombre hace referencia a planificación, la misma es sólo una técnica para organizar una sesión de estimación y no cubre cuestiones de planificación.

Antes de entrar en el detalle de la dinámica dejemos en claro algunas cuestiones de contexto:

- El objetivo de esta técnica es consensuar una estimación de tamaño o esfuerzo sobre un conjunto de ítems que generalmente son user stories.

- Los estimadores son los miembros del equipo de desarrollo.

- Es indispensable la participación del responsable de producto.

Para utilizar esta técnica son necesarios dos artefactos. El primero es el conjunto de story cards a estimar y el segundo es un mazo de cartas de planning poker para cada estimador. Cada una de las cartas del mazo tiene un valor dictado por la escala de estimación a utilizar. Como se mencionó anteriormente, la escala más utilizada es la dada por los valores de la serie de Fibonacci.

Al igual que todas las técnicas de estimación basadas en opiniones, es necesario que, antes de comenzar, los estimadores acuerden algunas generalidades y pongan en claro cualquier suposición preliminar como ser: unidad de estimación (*¿story points o días ideales?*), tecnología¹⁹ a utilizar, el alcance de la estimación (*¿estamos estimando el esfuerzo de análisis e implementación o también el de la prueba y la aprobación?*), etc.

Figura 4.2. Cartas utilizadas para hacer Planning Poker.



Uno de los participantes toma el rol de moderador, generalmente el responsable de producto, pero podría ser cualquiera. Este moderador toma una story card y la lee en voz alta. Los estimadores escuchan y realizan preguntas para despejar dudas, las cuales son contestadas por el responsable de producto. Es común que durante este intercambio de preguntas-respuestas, surjan las condiciones de aceptación de la user story, las cuales son anotadas al reverso de la story card²⁰.

Una vez que se han contestado todas las preguntas, cada estimador elige en forma privada la carta de su mazo que representa su estimación. Cuando todos han elegido su carta, cada uno exhibe la suya simultáneamente con los demás. De esta forma se busca evitar que la opinión de un estimador influya en el resto. Con las cartas dadas vuelta sobre la mesa se analiza la situación. Si todos los estimadores han dado el mismo valor, podemos dar por terminada la ronda: se toma ese valor como estimación, se lo anota en la story card y se pasa a la siguiente. En caso que haya diferencias, el estimador de la estimación más alta y el de la más baja discuten sus diferencias, no para intentar convencer el uno al otro, sino para intentar encontrar distintas interpretaciones de la story que pudieran ser la causa de la diferencia. Esta discusión debería arrojar más información, alineando las distintas interpretaciones. A la luz de la nueva interpretación, se repite el proceso: cada estimador vuelve a elegir un valor en forma privada y luego se comparten los valores elegidos. El proceso se repite hasta que el valor elegido es el mismo para todos los estimadores. No es habitual que se requieran más de tres rondas.

¿Y si luego de las tres rondas sigue habiendo divergencia en las estimaciones? Para resolver esta situación es importante no perder el foco. La estimación establecida debe ser consensuada entre todos, pues en una instancia posterior serán esos estimadores los que deberán asumir un compromiso. Imaginemos la siguiente situación: con el fin de cerrar la estimación de un ítem luego de varias

rondas, podríamos tomar un promedio de los valores extremos dados por los estimadores. Aquellos que hayan dado una estimación por debajo del valor promedio, seguramente se sentirán cómodos. ¿Pero qué hay de aquel cuya estimación está por encima?, ¿se sentirá cómodo?, ¿podrá esa persona, en una instancia posterior, asumir un compromiso basado en una estimación más optimista que la suya? Para evitar este tipo de situaciones hay distintos enfoques, de los cuales sólo describiremos dos:

- Tomar el valor más alto, pues de esta forma todos los estimadores estarán en condiciones de asumir un compromiso posterior.
- Si estamos en una estimación de iteración, adelantarse en el tiempo y pensar ya en el momento del compromiso, entonces se toma la estimación de la persona que asumirá el compromiso del ítem.

En resumen

En este capítulo hemos visto el enfoque de los métodos ágiles para la estimación en proyectos de software. Se presentaron algunas técnicas y los artefactos más utilizados.

Resulta interesante destacar que algunas de las cuestiones presentadas son aplicables, incluso a contextos más allá del desarrollo de software.

Puede que, a pesar de las explicaciones, el lector aun desconfie de la efectividad y/o utilidad de las técnicas, pero antes de hacer un juicio es necesario ver como se articulan con las técnicas de planificación y manejo de cambios, dos cuestiones que veremos en próximos capítulos.

13 Software Estimation: Demystifying the Black Art.

14 Dependiendo del tamaño y tipo de proyecto, puede haber más instancias de estimación. En general hay, al menos, dos como las aquí mencionadas.

15 La serie de Fibonacci se define como $f(n) = f(n-1) + f(n-2)$, con $f(1) = 1$ y $f(0)=1$. El uso de esta serie se debe a que sus valores representan saltos razonables de orden de magnitud.

16 <http://www.crystalreports.com/>

17 <http://community.jaspersoft.com/project/jasperreports-library>

18 <http://www.google.com/analytics/>

19 Decimos tecnología en sentido amplio, lo cual incluye lenguaje, framework, etc.

20 Anotar las condiciones de aceptación al reverso de la story card no es una práctica del Planning Poker, sino que es una práctica muy aceptada en el ambiente ágil. Para más información al respecto consultar la sección User Stories en el capítulo “Delineando el alcance”.

Planificación constante

En los capítulos anteriores vimos técnicas para identificar el alcance del proyecto y para poder estimarlo, en este veremos como, a partir de dicha información, es posible generar un plan ágil. Una de las frases del manifiesto ágil dice “Respuesta al cambio sobre seguir un plan”, lo cual ha creado el mito (erróneo) que al utilizar métodos ágiles no se planifica. Hablando seriamente ¿alguien en su sano juicio cree posible desarrollar un proyecto sin planificar? Al utilizar métodos ágiles se planifica y la planificación es una actividad muy frecuente. Al final de cada iteración se revisa lo realizado y al comienzo de cada iteración se planifica lo que viene. La diferencia del enfoque ágil en lo que respecta a planificación está en el horizonte del plan y en el nivel de detalle que este tiene. Veamos entonces cómo es la cuestión.

De la estimación a la planificación

Para hablar de planificación en el contexto ágil debemos comenzar por el manifiesto, el cual establece prioridades: La respuesta al cambio, por encima del seguimiento de un plan. Esta frase ha sido muchas veces malinterpretada a punto tal que se ha dicho que al trabajar con métodos ágiles no se planifica. El manifiesto apunta a que no tiene sentido hacer un plan al comienzo e intentar seguirlo sin considerar la forma en que se desarrolla el proyecto. Es natural que surjan cambios y es necesario adaptarse a ellos modificando el plan, lo cual nos lleva a una planificación continua. Como se suele decir, lo importante no es el plan sino la planificación. Por eso, es raro, en un contexto ágil, que se invierta mucho esfuerzo en generar diagramas de Gantt, como sí suele ocurrir en otros enfoques tradicionales.

Cuando hablamos de planificación tenemos que distinguir dos niveles: la planificación de alto nivel (conocida como planificación de versiones) y la planificación de iteración.

Pero antes de entrar en detalle sobre cada una es necesario introducir algunos conceptos.

El último momento prudente

Como ya hemos dicho, el trabajar con métodos ágiles implica adaptarse a los cambios. Al mismo tiempo, el costo de adaptación está relacionado al impacto que tengan los cambios en las decisiones tomadas. Una estrategia para minimizar ese costo de cambio es *posponer las decisiones hasta el último momento prudente*. Esto hace que a la hora de planificar el proyecto y tomar una determinada decisión, identifiquemos cuando es el último momento en que podemos tomarla y evitar así hacerlo prematuramente lo que implicaría un esfuerzo adicional para modificarla. Mientras tanto, reunimos la mayor cantidad de información posible para intentar tomar una mejor decisión.

Si bien esta idea puede resultar novedosa para algunos, la misma tiene varias décadas y fue introducida en el mundo del desarrollo de software por el movimiento de Lean Software Development a comienzos de la década de 2000 [Poppendieck 2003].

Velocidad²¹

Es la cantidad de trabajo que un determinado equipo es capaz de completar durante una iteración. Algo muy importante que se desprende de esta definición es que la velocidad es una propiedad particular de cada equipo. Si cambian los integrantes del equipo, entonces es otro equipo y por ende es muy probable que la velocidad sea distinta. Esto pone de relieve una vez más el manifiesto: *individuos y su interacción, por encima de los procesos y las herramientas*.

Otra cuestión que se desprende de la definición es que la misma está sujeta al tamaño de iteración. Por ende, si lo variamos, es de esperar que la velocidad se modifique.

Slack

Supongamos que hemos coordinado con unos amigos para que vengan a comer a nuestra casa el próximo jueves pero aún no hemos definido la hora. Sabemos que ese mismo día tenemos turno en el mecánico a las 20. El cambio de aceite lleva unos 30 minutos y el viaje desde el taller mecánico a nuestra casa otros 30 minutos. Entonces, ¿a qué hora le decimos a nuestros amigos que vengan a casa? En base a lo dicho, estimamos que estaremos en casa a las 21. Pero, ¿les decimos que vengan a esa hora? En general, no, porque sabemos que ante el mínimo imprevisto estaríamos dejando a nuestros amigos esperando en la calle. Si bien estimamos esa hora, nos comprometeremos para las 21:15. Ese es nuestro *compromiso*.

Luego de leer el ejemplo, seguramente algunos dirán que la diferencia entre estimación y compromiso es el tan conocido “colchón”, de 15 minutos en este caso, que hemos agregado. Y no es del todo incorrecto, pero en términos ágiles lo denominamos *slack* y la diferencia con el clásico colchón es que el esfuerzo o tiempo asignado al slack se planifica para realizar alguna tarea que no debe formar parte del compromiso asumido con el cliente aunque el equipo se planifica internamente para realizarla. Es muy común al planificar utilizar el slack para pagar deuda técnica²², reordenar el espacio de trabajo o para realizar actividades de investigación.

Planificación de alto nivel

La planificación de alto nivel en ambientes ágiles se denomina Planificación de versiones²³. La misma nos da una idea de cuánto nos llevará el proyecto, la cantidad de iteraciones que necesitaremos y cuando estarán disponibles las funcionalidades más importantes. Para realizarla, necesitaremos una estimación de orden de magnitud, la cual realizaremos al comienzo del proyecto. Para esta estimación, su pongamos que utilizamos Planning Poker para estimar cada uno de los ítems y obtenemos un valor final de story points o días ideales dependiendo de la unidad de estimación que hayamos decidido utilizar. Ahora la gran pregunta es: ¿cómo pasamos de cualquiera de estas unidades abstractas a una unidad concreta de tiempo o esfuerzo? Es aquí donde entra en juego la velocidad.

Como ya dijimos, la velocidad nos indicará la cantidad de trabajo que el equipo es capaz de realizar en una iteración, ya sea expresada en story points o días ideales. Entonces, si dividimos el número arrojado por nuestra estimación global del proyecto por la velocidad, sabremos aproximadamente la cantidad de iteraciones necesarias para completar el proyecto estimado. Esto nos lleva a la pregunta ¿cómo conocer la velocidad?

Estimación de la velocidad

Si el equipo ya ha trabajado como tal en otras ocasiones, tendremos una velocidad conocida, problema resuelto. Pero si el equipo es nuevo tendremos que hacer un trabajo adicional para determinarla²⁴. Aquí tenemos varias opciones posibles.

Una es tomar una velocidad basándonos en antecedentes de la organización. Si bien puede sonar una alternativa viable, la realidad es que no suena muy convincente, ya que como mencionamos al definir la velocidad, la misma es particular de cada equipo.

Otra alternativa sería determinar una conversión de story points (o días ideales) a días calendario. Para ello deberíamos tomar una user story, particionarla en tareas y estimar individualmente las tareas en horas. Con esto tendremos una conversión teórica de horas a story points (o días ideales). Si luego calculamos la cantidad de horas disponibles del equipo por iteración, entonces ya seremos capaces de determinar el valor de la velocidad.

Otra técnica para determinar la velocidad inicial es consultar la confianza del equipo. Esto es, una vez estimadas las user stories y determinada la duración de la iteración, el equipo va tomando una a una las user stories en orden de prioridad y se pregunta si podemos comprometernos a completar esta user story en el transcurso de una iteración junto a las otras user stories que ya hemos comprometido. En la medida que el equipo conteste positivamente a esta pregunta, se incrementa el valor de la velocidad acorde a lo que estaba estimada la story. Ejemplo: supongamos que establecemos que cada iteración durará 2 semanas, y el equipo cree que puede comprometerse a completar las primeras 4 user stories del backlog, las cuales suman un total de 16 story points. Entonces la velocidad inicial se establece en 16 story points.

Planificación de versiones

En primer lugar, debemos decir que cuando hablamos de versión nos referimos a una porción de software que agrega valor al cliente, lo cual implica software funcionando en un ambiente productivo. Como ya hemos indicado, el desarrollo ágil es iterativo e incremental, lo cual no es algo novedoso. Lo que sí puede resultar novedoso es que se propone salir a producción de manera frecuente. Por frecuente entendemos intervalos que van de 3 a 6 iteraciones, dependiendo del tamaño de las iteraciones del proyecto [Shore 2007]. El hecho de entregar frecuentemente se debe al énfasis puesto en la entrega de valor. Al mismo tiempo el adelantar la entrega de valor colabora a adelantar el retorno de la inversión, algo siempre bienvenido por los clientes.

Habiendo determinado la velocidad del equipo, podremos entonces determinar la cantidad aproximada de iteraciones que necesitaremos para completar el proyecto. Bastará con dividir la cantidad total de story points resultante de la estimación y dividirla por la velocidad.

El siguiente paso sería determinar la cantidad de versiones y qué stories formarán parte de cada versión. Para esto resulta muy útil el Visual Story Mapping, ya que como se muestra en la Figura 5.3, las versiones pueden identificarse a partir de cortes horizontales que agrupen las stories de cada versión. Una vez identificadas las versiones con su correspondiente contenido, es posible establecer fechas de liberación para cada versión.

Es importante destacar que al hacer planificación de versiones identificaremos las distintas versiones tentativas, pero solo entraremos en detalle en la primera de ellas, indicando tentativamente los ítems del backlog de las primeras iteraciones. Hay que notar que a este nivel de planificación, las funcionalidades no se partitionan en tareas y tampoco hay una asignación de trabajo a miembros del equipo.

Figura 5.1. Proceso de Planificación.

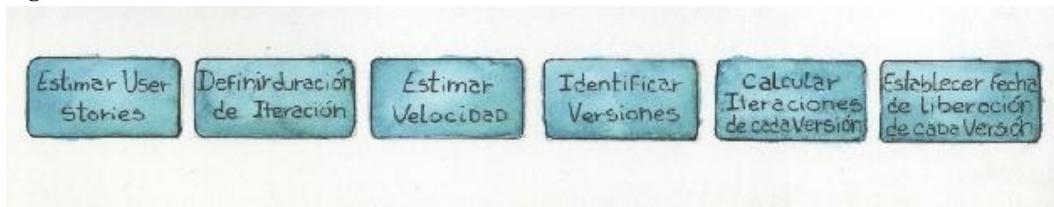


Figura 5.2. Plan de versiones.

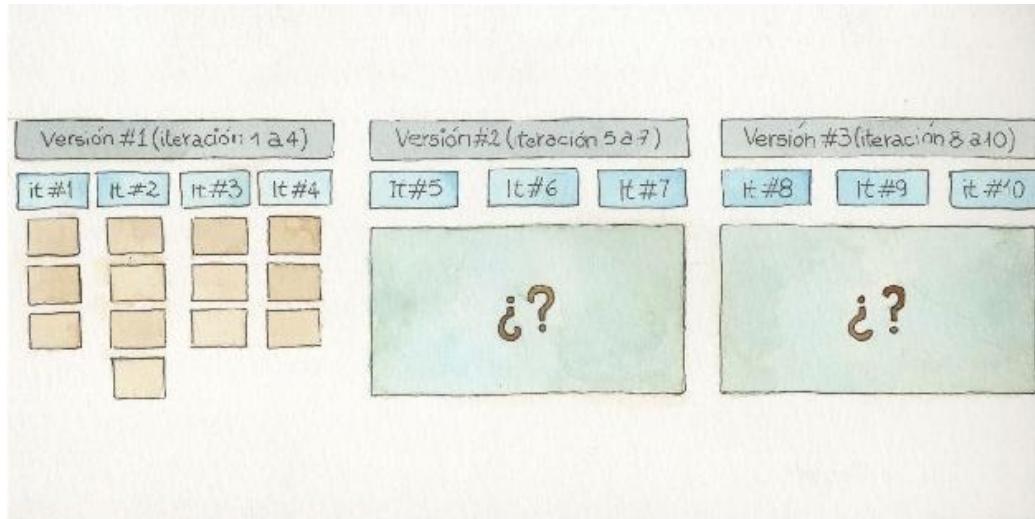
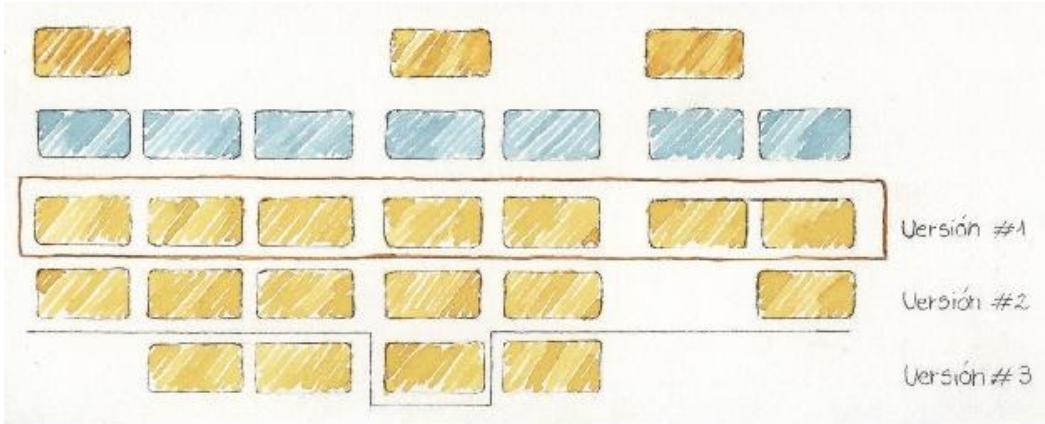


Figura 5.3. Plan de versiones visto desde el VSM.



Priorización de funcionalidades

Para determinar los ítems de backlog a incluir en cada iteración, en primer lugar, tendremos en cuenta el valor de negocio de cada uno de los ítems y luego las dependencias existentes entre ellos. Podríamos decir que hacer esto es básicamente un ejercicio de priorización que debe realizar el responsable de producto con la asistencia del equipo.

Para realizar esta priorización existen diversas técnicas. Una de ellas es el Visual Story Mapping que vimos en el capítulo “Delineando el alcance”, que además de ayudarnos a identificar el alcance del producto, nos permite identificar la prioridad de cada funcionalidad. Asimismo, el VSM nos explica el conjunto mínimo de funcionalidades necesarias para una primera versión.

Si bien todos los ítems del backlog tienen valor de negocio, cuando los analizamos desde la óptica de la planificación de versiones la cuestión es distinta, pues puede que algunos ítems en solitario no provean valor de mercado. Un ejemplo de esto puede ser la típica funcionalidad de inicio de sesión presente en casi toda aplicación web. Si bien esta funcionalidad seguramente tenga valor para el responsable de producto, la realidad es que aislada difícilmente resulte de valor para un usuario. ¿Qué usuario va a estar interesando en una aplicación que solo le permite iniciar sesión y nada más?

Es por esto que al trabajar en la planificación de versiones se suele hablar de MMFs (por sus siglas en inglés Minimum Marketable Feature). Un MMF es un conjunto mínimo de funcionalidades con valor para el mercado de la aplicación, entendiendo mercado en forma genérica, independientemente de si hablamos de un producto comercial o un desarrollo para usuarios internos a la organización. Entonces en términos de MMFs, cada versión de la aplicación debería incluir al menos un MMF.

En líneas generales, muchos sistemas de información consisten en una cierta cantidad de ABMs²⁵ y reportes asociados. En este contexto he visto casos de planes en los que se plantea completar primero todas las funcionalidades relacionadas a ABMs dejando para el final todos los reportes.

Esto no siempre es lo más apropiado desde un enfoque ágil, pues puede ocurrir que un determinado reporte tenga más valor que la funcionalidad de bajas. Es aquí donde la priorización juega un rol fundamental para lograr un retorno de la inversión más temprano.

N.P.

Planificación de iteración

La planificación de la iteración la realizamos para determinar los ítems que el equipo se comprometerá a trabajar durante la iteración actual. Esta planificación, como es de esperar, se realiza al comienzo de la iteración, en una reunión de la que participa todo el equipo y el responsable del producto. Veamos la dinámica de esta reunión de planificación.

El responsable del producto llega a la reunión con un conjunto de ítems de backlog seleccionados que deberían ser los de mayor valor de negocio. En general, cuentan con una estimación de orden de magnitud que se realizó al comienzo del proyecto, pero podría pasar que se trate de nuevos ítems y que, por lo tanto, no estén estimados.

En ambos casos, el equipo debería estimarlos otra vez, pues sin duda va a haber ganado más conocimiento del negocio, la tecnología y la arquitectura, y podrá realizar una estimación mejor contextualizada. Entonces, el responsable del producto va tomando uno a uno los ítems, que según las prácticas más habituales debieran ser user stories y como tales estar escritas en story cards, y los va leyendo en voz alta. El equipo escucha atentamente y realiza consultas al responsable del producto para entender los detalles de la user story. Durante este intercambio se van identificando condiciones de aceptación, que se van anotando al dorso de la story card y se va particionando el ítem en tareas. Una vez contestadas todas las dudas, el equipo estima cada ítem (con conciencia de todas las tareas involucradas) utilizando la técnica de Planning Poker explicada previamente²⁶. Esta dinámica continúa hasta que la suma de la estimación de los ítems alcanza la velocidad del equipo.

En el desarrollo de esta dinámica puede ocurrir que surjan algunas tareas que no estén directamente relacionadas con una user story particular, por ejemplo configurar un servidor para ejecutar las pruebas de aceptación. Este tipo de tareas también deben ser estimadas y tenidas en cuenta en la planificación.

Es importante destacar que el equipo asumirá un compromiso por user stories completas. Por ejemplo, supongamos que la velocidad del equipo es de 20 story points, que se ha realizado la dinámica descripta y se obtuvo la siguiente estimación:

En base a esto, el equipo debería comprometerse a completar las stories 1, 2, 3, 4, y 5, que suman 18 story points.

¿Qué hacemos entonces con los 2 story points que nos faltan para llegar a 20?

La respuesta es simple: Slack. Esto implica utilizar los story points sobrantes para realizar tareas como, por ejemplo, investigar alguna cuestión técnica, realizar alguna prueba de concepto o pagar deuda técnica.

Es importante que, sea cual sea la tarea elegida, la misma sea concreta; es decir, la misma no debería ser enunciada como “Pagar deuda técnica” sino que debería indicarse concretamente qué deuda técnica se pagará, por ejemplo “Refactorizar el componente de auditoría”.

Es clave que los story points sobrantes sean planificados en tareas concretas para evitar el famoso síndrome del estudiante y evitar

una distorsión en la velocidad del equipo.

Al finalizar la iteración, se actualiza el valor de la velocidad considerando los story points correspondientes a las user stories que fueron completadas. En caso que haya habido stories que no hubieran podido completarse, el valor de la velocidad se verá disminuido; por el contrario, si el equipo fue capaz de completar más trabajo del comprometido al comienzo de la iteración, entonces la velocidad se verá incrementada.

Para que la planificación basada en velocidad funcione, es necesario que la duración de las iteraciones sea constante y que las mismas sean time-boxed.

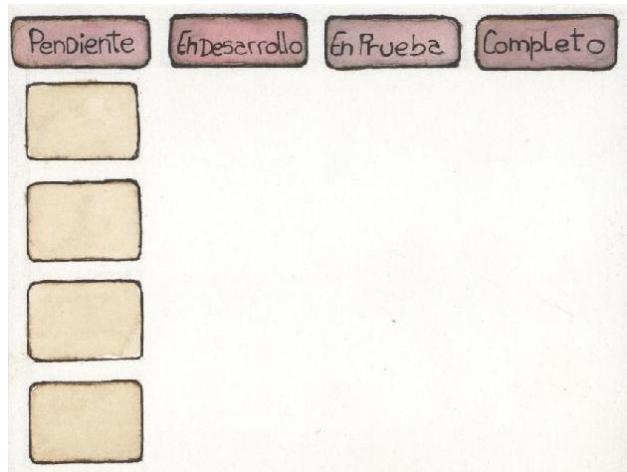
Finalmente, el resultado de la reunión de planificación queda plasmado en un tablero que cuenta con 4 columnas²⁷: pendiente, en desarrollo, en prueba y completo. Cada una de las stories comprometidas estarán pegadas en este tablero junto con sus tareas asociadas. Inicialmente todos los ítems del tablero estarán en la columna pendiente.

Táctica y Estrategia

Una forma interesante de dividir la reunión de planificación es considerar estratégica la primera parte, que trata de user stories y compromisos, incluyendo condiciones de aceptación, y táctica la que trata de las tareas necesarias para lograr implementar esas user stories. En general, los temas tácticos dependen más de una organización interna del equipo de desarrollo, y los estratégicos de su coordinación con el responsable del producto.

Una práctica cada vez más común en los equipos de desarrollo ágil es el uso de reuniones de refinamiento (también llamadas grooming sessions). Estas reuniones se hacen con el objetivo de refinar el backlog de producto de manera tal que el equipo y el responsable de producto lleguen mejor preparados a la reunión de planificación de iteración y que esta pueda ser más corta y efectiva. El refinamiento de backlog puede implicar tareas tales como: escribir nuevas stories, trabajar sobre las stories del backlog para que cumplan con las propiedades INVEST, estimar stories o definir criterios de aceptación de stories que serán parte de la próxima iteración.

Figura 5.4. Tablero de iteración.



En resumen

En el desarrollo ágil lo importante es la planificación y no el plan en sí mismo. Por eso, se planifica al comienzo de cada iteración, adaptando los planes a las necesidades del negocio. La planificación se realiza en base al compromiso que el equipo asume luego de estimar el trabajo a realizar. Tenemos planificación a dos niveles. La de alto nivel (planificación de versiones) tiene un horizonte de un par de iteraciones y solo indica tentativamente los ítems a entregar en cada iteración. La de bajo nivel o de iteración, indica los ítems y sus tareas asociadas solo para la iteración que se está iniciando. Al mismo tiempo, las versiones son de una duración variable en cuanto a cantidad de iteraciones, mientras que las iteraciones tienen una duración de tiempo fija.

²¹ En inglés se la llama Velocity.

²² Para más información sobre qué es la deuda técnica, consultar el capítulo “Arquitectura y diseño en emergencia”.

²³ Planificación de versiones es la traducción que hemos adoptado para el término Release Planning.

²⁴ Esta es una de las razones para mantener equipos formados.

²⁵ La sigla ABM hace referencia a Alta, Baja y Modificación, tres funcionalidades muy comunes en los sistemas de información.

²⁶ Estas tareas de planificación pueden ser realizadas también en reuniones de refinamiento del backlog durante las iteraciones anteriores

²⁷ Si bien aquí hacemos mención a un tablero de 4 columnas, existen diversas variantes, como veremos en el capítulo “Irradiando información”.

Empezando por la aceptación

Ya es viernes por la tarde y todo el equipo está muy cansado. Alejandro, que está probando la aplicación con el cliente, le dice a Marcio, desarrollador líder:

—Estuve probando incluir en la factura un ítem con precio unitario negativo, pero por algún motivo, no logro que funcione.

Marcio, satisfecho, contesta:

—Es que, precisamente, no tiene sentido que un artículo tenga precio negativo. Hasta incluí pruebas unitarias que validan que ningún precio unitario fuese negativo. ¿No es genial?

Sin embargo, Alejandro no está de acuerdo:

—Mm...Lamento tener que contradecirte, pero mientras definímos los requerimientos surgió esta duda y Diego, el analista, me dijo que podía haber ítems con precios negativos. Es decir, ningún producto puede tener precio negativo, pero hay ítems que corresponden a descuentos, cuyos precios sí pueden ser negativos.

Marcio no termina de entender por qué a él no se le ocurrió esa posibilidad y pide reunirse con Diego lo antes posible.

La reunión con Diego no hace más que confirmar lo que decía Alejandro: hay ítems, que no son productos, que tienen precios unitarios negativos. Incluso Diego les informa que ninguna factura puede tener un total negativo, cosa que ni Marcio ni Alejandro habían previsto en su diseño. Como conclusión, Diego decide comunicar al cliente que la entrega de esta semana no va a incluir los ítems con precio negativo y Marcio queda encargado de agregar estos dos temas a la lista de tareas a realizar en la semana siguiente. Por suerte es viernes; el lunes, con la mente despejada, podrán seguir trabajando.

En un proyecto tradicional suele haber roles, tales como clientes, especialistas de negocio, analistas funcionales, programadores, testers, entre otros.

Estos roles, que no son los únicos posibles, y que suelen adquirir nombres distintos según la metodología o la costumbre, tienen asignadas determinadas responsabilidades. Por ejemplo, el analista de negocio suele ser el único interlocutor con el cliente, que trabaja entendiendo sus necesidades y construyendo especificaciones de los requerimientos que primero valida con el cliente y que luego serán usadas por programadores y testers. Los programadores toman dichas especificaciones y construyen el producto (código, base de datos y demás) basándose en los mismos. Los testers parten también de los requerimientos elaborados por el analista de negocio, toman el producto tal como está construido por los programadores y verifican que se haya construido siguiendo aquellos requerimientos. Finalmente, en algún momento, el producto llega al cliente. Como este validó los requerimientos con el analista, el producto se construyó de acuerdo a estos y los testers verificaron el producto, el cliente debería estar conforme con lo que recibe.

Sin embargo, las pruebas que realizamos para comprobar la satisfacción del cliente luego del desarrollo, muestran a menudo una disconformidad con el producto recibido. En este capítulo vamos a ver qué proponen los métodos ágiles para resolver este problema.

Necesidad de criterios de aceptación

Una clave para mejorar los resultados es definir criterios de aceptación. Estos criterios son un conjunto de condiciones que nos deberían servir para definir, con la mayor claridad posible, los requerimientos justo antes de comenzar a desarrollarlos.

Si tuviésemos criterios de aceptación claramente definidos, y trabajados en conjunto entre el equipo de desarrollo y el cliente, las probabilidades de que este último no quede conforme al finalizar son mucho menores. Y todos ganaríamos confianza: el cliente en el equipo de desarrollo, y el equipo de desarrollo en el cliente.

Si, terminada una iteración, los criterios de aceptación de los requerimientos que se hayan encarado no son satisfechos por el producto, está claro que el cliente no va a estar conforme con lo que le entreguemos. Por lo tanto, conocer a priori los criterios de aceptación es una necesidad para todos los miembros del equipo de desarrollo, que pueden verificar lo que hacen contra esos criterios.

Si, en cambio, el producto satisface los criterios de aceptación, pero el cliente no está conforme, algo está mal en el proceso y eso es responsabilidad de todos los involucrados, tanto el cliente como el equipo. Por supuesto que esta realidad no es para ponernos contentos ya que se supone que el cliente debe estar satisfecho con lo que recibe. Estas situaciones deberían preocupar a todos los involucrados y ser tomadas como tema prioritario de la siguiente retrospectiva²⁸. En un proceso iterativo, esto no debería ocurrir en varias iteraciones seguidas, ya que se supone que mediante las retrospectivas vamos aprendiendo a no cometer los mismos errores.

Pero eso no es todo. Los criterios de aceptación definidos con rigurosidad sirven para evitar que trabajemos para desarrollar funcionalidades que el cliente no esperaba. El famoso *gold plating*²⁹, que se deriva de desarrollar más de lo que el cliente espera, no va a ocurrir, pues las funcionalidades superfluas no contaría con criterios de aceptación que las validen.

Por lo tanto, los criterios de aceptación sirven a los programadores para saber cuándo han completado el desarrollo de una funcionalidad.

La clásica pregunta “¿Cuándo sé que terminé mi trabajo con este requerimiento concreto?” se responde fácilmente: “cuando se satisfagan todos sus criterios de aceptación”.

Antes de seguir adelante, hagamos una aclaración. Cuando hablamos de criterios de aceptación para los requerimientos, no nos estamos restringiendo a los requerimientos funcionales. Los requerimientos de seguridad, de tiempo de respuesta, de escalabilidad, o cualquier otro no funcional o atributo de calidad, también necesitan criterios de aceptación claros que nos indiquen cómo se va a evaluar el producto una vez entregado. Tal vez incluso sean más importantes en estos casos que en los de requerimientos funcionales.

Por supuesto, los criterios de aceptación se pueden usar para elaborar indicadores de avance de un proyecto. Toda funcionalidad que satisfaga sus criterios de aceptación puede darse por concluida, con lo que la suma de requerimientos que cumplen con los criterios de aceptación nos brinda una medida clara del avance del proyecto y del valor entregado al cliente.

Como corolario:

Ningún requerimiento, funcional o de los otros, debería existir sin sus criterios de aceptación que permitan que el equipo sepa con qué se va a medir la corrección de aquello en lo que se está trabajando.

En otras palabras:

No deberían desarrollarse funcionalidades que no estén acompañadas por una prueba de aceptación. Y si durante una prueba, o en producción, surgiera un problema, habría que analizar que pasó con la prueba de aceptación correspondiente antes de resolverlo.

Los criterios de aceptación se especifican mediante pruebas

La forma más natural de especificar criterios de aceptación es mediante pruebas. Al fin y al cabo, si un criterio de aceptación es aquello que sirve para determinar si el producto cumple o no con las expectativas del cliente, ¿no es muy parecido a lo que debería ser una prueba?

En efecto, un caso de prueba de aceptación, bien entendido, es un criterio de aceptación escrito de manera tal que un tester pueda ejecutarlo y ver si el producto cumple los requerimientos. Y un requerimiento, según lo que definimos más arriba, es un artefacto que debe venir acompañado de los criterios de aceptación que definen lo que el cliente espera del sistema.

Por lo tanto, una buena manera de elaborar un criterio de aceptación es darle el formato de un caso de prueba. Solo que este caso de prueba no se construye por y para los testers, sino que le sirve también a los programadores para guiar el desarrollo, medir avance y saber cuando terminaron, y al cliente para validar la entrega.

Como corolario, hemos llegado a una nueva definición de lo que es un caso de prueba de aceptación:

Un caso de prueba de aceptación es un escenario de ejecución del sistema con ciertos datos que permite definir si se cumplen o no los criterios de aceptación definidos por el cliente.

Un ejemplo de criterio de aceptación para una story de extracción de dinero podría ser *si el usuario ingresa el monto 250 cuando su saldo es 100, el sistema debe rechazar la operación*.

Las pruebas de usuario son ejemplos de uso

Por otro lado, los casos de prueba suelen ser escritos para probar el comportamiento del sistema ante determinadas condiciones que el tester considera que deben ser probadas: el uso normal, tanto con valores típicos como con casos límite, entradas erróneas por parte de usuarios, fallas diversas, comportamiento en casos atípicos, entre otras. Pero, en cualquier caso, no son otra cosa que ejemplos de uso del sistema, teniendo en cuenta esas condiciones normales o excepcionales.

Lo que se busca al ejecutar pruebas es determinar si el sistema responde de acuerdo a lo esperado. En los casos de uso normal del sistema, se probará si responde a lo que se espera de él. Y en los demás casos también: si el manejo de errores es correcto, si se recupera correctamente luego de un fallo. Incluso puede que haya situaciones en las cuales se haya aceptado que el sistema no va a responder satisfactoriamente. Esos escenarios también deberían cubrirse por los casos de prueba.

Como se puede ver, estamos dando vueltas en círculos: planteamos que los requerimientos necesitan criterios de aceptación, que estos se expresan más claramente con casos de prueba, que estos son ejemplos de uso del sistema, y esos ejemplos surgen de los requerimientos.

Pero ese recorrido circular dista mucho de ser un problema.

En definitiva, necesitamos ejemplos de uso del sistema

Todo lo anterior nos dice que si construimos ejemplos de uso del sistema mientras elaboramos nuestros requerimientos, van a servir para los desarrolladores, porque les darán una idea clara de lo que se espera del requerimiento, cuales son sus límites, y cuando pueden saber que su trabajo está terminado. Esos mismos ejemplos podrían servir como casos de prueba para verificar la calidad de lo que se va a entregar. Para el propio cliente, esos ejemplos le pueden servir para ver si se comporta como él esperaba. Y para los analistas de negocio, son materializaciones prácticas de las reglas de negocio.

Esto último también es importante para lo que venimos diciendo. Un analista de negocio suele elaborar sus requerimientos en base a distintas técnicas, pero en la generalidad de los casos llevan a especificaciones abstractas, del estilo: “el usuario debe poder hacer envíos de dinero a través del sistema desde y hacia países de América y Europa”. En general, estas especificaciones de requerimientos se acompañan de una gran cantidad de aclaraciones, en la forma de excepciones, prototipos de pantalla, flujos de actividades y reglas de negocio. Las reglas de negocio también suelen quedar expresadas en forma assertiva como, por ejemplo, “los envíos de dinero sólo se podrán realizar en dólares estadounidenses, euros o monedas de países de la Unasur”.

El problema con estas especificaciones de requerimientos y reglas de negocio del sistema, es que no permiten ver con claridad que ocurre en casos límite: que debe hacer el sistema si el usuario intenta usar una moneda distinta o si el usuario desea seleccionar un país fuera de los permitidos, entre muchas otras posibilidades. Algunas de estas situaciones puede que sean informadas proactivamente por el cliente, o incluso que las prevea un analista de negocio sagaz y pregunte a aquel, y en ambos casos serán registradas como parte de los requerimientos o de las reglas de negocio. Pero suelen ser muchos los casos que se pasan por alto, y que recién se plantean cuando el tester se dispone a escribir casos de prueba, o incluso cuando tiene la aplicación a probar ya construida.

El inconveniente más serio de esta omisión es que se advierte luego de la construcción, cuando seguramente el programador ya tomó decisiones basadas en su mejor manera de entender los requerimientos, que puede o no coincidir con las expectativas del cliente.

Hay quienes argumentan en favor de las especificaciones abstractas por sobre los ejemplos concretos. Sin embargo, estos últimos, precisamente por ser concretos, son más fáciles de comprender y acordar con los analistas de negocio y otros interesados, que entienden mejor los ejemplos concretos.

Al fin y al cabo, habitualmente un analista de negocio escribe especificaciones, que le sirven a los desarrolladores y a los testers. Los desarrolladores construyen su código y definen pruebas unitarias, para las cuales deben basarse en ejemplos de entradas y salidas. Los testers, por su lado, desarrollan casos de prueba, que contienen a su vez ejemplos. En algunas ocasiones, para fijar las especificaciones, los desarrolladores y los testers le solicitan a los analistas que les den ejemplos concretos para aclarar ideas. E incluso hay circunstancias en que los propios analistas proveen escenarios que no son otra cosa que requerimientos instanciados con ejemplos concretos.

En definitiva, hay varias ocasiones en que se plantean ejemplos que sirven como complementos de requerimientos. Por eso no es raro que se haya pensado en especificar directamente con ejemplos, o al menos acompañando los requerimientos con ellos.

Estos requerimientos con ejemplos tienen ciertas ventajas, de las cuales las más importantes son:

- Sirven como herramienta de comunicación.
- Se expresan por extensión, en vez de con largas descripciones y reglas en prosa, propensas a interpretaciones diversas.
- Al ser más concretos, son más sencillos de acordar con los clientes.
- Fomentan que los ejemplos que se usen sean los mismos para las distintas actividades, evitando que cada vez se escriban unos distintos, con su potencial divergencia.

- Sirven como pruebas de aceptación.
- Clarifican los criterios de aceptación.
- Facilitan la detección de errores mientras el contexto está fresco en la mente de los participantes.

Algunas limitaciones

Sin embargo, no hay que olvidar que los requerimientos tradicionales tienen un nivel de abstracción mayor y expresan cuestiones que no siempre nos acordamos de llevar a los ejemplos. Por lo tanto, así como antes dijimos que todo requerimiento debe ir acompañado de pruebas de aceptación, también debemos poner el foco en que los escenarios o pruebas de aceptación deben estar claramente asociados a un requerimiento.

Además, no todo requerimiento puede llevarse a ejemplos. Por ejemplo, si una aplicación debe generar números al azar, los ejemplos que podamos escribir no van a servir como pruebas de aceptación.

Otra limitación de esta práctica se da en las situaciones en que la interacción es un requerimiento más importante que el comportamiento, como ocurre con las pruebas de usabilidad. Ya volveremos sobre esto.

Finalmente, nunca vamos a poder decir si el requerimiento quedó especificado en forma completa si solo definimos requerimientos con ejemplos, aunque sí vamos a poder garantizar que no sea ambiguo, validable contra las expectativas del cliente y verificable contra el producto.

Pero sigamos adelante para ver como se construyen estos ejemplos.

Una manera distinta de trabajar

Lo que necesitamos, en definitiva, es tener ejemplos desde el comienzo. Para ello, lo ideal es mantener reuniones del equipo de desarrollo (incluyendo todos los roles del mismo: cliente, analistas, testers, programadores) que permitan construir esos ejemplos.

Conviene que estas reuniones sean en modalidad de taller³⁰. La idea es ponerse de acuerdo en un lenguaje, una jerga, debatir y construir ejemplos en conjunto con el cliente, que luego sirvan para todo el equipo.

Aquí deben surgir los casos límite y las excepciones, y el equipo de desarrollo, incluyendo los que vayan a trabajar en testing, debe salir con todas sus preguntas respondidas o, al menos, con promesa de respuesta en el corto plazo.

El propio formato de taller hace que haya una revisión implícita, que surjan más requerimientos y reglas, y que se los tenga más presentes que cuando hay que leerlos de un documento escrito (esto independientemente de que luego se trasladen a un documento escrito).

Esas reuniones del cliente y el equipo de trabajo pueden tener distintas periodicidades, según las circunstancias del proyecto. Si podemos tener al cliente o un representante del mismo en contacto permanente con el equipo (que, como veremos en el capítulo “Reuniendo al equipo”, es lo más conveniente), las reuniones se podrán hacer al comenzar a trabajar en cada funcionalidad. Si el cliente solo puede mantener una presencia limitada o esporádica, habrá que agendar reuniones periódicas, tipo talleres, idealmente como mínimo al comenzar cada iteración. Si el cliente no está en el lugar, por ser un cliente remoto, vamos a necesitar hacer reuniones por videoconferencia o teleconferencia, cuanto más visuales mejor.

Lo que no puede ocurrir es que el cliente no esté presente siquiera en forma remota. Sin un cliente definiendo criterios de aceptación no hay criterios de aceptación, porque para poder construirlos de manera genuina se necesita de él, conocer sus opiniones y poder preguntar cara a cara. En este caso, como en tantos otros, la presencia del cliente es la clave del éxito.

No podemos dejar de recalcar que la comunicación cara a cara es fundamental en estas reuniones. Como veremos más adelante, las reuniones presenciales son una de las grandes recomendaciones del desarrollo ágil. En este caso, su importancia radica en que en situaciones en las que se está definiendo nada menos que lo que hay que hacer, los criterios de éxito y lo que no hay que hacer, la riqueza que da el lenguaje corporal no es reemplazable por otras formas de comunicación [Cockburn 2001].

Por eso, para evitar el síndrome del teléfono descompuesto³¹, se deben preferir siempre las reuniones cara a cara. Si no fuera posible, se pueden hacer talleres por videoconferencia, que es el mejor sucedáneo, sin la ventaja de la presencia real. Si esta modalidad también es imposible, la reunión telefónica podría servir, aunque estaremos perdiendo toda la información que brinda el lenguaje gestual y corporal. Finalmente, está el recurso de recurrir a la escritura, mediante mensajería instantánea o correo electrónico, que son desde todo punto de vista las peores alternativas, porque ni siquiera permiten dilucidar los tonos de voz³².

El diseño también se puede especificar con pruebas

Venimos hablando de construir criterios de aceptación de los requerimientos, con formato de ejemplos, que luego sirvan como casos de prueba del producto.

Sin embargo, no estamos teniendo en cuenta en todo esto al diseño, que también admite un tratamiento similar.

Una vez definidos los criterios de aceptación de un requerimiento, en forma de pruebas, se pueden escribir pruebas de mayor granularidad que, cumpliendo con estos criterios, sirvan como una medida de la calidad de su diseño e implementación.

Esta tarea la puede hacer un conjunto de programadores, un par de ellos, o simplemente un programador aislado antes de comenzar a desarrollar una funcionalidad.

Hace ya más de una década que surgió una práctica de diseño de software orientado a objetos, llamada TDD (Test-Driven Development³³), que se basa en derivar el código de pruebas escritas antes del mismo. Fue presentada como parte de Extreme Programming (XP) por Kent Beck [Beck 1999].

Se ha utilizado para poner el énfasis en hacer pequeñas pruebas de unidad que garanticen la cohesión de las clases, así como en pruebas de integración (de escenarios con varias clases) que aseguren la calidad del diseño y la separación de incumbencias, disminuyendo el acoplamiento. Dicho sea de paso, TDD recomienda que las pruebas sean especificadas en código, pero este tema lo analizaremos en el capítulo “Probar, probar, probar”.

Hay una regla de oro de TDD que conviene destacar: “Nunca escribas nueva funcionalidad sin una prueba que falle antes” [Beck 2002]. Otra dice: “Si no puedes escribir una prueba para lo que estás por codificar, entonces no deberías estar pensando en codificar” [Chaplin 2001]. El corolario obvio es que ninguna funcionalidad futura debería escribirse por adelantado, si no tiene el conjunto de pruebas que permita verificar su corrección. Si a eso se le suma que solo se debería escribir de a una prueba por vez, tenemos un

desarrollo incremental extremo, definido por pequeños incrementos que se corresponden con funcionalidades bien acotadas.

Notemos que estamos hablando de una práctica de diseño, no de control de calidad (o al menos no principalmente), ya que al escribir las pruebas antes del propio código productivo estamos derivando código a partir de las mismas. En definitiva, las pruebas explicitan el diseño del sistema.

Esto ha llevado a algunos equívocos que provienen del propio nombre de TDD, que incluye la palabra “test”. En efecto, mientras por un lado se afirma que es una técnica de diseño y no de pruebas, por el otro, el nombre invita a pensar otra cosa. Incluso las herramientas que fueron surgiendo a partir de TDD, desarrolladas incluso por los mentores de la práctica, requerían que el código de pruebas tuviese métodos cuyos nombres empezasen con *test* y las clases fueran descendientes de algo como *TestCase*³⁴. Nadie niega que TDD genera un buen conjunto de pruebas de regresión, pero ese no pretende ser su objetivo principal, sino más bien un efecto lateral positivo. Para colmo, es una práctica centrada en la programación, con lo cual no parece enfocada en especialistas del negocio ni testers; de hecho, los testers tradicionales tienden a desconfiar de ella por este mismo motivo.

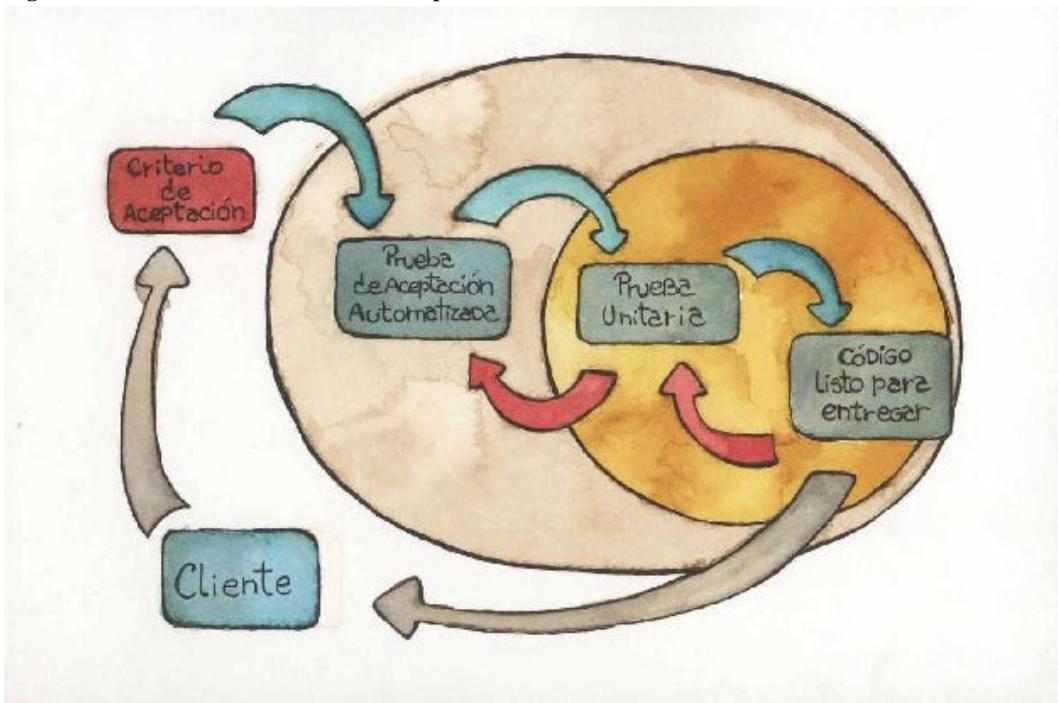
Un poco al pasar, dijimos que en los talleres de discusión de requerimientos con los clientes podía ir surgiendo el vocabulario del proyecto. Esto también es una cuestión que hace al diseño. Eric Evans [Evans 2003], al presentar su práctica de Domain Driven Design³⁵, asignó importancia fundamental a lo ubicuo del lenguaje, poniendo énfasis en que los nombres de las clases y métodos³⁶ de un sistema deberían surgir de los conceptos del dominio.

Efectivamente, el uso de un vocabulario común entre personas que cubren distintos roles y en actividades distintas es una ventaja que debería llegar hasta los nombres en el código.

El mundo al revés

Como estamos viendo, en el desarrollo ágil, empezamos creando criterios de aceptación como pruebas de mayor nivel. Luego, escribimos pruebas técnicas que nos permitan definir el diseño, incluyendo los nombres que vamos a usar, tanto en la comunicación con el equipo y el cliente, como en el código. Y, por último, construimos el código que responda a ambos tipos de pruebas. El diagrama de la figura 6.1 muestra este proceso.

Figura 6.1 Ciclo de desarrollo basado en pruebas.



Puede que a las personas que trabajan con procesos tradicionales les parezca descabellado escribir pruebas antes de construir el software que esas pruebas van a verificar. Creemos que lo explicado en este capítulo debería bastar para contar con buenas razones para hacerlo. Como veremos enseguida, estas ideas llevan ya bastante tiempo dando vuelta en el desarrollo de software, y se les ha dado nombres diversos.

Además, hay una ventaja adicional en escribir las pruebas basándonos en requerimientos o en cuestiones de diseño, antes de programar una sola línea de lo que las pruebas deberían verificar. Es una aspiración casi tan vieja como el desarrollo de software: al escribir las pruebas antes, estamos especificando lo que queremos (el qué) sin ceñirnos a una implementación (el cómo). Esto permitiría que más adelante realicemos implementaciones alternativas para las mismas pruebas, y esas implementaciones se podrían verificar con el mismo conjunto de pruebas que la implementación original.

De TDD a STDD

Como hemos dicho antes, si hiciésemos historia, la primera noción de TDD, y por lo tanto de las ventajas de escribir pruebas antes del código, fue conocida para el gran público a partir de la publicación del libro de XP [Beck 1999]. En este libro, Beck hacía énfasis tanto en las pruebas de menor nivel, como en las de aceptación, a las que llamaba “pruebas de cliente”.

Sin embargo, poco a poco, y sin que nadie pueda explicar bien por qué, TDD fue haciéndose sinónimo de algo como UTDD (Unit Test-Driven Development, o TDD con pruebas de unidad). Las herramientas que fueron surgiendo, entre ellas algunas en las que el mismo Beck tuvo mucho que ver, impulsaron aún más esta noción.

Como respuesta a este sesgo terminológico, que fue convirtiendo a TDD en sinónimo de UTDD, fueron surgiendo otros acrónimos, que pretendieron extender TDD para cubrir nuevamente a las pruebas de aceptación, bien en línea con lo que proponen los métodos ágiles que, como ya sabemos, tienen como foco principal el valor para el cliente.

Uno de los primeros que advirtió esto fue Dan North [North 2006], quien sostiene que poner el foco en el comportamiento logra un grado mayor de abstracción al escribir las pruebas desde el punto de vista del consumidor y no del productor. Por supuesto, recibió críticas por esto, al punto que hubo varios que dijeron que eso era lo mismo que hacer bien TDD³⁷.

Los acrónimos más habituales³⁸ son:

- TDD (Test-Driven Development): en teoría, el más abarcativo, pero habitualmente usado como sinónimo de UTDD. También se usa como una extensión de UTDD mediante pruebas técnicas de integración, tema que veremos en el capítulo “Probar, probar, probar”.
- UTDD (Unit Test-Driven Development): centrado en especificación del diseño detallado (por ejemplo, si trabajamos en programación orientada a objetos, a nivel de métodos y objetos).

• BDD (Behavior-Driven Development³⁹): se basa en criterios de aceptación, aunque se los denomine especificaciones de comportamiento. Se suele usar tanto para especificar requerimientos como diseño de alto nivel. Muchos lo consideran sinónimo de ATDD, STDD y SBE.

• ATDD (Acceptance Test-Driven Development⁴⁰): basado en criterios de aceptación exclusivamente. Muchos lo consideran sinónimo de BDD, STDD y SBE.

• STDD (Storytest-Driven Development⁴¹): basado en la especificación de requerimientos usando pruebas. Muchos lo consideran sinónimo de BDD, ATDD y SBE.

• SBE (Specification By Example⁴²): prácticamente lo mismo que STDD, aunque el foco está puesto en los ejemplos como herramienta de comunicación entre roles y de especificación de criterios de aceptación en forma de ejemplos.

No obstante, como todo lo que se refiere a TDD implica por lo menos automatización y refactorización, y esos son temas que trataremos en el capítulo mencionado.

- Mejorar la visibilidad de la satisfacción de requerimientos y del avance.
- Disminuir el *gold-plating*.
- Usar un lenguaje único, más cerca del consumidor.
- Mejorar la comunicación.

Tengamos en cuenta, además, que hay dos maneras de ver la calidad: una interna, la que les sirve a los desarrolladores, y otra externa, la que perciben usuarios y clientes. UTDD y la acepción más habitual de TDD apuntan a la calidad interna, mientras que BDD, ATDD y STDD apuntan a la externa. Otra manera de verlo es que, así como UTDD pretende ser una técnica de diseño detallado, BDD se presenta como una de diseño basado en dominio y ATDD una de requerimientos. En el capítulo “Probar, probar, probar” nos centraremos en temas de automatización de todo este proceso.

³⁸ Una retrospectiva es una reunión típica de los proyectos ágiles, que se realiza para analizar qué se puede mejorar y qué mantener, con foco en la última iteración. Como veremos en el capítulo “En retrospectiva”, es una instancia de mejora y aprendizaje.

³⁹ En la jerga de administración de proyectos, se denomina “gold-plating” (literalmente, “enchapado en oro”) a incorporar funcionalidades o cualidades a un producto aunque el cliente no lo necesite ni lo haya solicitado.

⁴⁰ Llamamos “taller” a una reunión presencial, en la que varias personas, actuando como pares, discuten puntos de vista, plantean dudas, hacen preguntas, proponen alternativas y clarifican conceptos.

⁴¹ Nos referimos al juego del teléfono descompuesto o roto, como se lo conoce en distintos países. Es un juego en el que se ve cómo un mensaje se va distorsionando al pasar de un participante a otro, al punto de quedar irreconocible al final de una ronda.

⁴² Véase el patrón “Face to Face Before Working Remotely” (en castellano, “Cara a cara antes de trabajar en forma remota”) en [Coplien 2004].

⁴³ En castellano, “Desarrollo guiado por las pruebas”.

⁴⁴ Si bien NUnit y la versión 4 de JUnit mejoraron esto, sigue estando presente la palabra Test en las anotaciones que utilizan.

⁴⁵ En castellano, “Diseño guiado por el dominio”.

⁴⁶ Evans trabaja dentro del paradigma de objetos.

⁴⁷ Véase, por ejemplo, [Glover 2007].

⁴⁸ Hay decenas de acrónimos y nombres que se han difundido. Aquí enumeramos los más habituales, incluso sabiendo que los límites entre lo que significan unos y otros no están claramente definidos.

⁴⁹ En castellano, “Desarrollo guiado por el comportamiento”.

⁵⁰ En castellano, “Desarrollo guiado por pruebas de aceptación”.

⁵¹ En castellano, “Desarrollo guiado por pruebas de requerimientos”. Ver [Mugridge 2008].

⁵² En castellano, “especificación mediante ejemplos”. Véase [Marick 2002], [Marick 2003], [Adzic 2009].

Arquitectura y diseño en emergencia

Introducción

Cuando construimos software, todo lo que hacemos es diseño, desde que decidimos las características a incluir en el producto hasta que se instala en el entorno en que será usado (podríamos incluso decir que la instalación requiere en la mayoría de los casos de una cuidadosa definición previa que también es diseño), pasando por la escritura del código. Contradicatoriamente, en la mayoría de las estructuras organizadas de conocimiento sobre ingeniería de software, desde el CMMI hasta los planes de estudio de las universidades, el diseño de software como actividad no recibe mucha atención. Basta revisar cuántas materias se le dedican en una carrera de ingeniería en sistemas o computación, y cuántas áreas de proceso tiene en el CMMI para Desarrollo. La respuesta es una: Solución Técnica; dos, si contamos Integración del Producto.

Afortunadamente, el diseño es uno de los focos principales de nuestra actividad para una gran parte de la comunidad de ingeniería de software, en particular la más involucrada con la práctica de la construcción de software y no con la perspectiva de proceso. Basta pensar en el impacto que tuvo la publicación en 1994 del libro *Design Patterns* [Gamma 1994], o el fuerte contenido de diseño en las prácticas ágiles, desde el desarrollo guiado por pruebas y la refactorización, o el foco en el diseño que acompaña a todo el movimiento de Orientación a Objetos.

Este capítulo presenta la práctica de diseño de software desde nuestra experiencia, entendida como una actividad creativa y productiva, inserta en un contexto real con restricciones y objetivos, pero siempre dependiente de las personas, de sus experiencias pasadas de realizar diseños similares (o muy diferentes) y de la forma en que piensan.

También recorremos algunos de los temas fundamentales del diseño de software, desde patrones y tácticas de diseño y arquitectura hasta las prácticas específicas de la agilidad, como el desarrollo guiado por pruebas.

¿Qué hace a un buen diseño?

Un buen diseño tiene algunas características, fáciles de identificar pero no de lograr en la mayoría de los casos.

- *Correcto*: cumple con resolver el problema.
- *Claro*: el diseño es fácil de comprender, tanto para su uso como para su modificación.
- *Coherente*: el diseño es estructuralmente consistente, sin contradicciones. Además, sus componentes forman parte de un todo armónico, sin divergencias ni elementos desalineados. La coherencia se describe en algunos casos como Integridad Conceptual.

Figura 7.1. Esta bola de bandas elásticas tiene un diseño coherente.



- *Simple*: menos es más, el diseño cumple con resolver el problema con los mínimos elementos posibles⁴³.
- *Elegante*⁴⁴: la apreciación estética del diseño produce una resonancia en el observador que lo acerca al diseño y le permite relacionarse con él de una forma especial, más allá de las características objetivas del mismo.
- *Preciso*: el diseño no trata ningún aspecto que no responda concretamente a una necesidad.
- *Robusto*: el diseño resiste cambios menores en el problema inicial⁴⁵.
- *Continuo (Suave)*⁴⁶: pequeños cambios en el problema implican pequeños cambios en el diseño [Meyer 1985].

Tras un diseño emergente

Una de las características deseables de cualquier diseño es la simplicidad, que le aporta elegancia y lo hace fácil de comprender por otros. En el caso particular del software, dado que los problemas tienden a ser complejos y difíciles de comprender, crear diseños simples ayuda a no agregar complejidad al problema cuando pensamos la solución.

En la mayoría de los casos, sin embargo, es muy difícil lograr diseños simples cuando los problemas son complejos. Concretamente,

tienden a ocurrir algunas de las siguientes cosas:

- La solución simple que encontramos no cumple con algunos aspectos del problema, pero sí con su mayoría, lo que hace necesario agregar parches al diseño para completarlo. El típico parche es un bloque condicional que solo sirve para un caso excepcional.
- El diseño agrega capacidades que no son estrictamente necesarias. Por ejemplo, agregamos un método a una clase solo porque parece fácil hacerlo (pero no tenemos en cuenta el costo de mantenimiento).
- Una solución resulta atractiva por su propia complejidad. Por ejemplo, asignar números primos para identificar opciones no excluyentes y luego calcular el producto de todos los factores primos para las opciones seleccionadas, para que al dividir ese producto por cada uno de sus factores primos pueda determinarse que están seleccionadas.

En todos estos casos, el resultado general es negativo, porque el diseño que logramos resulta peor que otras alternativas. Existen varias formas de lograr diseños simples, tratando las situaciones descriptas.

- Para ayudar a tener en cuenta todos los aspectos de un problema conviene tener múltiples personas interactuando para hacer el diseño. Así, las múltiples perspectivas aportan para cubrir todas las facetas del problema.

• Para evitar agregar capacidades, es necesario atar cada decisión de diseño a alguno de los aspectos del problema a resolver. Esta vinculación debe hacerse explícitamente, a priori y como parte del proceso de diseño, aunque también puede complementarse con una revisión posterior.

• Aprender a desapegarnos de nuestros diseños⁴⁷ y valorar su simplicidad es algo que logramos con el tiempo, a través de múltiples experiencias propias y de ver el trabajo de otros, que cuando es simple pero cumple perfectamente con resolver el problema nos produce una impresión especial (nos afecta, a partir de que algo en su estructura resuena con nosotros [Austin 2012]).

• Hacer generalizaciones solo por inducción, cuando hay suficientes ejemplos para justificar la generalización a partir de los casos particulares. Por ejemplo, crear una jerarquía de clases con herencia solo cuando hemos construido dos o tres clases que aparecen como buenas candidatas a heredar de una clase padre común.

La primera y la última de estas recomendaciones son técnicas concretas para dejar emerger un diseño, como opuesto a definirlo completamente de antemano. La emergencia es una característica de ciertos sistemas complejos; en el primer ejemplo, un grupo de personas colaborando para producir un diseño y, en el segundo, un conjunto de aspectos del problema tratados iterativamente para definir una solución.

El diseño emergente requiere ciertas condiciones:

1. Diferir hasta el último momento prudente las decisiones: si decidimos antes, es poco probable que contemos con toda la información necesaria. ¿Cómo identificamos el último momento prudente? Cuando estamos a punto de perder alguna de nuestras alternativas de diseño debemos decidir; de otra manera, si se pierden las alternativas y nos quedamos sin opciones, lo que hacemos no es diseño.

2. Controlar dejando hacer: debemos aceptar que no todo el diseño está bajo nuestro exclusivo control para poder colaborar con otros.

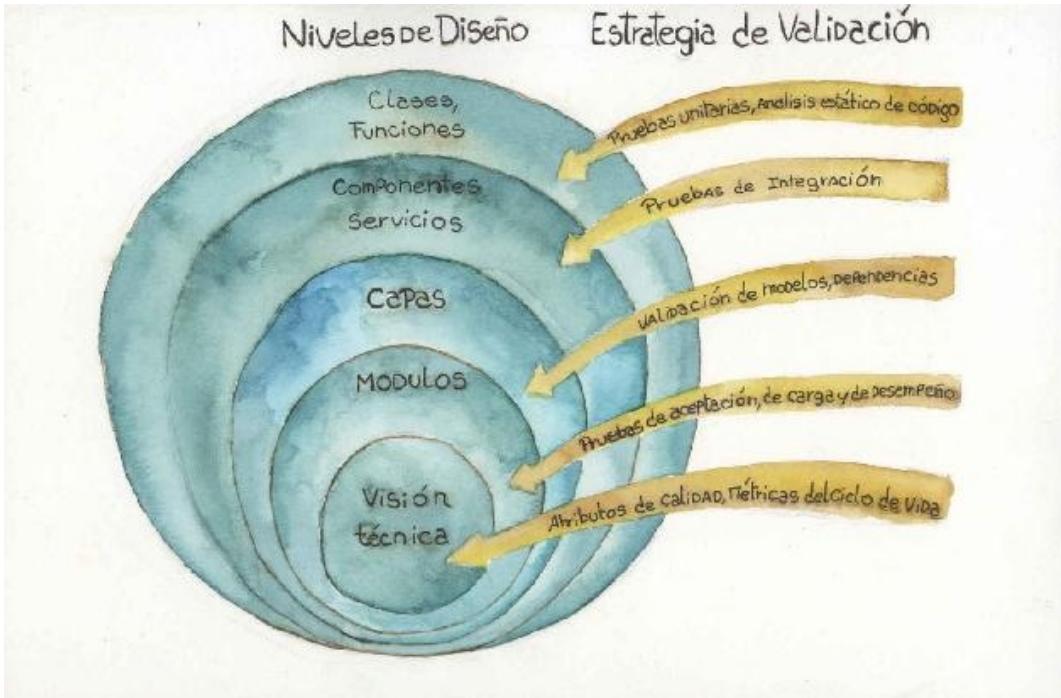
3. No agregar cosas hasta no estar seguros de que son necesarias, aunque parezcan buenas (como en el ejemplo de la herencia, tendemos a querer usar las técnicas que aprendimos, aun cuando no son necesarias o incluso apropiadas).

Como planteamos, desapegarnos de nuestros diseños es un paso fundamental, porque nos permite aceptar que el diseño fluya desde nosotros pero no que esté totalmente bajo nuestro control.

Arquitectura y diseño en el ciclo de vida ágil

La arquitectura de software de un sistema es una visión técnica, compartida por los involucrados, sobre el diseño del mismo. Vista de esta forma, la arquitectura tiene un sesgo estratégico, pero no es necesariamente una representación completa de esa solución, sino más bien un idea de la dirección en la que el equipo piensa dirigirse en la búsqueda de la solución, y una guía tentativa para futuras decisiones de diseño. En ese marco, las futuras decisiones se tomarán de manera que estén alineadas con la arquitectura, o bien presentarán el desafío al equipo de redefinir la arquitectura para ajustarla a los nuevos descubrimientos. Estos pueden ser cambios en el contexto en la propia visión del equipo sobre la solución. Como ejemplo del primer caso, pueden mencionarse cambios en restricciones, requerimientos o actores. Ejemplos del segundo caso son las inconsistencias o incoherencias encontradas en el diseño al tomar una nueva decisión que degradan la calidad del mismo.

Figura 7.2. Niveles de organización y validación en una arquitectura⁴⁸.



Una de las características de un proceso ágil es la validación iterativa temprana del producto. A nivel de arquitectura y diseño, la figura 7.2 muestra varios niveles de descomposición del diseño y estrategias de validación para cada uno. Como se ve en la figura, cada uno de los niveles está alineado con la visión técnica y con los demás niveles. En palabras de Frederick P. Brooks Jr., “Esto significa que en cada nodo de un árbol de decisiones, uno no enfrenta una simple elección entre múltiples opciones de diseño, sino una elección entre múltiples diseños completos tentativos” [Brooks 2010]. Esta idea integradora de la arquitectura no es contradictoria con otras definiciones más tradicionales. Algunas que nos resultan útiles son:

“la estructura o estructuras del sistema, comprendiendo elementos de software, las propiedades externamente visibles de esos elementos, y las relaciones entre ellos” [Bass 1997]⁴⁹.

Es el conjunto de decisiones de diseño que es muy importante tomar correctamente (Ralph Johnson)⁵⁰.

Es el conjunto de decisiones de diseño que es difícil cambiar [Fowler 2003].

En todas está presente la idea de su importancia estratégica, aunque cada una aporta una mirada propia.

Tácticas y patrones de diseño

Un ejemplo muy interesante de diseño emergente, propuesto por el grupo de arquitectura del Instituto de Ingeniería de Software (SEI) de la Universidad Carnegie Mellon, es el de seleccionar tácticas de diseño apropiadas para los atributos de calidad antes de seleccionar patrones de arquitectura⁵¹. La lógica es que las tácticas permiten una mayor granularidad en la selección de elementos de diseño que apliquen a un conjunto de requerimientos, y que el patrón de arquitectura (como superestructura) se selecciona al final de manera que le dé marco a esas tácticas o patrones de diseño. Este ejemplo es interesante porque proviene de la comunidad de arquitectura tradicional y promueve un proceso emergente de abajo hacia arriba, contrario a las aproximaciones tradicionales que progresan desde las grandes decisiones hacia el diseño detallado.

Desencuentros entre la agilidad y la arquitectura

Hay algunas diferencias en las comunidades ágil y de arquitectura sobre la aplicación de los principios y prácticas ágiles a la arquitectura de software. La mirada más tradicional tiende a ser más analítica y considera que el diseño se puede planificar en detalle al comienzo, mientras que la mirada ágil tiende a ver el diseño como un proceso empírico.

La mayoría de estas diferencias puede agruparse alrededor de unos pocos temas:

- *Diseño inicial grande*: algunos miembros de la comunidad ágil consideran a la arquitectura un caso de “diseño inicial grande” (*big design up-front*).
- *Planificación y estrategia técnica*: en la comunidad de arquitectura hay quienes consideran que la agilidad no tiene lugar para la planificación y estrategia técnica que puede implicar la arquitectura.
- *Documentación*: están difundidas las creencias erróneas de que la agilidad implica no documentar y que la arquitectura implica siempre documentar⁵².
- *Tamaño*: muchas personas creen que los principios y prácticas ágiles solo son aplicables en proyectos o productos pequeños, y algunas piensan lo contrario sobre la arquitectura.

Asumiendo un proceso iterativo por naturaleza, independiente de la agilidad, no es razonable considerar a la arquitectura un trabajo que ocurre solo en las primeras etapas del proceso, porque eso indicaría que todas las decisiones importantes ya se han tomado y, por lo tanto, que toda la información significativa ya estaba disponible; y, en ese caso, no tiene sentido pensar en que el proceso sea iterativo.

Los autores creemos que la arquitectura sirve para organizar no solo el producto sino también los equipos, que pueden o no seguir la estructura del producto, en proyectos grandes. Además, creemos que las prácticas ágiles son claramente aplicables en grandes proyectos, puesto que estos siempre necesitan subdividir el trabajo en equipos más pequeños⁵³.

Más allá de estos puntos, que en casi todos los casos son diferencias de perspectiva y no problemas reales de aplicación, las comunidades ágil y de arquitectura tienen también varios puntos en común, en particular conceptos como la excelencia técnica y la

deuda técnica sirven como punto de contacto⁵⁴.

Deuda técnica: Término acuñado por Ward Cunningham [Cunningham 1992] para describir la degeneración progresiva de un diseño por no ser refinado. La metáfora de deuda tiene sentido porque cada modificación del código cuyo diseño es pobre implica un costo mayor (los intereses), que aumenta hasta que el producto no puede evolucionar y debe ser descartado.

Coherencia e integridad conceptual

Uno de los atributos más importantes de un diseño, tanto desde el punto de vista práctico como desde el estético, es la coherencia. Devin y Austin definen la coherencia como “la característica de un diseño de tener una trama bien construida”. En otros contextos, en general, al hablar de arquitectura de software, se utiliza la expresión “Integridad conceptual”. Ambos nos hablan de una característica esquiva pero difícil de ignorar cuando está presente, e imposible de fingir. Cuando experimentamos un diseño, tanto como creadores iniciales como usuarios, es fácil percibir si ese diseño es bueno o no. Esa percepción es una experiencia muy personal y que tiende a afectarnos, ya sea positiva o negativamente, por eso la metáfora de los olores resulta tan apropiada. Más allá de consideraciones analíticas como las que proponemos en las secciones anteriores, evaluar o crear un diseño coherente produce una resonancia particular en las personas involucradas:

“La gente se relaciona con esa ‘forma’ emergente y experimenta, por lo general no del todo conscientemente, su totalidad unificada. Las sensaciones que experimentamos cuando nos encontramos con una cosa especial surgen de la experiencia de esa forma, muchas veces a partir de la sensación de sorpresa que la gente siente cuando reconocen que la forma de una cosa emergente es mejor (más atractiva o más útil, digamos) que lo esperado o familiar”⁵⁵ [Devin 2012], pág. 14.

Esta resonancia, aunque es una función estética, es muy importante a la hora de motivar a un equipo a usar un diseño y promover la alineación de nuevas decisiones con las anteriores. Mantener un diseño coherente a lo largo del tiempo, en particular a medida que el equipo cambia, es uno de los grandes desafíos de la organización.

Modelar o no modelar

Una confusión común relacionada con el diseño de software es considerar que diseñar es crear modelos, en general, diagramas, y a partir de ellos pasar a la codificación. Una definición de diseño que puede servir es que diseñar es decidir entre alternativas, y plasmar luego esa decisión. En esos términos, un modelo puede servir para plasmar esa decisión, pero no es la única manera. Por ejemplo, la representación en el código de una decisión de diseño es una forma válida, que tiene la ventaja de ser ejecutable directamente y no necesitar traducción. La desventaja, muy fuerte en lenguajes de bajo nivel, es la cantidad de detalles que es necesario plasmar en el código con el consiguiente ofuscamiento de la decisión de diseño involucrada. El uso de lenguajes de alto nivel orientados a objetos dinámicos facilitan la representación de decisiones de diseño en el código, aunque todavía están lejos de permitir representarlas todas.

Como siempre, el camino ágil debe ser el camino del medio⁵⁶, ni el extremo de plasmar todas las decisiones en documentos, ni el otro de creer que el código puede mantener toda la información necesaria. Un buen ejemplo de esto último son las razones por las cuales se tomó una decisión, que en el mejor de los casos pueden representarse en comentarios en el código.

Técnicas de diseño ágil

En esta sección describimos algunas de las técnicas de diseño típicas de los métodos ágiles.

Desarrollo guiado por pruebas

El desarrollo guiado por pruebas es una técnica de diseño definida por Kent Beck a partir de su experiencia en desarrollo con lenguajes dinámicos como Smalltalk.

Esta técnica, a veces identificada con la idea de Test-First (probar primero), pero mucho más comprensiva, propone hacer crecer el código orgánicamente a partir de pruebas unitarias⁵⁷ que fallan inicialmente y promueven la escritura del código necesario para que esas pruebas pasen.

El ciclo de desarrollo guiado por pruebas es:

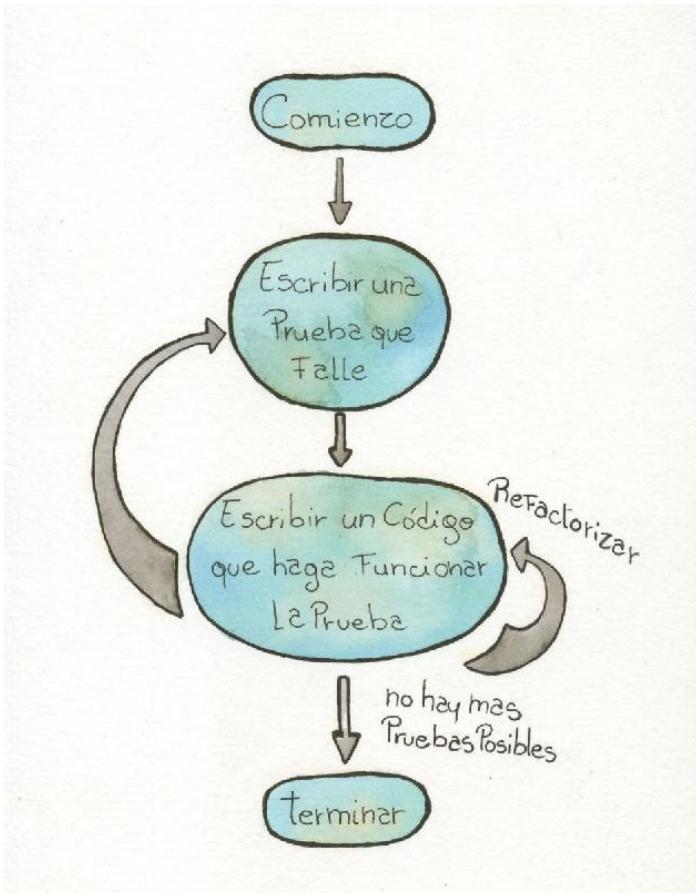
1. Escribir una prueba que falle, en código.
2. Escribir el mínimo código necesario para que esa prueba pase.
3. Si emerge la necesidad de emprolijar el código, modificarlo sin cambiar su comportamiento, de manera que las pruebas sigan pasando. Este paso es conocido como refactorización.
4. Repetir el proceso desde el punto 1, refinando la prueba o agregando una nueva prueba.

El ciclo es claramente iterativo y por incrementos, tanto en las pruebas como en el código funcional, con cada paso correspondiendo a un pequeño cambio en el código o en las pruebas. Requiere facilidad para correrlas luego de cada paso⁵⁸.

Una de las objeciones o confusiones más comunes está en el tamaño apropiado de cada paso (refinamiento o incremento). Una heurística para este caso es agrandar el tamaño si siguiendo el proceso no tenemos sorpresas (es decir, siempre que las pruebas pasen o fallen cuando lo esperamos), y disminuirlo cuando tengamos sorpresas.

Algunas ventajas de hacer desarrollo guiado por pruebas son:

Figura 7.3. Diagrama de estados del proceso de desarrollo guiado por pruebas.



- Como escribimos las pruebas al principio, reducimos el riesgo de no tener pruebas porque se acabó el tiempo.
- Como escribimos pruebas que inicialmente fallan, sabemos que las pruebas son sensibles al código que escribimos⁵⁹ o, dicho de otra forma, que realmente evalúan el código bajo prueba.
- Poder ejecutar fácilmente pruebas automatizadas después de cada cambio nos da el coraje necesario para modificar el código sin temor a una regresión.
- Promueve código simple, modular y con bajo acoplamiento, porque si tiene muchas dependencias es muy difícil escribir pruebas unitarias.
- Las pruebas están escritas en el mismo lenguaje que el código, facilitando la especificación de pruebas por parte de los desarrolladores.
- Las pruebas se escriben como parte de la actividad de desarrollo, promoviendo que los desarrolladores requieran en forma temprana información y mejor aún, trabajo de a pares con testers, analistas, etc., favoreciendo la integración multifuncional del equipo.
- Está ampliamente soportado por herramientas y frameworks.
- Las pruebas escritas documentan los requerimientos implementados.

Cualquier defecto se trata revisando primero las pruebas. En concreto, esto permite reproducir errores en otros entornos, detectar pruebas faltantes o limitadas, y simplifica la navegación del código, porque iniciamos nuestra búsqueda de un defecto desde las pruebas, y no desde la estructura modular.

Las pruebas disminuyen el costo de los cambios futuros y ofrecen análisis de impacto de los cambios.

Algunas desventajas del desarrollo guiado por pruebas son:

- Requiere un cambio profundo en la práctica de los desarrolladores. Lo más difícil, muchas veces, es el cambio de mentalidad.
- Requiere una inversión inicial en infraestructura de pruebas, que puede ser muy básica o muy compleja dependiendo de la tecnología y del proyecto⁶⁰.
- Requiere invertir un esfuerzo en la construcción de las pruebas que no es negociable. Aunque pueda parecer alto, el esfuerzo involucrado tiende a ser menor que lo esperado una vez que los equipos conocen bien la técnica, pero lo que sí es inevitable es que con esta práctica, el esfuerzo de prueba no es sacrificable, con lo cual se frustran opciones perniciosas como costos eliminando las pruebas en etapas posteriores a la construcción.

Figura 7.4. Ejemplo de prueba unitaria incipiente en Java y JUnit que podría iniciar un proceso de desarrollo guiado por pruebas para una jerarquía de cuentas bancarias.

```

import static org.junit.Assert.*;
import org.junit.Test;
public class TestCuenta {
    @Test
    public void deberiaCrearConSaldo0() { Cuenta cuenta = new Cuenta(); assertEquals( 0, cuenta.saldo() );
    }
}

```

}

Refactorización

Como vimos, el proceso de desarrollo guiado por pruebas propone una actividad de mejora del código llamada refactorización. El propósito principal es mejorar la calidad interna del código y el diseño en forma independiente de la funcionalidad implementada, que está garantizada por las pruebas. El proceso propone que la necesidad para refactorizar emerge de las modificaciones sucesivas del proceso y, por lo tanto, no está sesgada por una definición a priori. En general, como vimos en la sección de diseño emergente, esto difiere complejizar el diseño hasta que es claramente necesario (porque observamos la degradación del código, no la pronosticamos).

El concepto de refactorización ha tenido profundo impacto en la comunidad de desarrollo de software: existen múltiples libros que tratan específicamente del tema⁶¹, e incluso ciertas prácticas como el uso de patrones de diseño, uno de los más significativos avances de la comunidad de desarrollo de software de las últimas décadas, está cada vez más asociada a la refactorización.⁶²

Las operaciones de refactorización están formalizadas en la bibliografía y múltiples herramientas de desarrollo. Algunos ejemplos concretos son:

Extraer método: crear un método a partir de un conjunto de líneas de código, para aumentar la claridad y reducir la redundancia.

Extraer clase madre: separar una clase en dos, una madre y otra hija. Permite crear una jerarquía a partir de una clase cuando existe comportamiento variable.

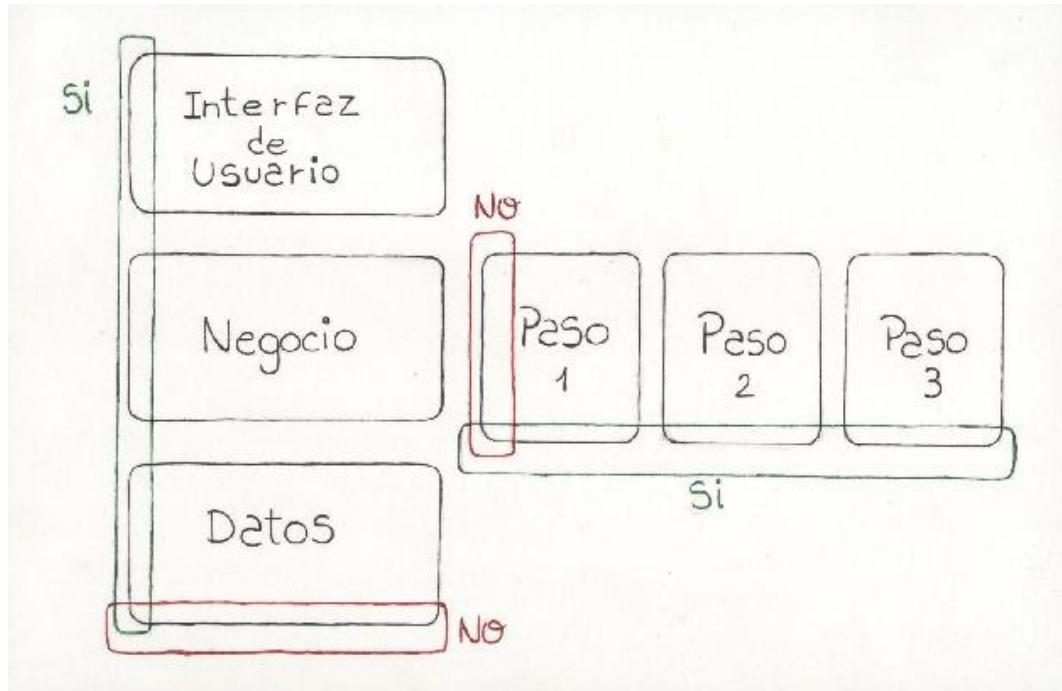
El concepto de refactorización, más allá del término, hace mucho tiempo que está presente en el mundo del software, como la idea de que el mantenimiento de un sistema debe implicar una reinversión periódica en mejoras internas de la estructura del sistema para permitir su evolución futura. En [Jacobson 1992] los autores proponen como analogía el concepto de entropía, con la idea general de que modificar un sistema sin reorganizarlo implica inevitablemente aumentar su complejidad.

Olor (Smell): Término acuñado por Kent Beck y Martin Fowler para describir cuando el código se ha degradado y emerge la necesidad de una refactorización. Ejemplos son: métodos demasiado largos, redundancia, nombres de variables confusos, etc.

Esqueleto caminante⁶³/desarrollo orgánico

Esta técnica propone crear desde el principio un incremento del sistema que contenga en forma embrionaria todos los elementos futuros, o por lo menos sugiera la estructura final del sistema. La idea es que los componentes principales de la arquitectura estén presentes, normalmente en forma de cáscaras vacías, que permiten la integración de punta a punta del sistema y probar el concepto de la arquitectura. A partir de ahí, cada incremento o refinamiento hace crecer los componentes, pero siempre manteniendo funcional la integración del sistema como un todo. Esto permite evitar los riesgos de una integración tardía, como discutiremos en el capítulo “Integrando el producto al instante”. Además sirve para plasmar en forma temprana la estructura del sistema y tener una validación de la visión técnica.

Figura 7.4. Distintos cortes de una arquitectura de 3 Capas, los que dicen sí, promueven el desarrollo orgánico.



Diseño colaborativo

En el capítulo “Quién manda a quién” vamos a desarrollar en detalle el concepto de colaboración, por ahora presentamos en esta sección algunas de las dinámicas propias de la agilidad para la toma de decisiones de diseño.

Sesiones de diseño colaborativo: el equipo de desarrollo (no solo el arquitecto) participan de sesiones de intercambio de ideas y toman decisiones en conjunto. Es fundamental contar con un pizarrón para poder plasmar fácilmente (y también borrar) las ideas propuestas. El entorno debe ser abierto (por ejemplo, es mejor si evitamos empezar cada frase con la palabra “NO”), escuchar las ideas de todos y el arquitecto en su rol de facilitar la reunión y aportar su experiencia y visión en pos de los objetivos del negocio.

Toma de decisiones: en muchos entornos se considera que el arquitecto debe tomar todas las decisiones de arquitectura. En cambio, en un equipo ágil buscamos que el conjunto de las habilidades y experiencias hagan emerger un diseño mejor en forma colaborativa,

por lo tanto ese rol puede estar limitado a guiar al resto del equipo y ayudarles a aprender. Tobías Mayer⁶⁴ destaca la diferencia entre consenso y consentimiento. La autoorganización no significa que el diseño sea algo malo ni que deba frenar a un equipo que no logra tomar una decisión. La idea de consentimiento es que los miembros del equipo, cuando no logran consenso más allá de un cierto tiempo limitado determinado de antemano, pueden aceptar las ideas de otros, aún si creen que la propia es mejor. Una alternativa límite es que el arquitecto resuelva en caso de desacuerdos que no pueden ser reconciliados.

Actitudes de buen diseñador

Una mirada ágil no puede dejar de lado las actitudes y aptitudes del diseñador como persona. Estas son algunas de las cualidades que hacen a un buen diseñador:

- Diseña para gente (usuarios, desarrolladores, administradores, operadores, etc.).
- Es creativo.
- Ofrece razones, explica y escucha a los demás.
- Hace participar a otros.
- Promueve y soporta un lenguaje compartido del diseño.
- Diseña a partir de su experiencia pero está abierto a las sorpresas.
- No está aferrado a sus propias ideas y diseños.

En resumen

En este capítulo propusimos algunas de las cuestiones principales del diseño y arquitectura de software en entornos ágiles. Recorrimos los puntos de contacto y desencuentro, que no son tantos ni tan profundos como puede parecer, entre las comunidades ágil y de arquitectura. También repasamos las características de un buen diseño, en particular la simplicidad, y de un buen diseñador. Finalmente vimos algunas de las principales técnicas propias de la agilidad, como el desarrollo guiado por pruebas y el esqueleto caminante. Esperamos que estas ideas los motiven a profundizar en los temas tratados y a intentar aprender las prácticas a través de aplicarlas a situaciones reales.

⁴³ Una alternativa que proponen Devin y Austin es que sea tan complejo como sea posible sin dejar de ser accesible [Devin 2012].

⁴⁴ Martin Fowler y Kent Beck acuñaron la expresión “olor” para describir un mal diseño que ofende nuestras sensibilidades, véase el recuadro en p.101.

⁴⁵ Como opuesto a frágil.

⁴⁶ El término suave es nuestro.

⁴⁷ Alan Cyment enfatiza la humildad intelectual como parte del espíritu de la agilidad [Cyment 2012].

⁴⁸ Adaptado de [Fontdevila 2010].

⁴⁹ La traducción es nuestra, el original en inglés es “the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them”.

⁵⁰ Correspondencia citada por [Fowler 2003].

⁵¹ Véase [Bass 2003], capítulo 5, esta referencia es específica a la segunda edición de 2003.

⁵² En palabras de Martín Salías, uno de los referentes de la comunidad ágil en América Latina, “La arquitectura es como el diseño, podés no ocuparte de ella pero siempre vas a tener una, que en ese caso probablemente no sea muy buena”.

⁵³ Véase el capítulo Todo muy lindo pero...

⁵⁴ Véanse ejemplos de la iniciativa de arquitectura y agilidad del SEI, Instituto de Ingeniería de Software, en [SEI 2013].

⁵⁵ La traducción es nuestra, el original en inglés es “*People engage with that emerging ‘shape’ and they experience, usually not completely consciously, its unified wholeness. The feelings we experience when we encounter a special thing arise from an experience of form, often from the sense of surprise people feel when they recognize that the form of an emergent thing is better (more attractive or more useful, say) than the expected or familiar*”.

⁵⁶ Expresión propia del budismo utilizada por Alan Cyment durante sus cursos de Scrum para explicar cómo la agilidad reconcilia pragmatismo a corto plazo con idealismo en el mediano y largo plazo.

⁵⁷ Como vimos en el capítulo “Empezando por la aceptación”, en realidad hay otras técnicas como ATDD que son guiadas por pruebas distintas de las pruebas unitarias.

⁵⁸ Aun sin entornos integrados de desarrollo, algunas herramientas soportan este proceso mediante la ejecución periódica frecuente de las pruebas, como en el caso de JavaScript.

⁵⁹ Al cambiar el código que hacía que una prueba falle y ver que luego de ese cambio la prueba pasa, asumimos que es ese cambio lo que produjo el efecto observado y, por lo tanto, que estamos probando el código (en realidad, esto es una falacia conocida como post hoc ergo propter hoc. Pero en la mayoría de los casos es aplicable, puesto que las pruebas no deberían depender de otros aspectos volátiles como, por ejemplo, la hora a la que fueron ejecutadas).

⁶⁰ Por ejemplo, algunos lenguajes de alto nivel y sobre todo algunos frameworks proveen herramientas muy accesibles de pruebas unitarias, mientras que en lenguajes como C++ tienden a ser más complejas. En general, existen implementaciones muy simples de bases de datos en memoria y frameworks de soporte (por ejemplo mocking) que permiten realizar implementaciones muy poderosas con simpleza. Sin embargo, si esta inversión no se realiza al principio del proyecto, los costos y la complejidad tienden a aumentar muy rápidamente.

⁶¹ Véase [Fowler 1999] y [Kerievsky 2005].

⁶² Existe un libro especializado en este proceso, véase [Kerievsky 2005].

⁶³ Walking skeleton, término acuñado por Alistair Cockburn, véase [Cockburn 2004].

⁶⁴ [Mayer 2013], capítulo “Seeking Consent”, pág. 144.

Probar, probar, probar

La mayor parte de los proyectos comienzan pequeños. Por eso nos parece que los podemos gestionar sin preocuparnos demasiado. Unas pocas máquinas en red, con un entorno de desarrollo que permita compilar y ejecutar algunas pruebas unitarias, un repositorio de archivos no muy sofisticado, suelen bastar. Las pruebas de integración integran pocas funcionalidades y módulos, así que pueden hacerse con relativa facilidad y sin demasiadas complicaciones. Y cuando necesitamos desplegar el software en el equipo del cliente, lo hacemos con métodos artesanales.

El problema comienza cuando el sistema que hemos construido se empieza a parecer a las cañas de la Patagonia: lo que había nacido como una pequeña hierba, se ramifica en verdaderos bosques que, con el tiempo, alcanzan varios metros de altura. Con el software pasa algo parecido: cada tanto se le agregan más funcionalidades, a la vez que crece en complejidad, ramificándose en direcciones no previstas al comienzo.

En estos casos, todas las herramientas y técnicas que veníamos usando empiezan a ser insuficientes. Nos empezamos a dar cuenta cuando un despliegue que creímos que nos iba a llevar media hora, nos conduce a largas sesiones de instalación debido a problemas en el ambiente del cliente. O cuando el último cambio solicitado por este, en vez de hacerlo en la semana que creímos que nos iba a llevar, consume meses corrigiendo errores que no terminan de aparecer.

Hay muchas técnicas que pueden ayudar en estos casos. Los métodos ágiles han buscado apoyarse en la automatización. En este capítulo nos vamos a centrar en la automatización de las pruebas.

Por qué automatizar

Desde el paleolítico, por lo que sabemos, cuando el ser humano necesitó resolver problemas más complejos, recurrió a técnicas y herramientas que lo ayudaran. Con el paso de los siglos y milenios, también inventó máquinas que lo ayudaran, y no pocos enfoques metodológicos. Cada vez más, las máquinas ayudaron al ser humano a hacer tareas repetitivas o de fuerza bruta.

Con la aparición de las computadoras, las fantasías fueron mucho más allá, y muchos pretendieron que las máquinas pudieran suplir a los humanos en todo, incluso en tareas intelectuales. Esto se ha logrado solo en una mínima parte; sin embargo, cada vez resulta más atractivo usar las capacidades de las computadoras para realizar tareas que se pueden automatizar, logrando mayor velocidad, mayor fuerza de cálculo o incluso para minimizar los problemas causados por algunas características humanas, tales como el cansancio, la pérdida de concentración y la falta de motivación por realizar tareas de las que se presume el resultado.

En este contexto, en el desarrollo de software se viene intentando automatizar desde hace décadas, tal vez por el mismo hecho de trabajar continuamente con computadoras. Al principio, y en línea con las expectativas puestas en las máquinas inteligentes, se pensaba en reemplazar totalmente al programador humano, haciendo una analogía entre los obreros de las industrias manufactureras y los programadores. Debido a que la Inteligencia Artificial no mostraba los resultados esperados, o al menos mostraba que se iba a tardar mucho en alcanzarlos, se buscaron alternativas aparentemente más realistas: ¿no se podrá hacer que las tareas más decididamente humanas, como el análisis y el diseño, las hagan seres humanos, mientras que las que requieren menos trabajo intelectual, como la programación pura, la puedan resolver las máquinas? Ese modo de pensar llevó a las herramientas CASE en los años 1980 y 1990, que suponían que, partiendo de modelos suficientemente detallados del dominio y del diseño, la programación se podría dejar solo a las máquinas.

Hoy sabemos que esta visión ha fracasado, al menos en sus pretensiones de máxima. Tal vez el error fue pensar que la programación era un trabajo mecánico, o que convertirla en un trabajo mecánico sería una ventaja. Pero claramente fue la segunda desilusión, luego de la exagerada apuesta por la Inteligencia Artificial⁶⁵.

Mientras tanto, muchos programadores descubrieron que había estructuras de código que se repetían y desarrollaron sus propios generadores de código para altas, bajas, modificaciones y consultas a tablas simples de bases de datos. También se crearon herramientas para grabar la ejecución de las pruebas, de modo tal de poder repetirlas después, por ejemplo como pruebas de regresión. En casi todos los casos se buscaba ahorrar tiempo, ya que las máquinas pueden hacer lo mismo que nosotros en un tiempo mucho menor. También se pretendía bajar la tasa de errores que provocaba hacer lo mismo una y otra vez.

En esta forma de automatización es en la que se trabaja hoy. Las otras, al menos por el momento, han evidenciado escaso éxito. Tal vez el tiempo cambie las cosas, pero por el momento estamos aspirando a automatizar algunas tareas que son indudablemente mejor hechas por las máquinas. En esa línea vienen trabajando los métodos ágiles.

Quizá sean las menos ambiciosas de las formas de automatización, pero aún así muchos no las practican.

TDD y automatización

En el capítulo “Empezando por la aceptación” decíamos que la definición de TDD no era solo escribir pruebas antes del código que las mismas deben probar, sino también que dichas pruebas hayan sido automatizadas. Con automatizadas, TDD se refiere a que estén escritas en código o de alguna otra manera que permita ejecutarlas con una computadora.

En TDD, el desarrollador comienza escribiendo una prueba automática que defina un cambio o nueva funcionalidad, luego escribe el código que haga que esa prueba pase (hasta ese momento debía fallar) y luego refactoriza⁶⁶ para mejorar la calidad del código.

El diagrama de la figura 8.1 muestra el flujo típico de TDD.

Figura 8.1. Ciclo de TDD.



“Ver tabla comparativa en <http://unamiradaagil.com/tablas/>

También hemos hablado de variantes de TDD, según el tipo de pruebas que están involucradas. Como resumen, la tabla en página anterior muestra una buena clasificación de tipos de TDD y tipos de pruebas (basada en la planteada por Freeman y Pryce en [Freeman 2010]).

Ventajas de la automatización

Las ventajas de automatizar las pruebas son muchas. Entre ellas, destacan:

- Nos independizamos del factor humano, con su carga de subjetividad y variabilidad en el tiempo⁶⁷.
- Facilita la repetición de las mismas pruebas, con un costo ínfimo comparado con las pruebas realizadas por una persona. Esto es aplicable a regresiones, debugging y errores provenientes del sistema ya en producción.
- Da una red de seguridad de rápida ejecución para correr luego de cada refactorización, para asegurarnos de que el comportamiento observable no ha cambiado.
- Las pruebas evolucionan y crecen junto con el código, de un modo incremental que permite ejecutar conjuntos cada vez mayores a un costo que solo crece linealmente.

Además, en el caso de las pruebas de programador, en código, automatizarlas sirve para:

- Documentar el uso esperado de los módulos.
- Indicar con menor ambigüedad lo que se espera de los módulos.

Qué automatizar

No hay una taxonomía de tipos de pruebas que esté suficientemente aceptada por la comunidad de desarrollo de software. Por eso, vamos a realizar una, que es una simplificación de la clasificación tridimensional de Meszaros [Meszaros 2007]:

- Pruebas de cliente: especifican y validan las expectativas del cliente, además de servir como criterios de aceptación.
- Pruebas de componentes: definen el diseño de alto nivel del sistema y explicitan la intención del arquitecto; a veces se las llama pruebas de integración técnicas.
- Pruebas de unidad: definen el diseño a nivel del código y explicitan la intención del desarrollador.
- Pruebas de usabilidad: buscan determinar si el sistema es cómodo de usar (y aprender a usar) para sus usuarios.
- Pruebas exploratorias: comprueban si el sistema es consistente desde el punto de vista del usuario, si la apariencia es la esperada, etc.
- Pruebas de propiedades: especifican atributos de calidad, definiendo si el sistema es seguro, escalable, robusto, etc.

Lo interesante de la clasificación de Meszaros es que él propone que algunas de las pruebas se pueden y deben automatizar, mientras que las demás deben hacerse en forma manual.

En efecto, las de cliente, las de componentes, las de unidad y las de propiedades, se pueden y conviene automatizarlas. En cambio, las pruebas de usabilidad y las exploratorias, solo es posible hacerlas en forma manual.

Más allá de estas cuestiones, es deseable realizar algunas pruebas en forma manual, sobre todo porque las pruebas realizadas por humanos van a poder detectar los errores de las pruebas automáticas mal construidas: al fin y al cabo, las computadoras solo prueban lo que les decimos que prueben.

Niveles de pruebas y automatización

En los primeros tiempos, TDD se vio como sinónimo de pruebas unitarias automatizadas realizadas antes de escribir el código.

Hay un problema con esta visión de TDD que solo se refiere a pruebas unitarias, porque omite todas las demás cuestiones que pueden

validarse con pruebas automáticas, y que de hecho estaban previstas en el libro fundacional de XP [Beck 1999]. Asimismo, las herramientas reflejan este sesgo hacia las pruebas de unidad en sus nombres: JUnit, SUnit, NUnit, etc. Y cuando Beck escribe su libro de TDD [Beck 2002] se basa primordialmente en ejemplos de pruebas unitarias.

Pero todas las variantes de TDD que hemos visto son susceptibles de automatización.

Lo que a lo sumo ocurre es que los objetivos de cada tipo de TDD son distintos. Mientras con UTDD solo escribimos especificaciones y pruebas de las intenciones del desarrollador, con los objetivos de definir el diseño y realizar un control de calidad interno, con ATDD y sus técnicas afines escribimos especificaciones y pruebas de los requerimientos del cliente, buscando definir al sistema desde su comportamiento y permitiendo un control de calidad externo.

Automatizando pruebas de unidad

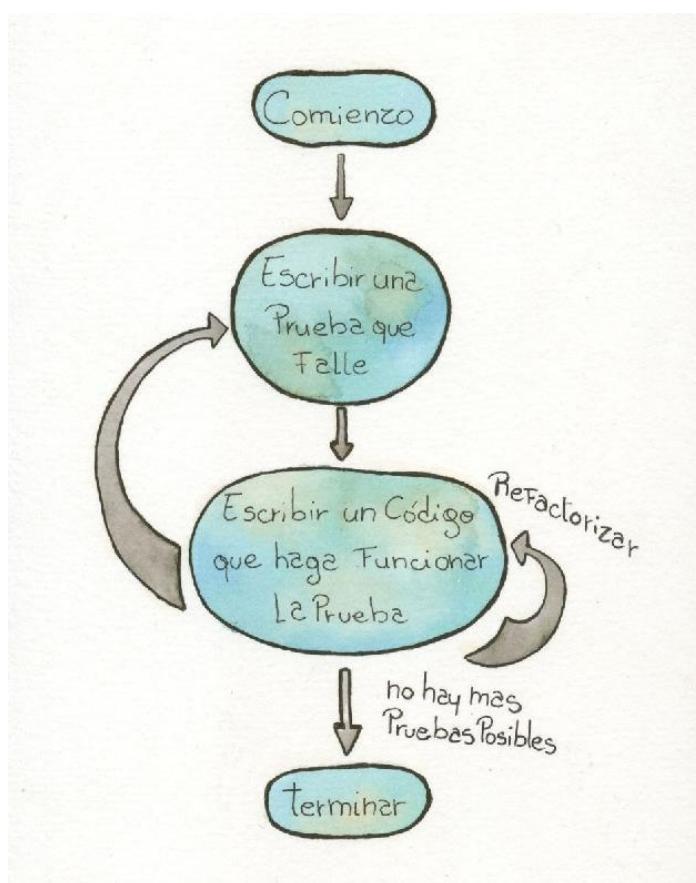
Como decíamos más arriba, en el marco de TDD se empezó automatizando pruebas unitarias. Por ejemplo, se diseñaban clases y métodos⁶⁸ a partir de pruebas que definían lo que esas clases y métodos debían hacer. Estas pruebas las realizan los propios desarrolladores, como un paso previo a la escritura de una porción de código. Por lo mismo, son pruebas muy pequeñas y cohesivas, cada una de ellas probando solo un escenario de uso muy restringido.

Para lograr esto, conviene entonces desarrollar pruebas simples, que cada una tenga una sola razón para fallar, que sea independiente de las demás, que sean rápidas de ejecutar y hacerlas independientes de todos los recursos externos (redes, bases de datos, archivos de configuración). Al hacer los componentes fáciles de probar, el acoplamiento entre los mismos tiende a disminuir.

La idea, entonces, es realizar un ciclo iterativo que vaya tomando de a una prueba por vez, logre que el código la haga pasar, refactorizar de ser necesario y luego escribir la prueba siguiente, como se muestra en la figura 8.2.

Esta forma de trabajo exige programadores disciplinados y dispuestos a escribir pruebas o corregir sus presunciones. Beck [Beck 1999] dice que si una prueba resulta difícil de llevar a código, probablemente estemos ante un problema de diseño; sostiene además que el código altamente cohesivo y escasamente acoplado es fácil de probar en forma automática.

Figura 8.2. TDD con pruebas unitarias.



Hay críticos que dicen que las pruebas llevan tiempo y que los programadores deberían dedicarse solo a escribir código, que es lo que mejor hacen, dejando las pruebas a otros. Sin embargo, como se pierde más tiempo corrigiendo errores que en la primera escritura de código, lo cierto es que conviene tener pruebas automatizadas y correrlas frecuentemente.

Con el éxito de UTDD, pronto fue necesario contar con herramientas que facilitaran las tareas del programador. Es así como surgieron los primeros frameworks de pruebas técnicas automatizadas: SUnit para Smalltalk y JUnit para Java. Los mismos fueron las bases para la construcción de las demás herramientas para los demás lenguajes y plataformas, a las que no sorprendentemente se las llamó xUnit de forma genérica.

Por ejemplo, lo que sigue es código JUnit que corresponde a una clase⁶⁹ de pruebas unitarias para la clase LineaDe3. Esta clase representa una línea de tres casilleros alineados en el juego de Ta-Te-Ti.

```
public class PruebasLineaDe3 {  
  
    private LineaDe3 linea;
```

```

private celda0, celda1, celda2;
private String descripcion = "línea diagonal inversa";
private Juego juego = new Juego (TipoJuego.JugadorVsJugador); private JugadorHumano jugadorX = new
JugadorHumano ('X', juego); private JugadorHumano jugadorO = new JugadorHumano ('O', juego);

@Test
public void estaCompletaDeberiaSerVerdaderoCuandoTresCeldasTienenElMismoJugador () {
    Celda[] celdas = { new Celda (2, 0, jugadorX), new Celda (1, 1, jugadorX), new Celda (0, 2, jugadorX) };
    linea = new LineaDe3( celdas, descripcion ); Assert.assertTrue( linea.estaCompleta() );
}

@Test
public void estaCompletaDeberiaSerFalsoCuandoUnaCeldaNoEstaOcupada () {
    Celda[] celdas = { new Celda (2, 0, jugadorX), new Celda (1, 1, null), new Celda (0, 2, jugadorX) };
    linea = new LineaDe3( celdas, descripcion ); Assert.assertFalse( linea.estaCompleta() );
}

@Test
public void hayTaTeTiDeberiaSerFalsoCuandoLasTresCeldasNoTienenElMismoJugador () {
    Celda[] celdas = { new Celda (2, 0, jugadorX), new Celda (1, 1, jugadorO), new Celda (0, 2, jugadorX) };
    linea = new LineaDe3( celdas, descripcion ); Assert.assertFalse( linea.hayTaTeTi() );
}

@Test
public void hayTaTeTiDeberiaSerVerdaderoCuandoTresCeldasTienenElMismoJugador () {
    Celda[] celdas = { new Celda (2, 0, jugadorX), new Celda (1, 1, jugadorX), new Celda (0, 2, jugadorX) };
    linea = new LineaDe3( celdas, descripcion ); Assert.assertTrue( linea.hayTaTeTi() );
}

@Test public void proximaJugadaPodriaSerTaTeTiDeJugadorDeberiaSerVerdaderoCuendoElMismoJugadorOcupaTresCeldas () {
    Celda[] celdas = { new Celda (2, 0, jugadorX), new Celda (1, 1, jugadorX), new Celda (0, 2, null) };
    linea = new LineaDe3( celdas, descripcion );
    Assert.assertTrue( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorX) ); Assert.assertFalse(
    linea.proximaJugadaPodriaSerTaTeTiDe(jugadorO) );
}

@Test public void celdaFaltanteParaTaTeTiDeberiaSerLaPrimeraCuandoLasOtrasDosEstenOcupadasPorElMismoJugador () {
    Celda primera = new Celda (0, 2, null);
    Celda[] celdas = { primera, new Celda (1, 1, jugadorX), new Celda (0, 2, jugadorX) };
    linea = new LineaDe3( celdas, descripcion );
    Assert.assertSame( primera, linea.celdaFaltanteParaTaTeTi(jugadorX) );
    Assert.assertNotSame( new Celda (1, 1, jugadorX), linea.celdaFaltanteParaTaTeTi(jugadorX) );
    Assert.assertNotSame( new Celda (0, 2, jugadorX), linea.celdaFaltanteParaTaTeTi(jugadorX) );
}

@Test
public void celdaFaltanteParaTaTeTiDeberiaSerLaUltimaCuandoLasOtrasDosEstenOcupadasPorElMismoJugador () { Celda
ultima = new Celda (0, 2, null);
    Celda[] celdas = { new Celda (2, 0, jugadorX), new Celda (1, 1, jugadorX), ultima };
    linea = new LineaDe3( celdas, descripcion );
    Assert.assertSame( ultima, linea.celdaFaltanteParaTaTeTi(jugadorX) );
    Assert.assertNotSame( new Celda (2, 0, jugadorX), linea.celdaFaltanteParaTaTeTi(jugadorX) );
    Assert.assertNotSame( new Celda (1, 1, jugadorX), linea.celdaFaltanteParaTaTeTi(jugadorX) );
}

```

Decíamos que la primera falla conceptual de hacer UTDD es que se escribían pruebas para pequeñas porciones de código aisladas, típicamente clases en el paradigma de objetos. Como un sistema es un conjunto de módulos interactuando, esto ya era problemático. Por lo tanto, necesitamos escribir pruebas que tengan en cuenta las relaciones entre distintos objetos.

Como siempre que hablamos de varios módulos caemos en el problema de qué desarrollar primero y cómo probar las interacciones con módulos aún no implementados. La idea más simple es la de construir módulos ficticios o *stubs*, y fue la primera que se encaró.

Así se pueden construir módulos ficticios que devuelvan valores fijos o tengan un comportamiento limitado, solo para probar.

Al avanzar en estos tipos de pruebas se fue notando la necesidad de distintos tipos de módulos⁷⁰ ficticios, que son los que Meszaros [Meszaros 2007] llama *Dobles de prueba*⁷¹, y que clasifica así⁷²:

- *Dummy Object* (literalmente, “objeto ficticio”): son aquellos que deben generarse para probar una funcionalidad, pero que no se usan en la prueba. Por ejemplo, cuando un método necesita un parámetro, pero el valor del mismo no se usa en la prueba.
- *Test Stub* (literalmente, “muñón”): son los que reemplazan a módulos reales del sistema, generalmente para generar entradas de datos o impulsar funcionalidades del módulo que está siendo probado. Por ejemplo, módulos que invocan mensajes sobre el módulo sometido a prueba.
- *Test Spy* (literalmente, “espía”): se usan para verificar los mensajes que envía el módulo que se está probando, una vez corrida la prueba.
- *Mock Object* (literalmente, “objeto de imitación”): son módulos que reemplazan a los reales del sistema para observar los mensajes enviados a otros módulos.
- *Fake Object* (literalmente, “objeto falso”): son módulos que reemplazan a otros con implementaciones alternativas. Por ejemplo, un reemplazo de una base de datos en disco por otra en memoria por razones de desempeño.

Notemos que con cada una de estas soluciones se busca separar el módulo que se está probando de los demás, disminuyendo

las dependencias. De alguna manera, lo que se está buscando es mantener las pruebas de integración como pruebas unitarias, al limitar la prueba a solo un módulo en particular.

Por lo tanto, no estamos ante verdaderas pruebas de integración, aunque estemos más cerca. De allí que algunos especialistas (por ejemplo, Freeman y Pryce [Freeman 2010] presentan lo que denominan NDD) recomendaron estrategias para ir reemplazando los dobles por módulos reales a medida que se iban generando. Y así llegaríamos a la prueba de integración mediante la construcción incremental a partir de pruebas unitarias.

Muchos de estos dobles no se usan solo porque haya módulos aún no implementados. A veces los usamos porque es muy lento o muy complicado probar usando una base de datos o un sistema externo. Por lo tanto, estas técnicas sirven también para probar desacoplando interacciones.

Por supuesto, a medida que surgían estas técnicas, fueron construyéndose frameworks que facilitaban la generación y ejecución de pruebas que las incluyeran. Entre ellos, y solo a modo de ejemplo, podemos nombrar a jMock⁷³, mockito⁷⁴ e EasyMock⁷⁵, para Java; NMock⁷⁶, moq⁷⁷ y Rhino Mocks⁷⁸ para .NET. En otros lenguajes incluso se pueden generar objetos ficticios desde herramientas más abarcativas, como ocurre en el caso de Ruby, con RSpec.⁷⁹

Automatizando pruebas de aceptación

En el capítulo “Empezando por la aceptación”, vimos las técnicas basadas en pruebas de aceptación, denominadas ATDD, BDD y STDD, y sus ventajas. Todas ellas, que son lo suficientemente parecidas para que las consideremos equivalentes, admiten ser automatizadas.

En la práctica completa de ATDD, se expresan requerimientos como ejemplos concretos antes del desarrollo de un requerimiento, se automatizan esos ejemplos para luego escribir el código. A medida que se va construyendo el sistema, se van corriendo las pruebas unitarias, de integración y de aceptación, todas ellas automatizadas.

De esta manera, las pruebas de aceptación de una *user story* se convierten en las condiciones de satisfacción del mismo, y los criterios de aceptación, al estar automatizados, se convierten en especificaciones ejecutables.

Por lo tanto, así como UTDD pretende ser una técnica de diseño detallado, BDD y ATDD se presentan como técnicas de diseño basado en dominio. En BDD y ATDD se pone el foco en que el software se construye para dar valor al negocio, y no debido a cuestiones técnicas, y en esto están muy alineadas con las premisas básicas de los métodos ágiles.

En muchos casos, para automatizar pruebas de aceptación, se trabaja con formularios en forma de tabla, que suelen ser mejor comprendidos por no informáticos, y de esa manera se consigue una mayor participación de personas no especializadas.

Las tablas resultan una notación usual para representar entradas y salidas calculables, aunque son definitivamente inadecuadas para representar flujos de tareas, y requieren bastante trabajo de programación para generar pruebas automáticas.

También se utilizan otros formatos, como texto anotado con las palabras “Dado, Cuando, Entonces”, que luego permitan generar código.

Por ejemplo, lo que sigue es una especificación de un escenario del juego de Ta-Te-Ti en Gherkin, uno de los lenguajes más populares para escribir especificaciones de pruebas de aceptación.

```
Característica: Ta-Te-Ti Jugador contra PC
  Para que el juego sirva para un jugador solo
  Como cliente
  Deseo que la PC pueda jugar con inteligencia contra el jugador

  Escenario: Bloqueo de posible Ta-Te-Ti por PC
    Dado Jugador tiene ocupada una celda en una línea
    Y Hay dos celdas libre en la línea
    Y Jugador está en su turno
    Cuando Jugador coloca una X en otra celda de la línea
    Entonces PC debe colocar una O en la celda libre de la linea
```

Lo que se busca con este texto anotado es una solución intermedia entre el texto libre y las especificaciones en código.

Respecto de las especificaciones en código, que también son posibles, su obvia ventaja radica en la facilidad de generación de pruebas automáticas, a veces incluso a un costo nulo. Sin embargo, son las menos adecuadas como herramientas de comunicación y provocan una resistencia casi instintiva de los roles no técnicos.

Últimamente ha ido ganando también habitualidad el uso de lenguajes específicos de dominio (DSL, por su sigla en inglés). Si bien en apariencia son la alternativa más conveniente, presentan como dificultad la falta de lenguajes para gran cantidad de tipos de dominio.

Automatizando pruebas de interacción

Las primeras pruebas en ser automatizadas fueron las de interacción con el usuario. Ya desde la década de 1980 hay herramientas que permiten grabar una prueba realizada a mano por un tester, para después ejecutarla en forma repetitiva, sea para checar la resolución de un problema o para hacer pruebas de regresión.

Todo esto es anterior a TDD y a los métodos ágiles. Ahora bien, esta manera de trabajar muestra un conflicto con la filosofía de TDD: la propia noción de construir algo para luego grabar su ejecución no se lleva bien con la práctica de escribir antes las pruebas, premisa básica de TDD.

Encima, ha habido muchas críticas a la automatización de pruebas a través de la interfaz de usuario:

- Mugridge y Cunningham [Mugridge 2005] dicen que no conviene hacerlas, porque la interfaz de usuario es muy cambiante y las pruebas se desactualizan con mucha frecuencia.
- North [North 2006], Adzic [Adzic 2009], Freeman y Pryce [Freeman 2010] ponen énfasis en separar el qué del cómo, diciendo que el cómo es menos relevante y más cambiante, y por lo tanto debe ser un tema a discutir como parte de las actividades de

implementación.

- Otros han destacado que las pruebas de interfaz de usuario suelen ser muy lentas de ejecutarse.
- También hay cuestiones que sólo las puede probar satisfactoriamente un usuario humano. Por ejemplo, la ubicación de los botones, los colores, la consistencia de la aplicación en su totalidad, no son fáciles de automatizar. Y cada cambio en la interfaz de usuario requiere al menos un recorrido por la misma.

Incluso Meszaros [Meszaros 2007] ha destacado los problemas que él denomina “de la prueba frágil”, que consisten en:

- Sensibilidad al comportamiento: los cambios de comportamiento provocan cambios importantes en la interfaz de usuario, que hacen que las pruebas de interacción dejen de funcionar.
- Sensibilidad a la interfaz: aun cambios pequeños a la interfaz de usuario suelen provocar que las pruebas dejen de correr y deban ser cambiadas.
- Sensibilidad a los datos: cuando hay cambios en los datos que se usan para correr la aplicación, los resultados que esta arroje van a cambiar, lo que hace que haya que generar datos especiales para probar.
- Sensibilidad al contexto: igual que con los datos, las pruebas pueden ser sensibles a cambios en dispositivos externos a la aplicación.

Esta fragilidad, más el tema ya expuesto de la lentitud de las pruebas a través de la interfaz de usuario son las que más quejas han provocado.

Existen muchos casos de empresas que apuestan por automatizar pruebas de interacción. Para hacerlo con algo de éxito, es indispensable dedicar los recursos adecuados. Hemos visto en muchos proyectos equipos enteros de 5, 8 o más programadores dedicados exclusivamente a la programación y mantenimiento de pruebas de comportamiento automatizadas.

En estos proyectos, el equipo de programadores no especifica las pruebas (esto lo hacen analistas de calidad), sino que se dedican solo a las tareas técnicas de programación y mantenimiento.

Con el tiempo, es usual que el mayor énfasis sea en el mantenimiento del conjunto de pruebas, dado que por su naturaleza estas son muy frágiles y tienen a dar muchos falsos negativos.

En proyectos complejos, es usual que se utilice o arme un framework para trabajar ordenadamente, en especial en las aplicaciones web modernas, donde el comportamiento es muy complejo.

En una ocasión, una empresa contaba con dos sistemas de CRM diferentes, uno para el mercado masivo y otro para grandes clientes.

Se hizo una unificación de ambos, en forma progresiva, hasta lograr una sola base de datos e interfaz, todo sin interrumpir el funcionamiento ni hacer cambios repentinos para el usuario. Se consideró crítico para el éxito de este proyecto contar con un conjunto de pruebas de interacción. En este caso había 8 programadores trabajando exclusivamente en desarrollo y mantenimiento de pruebas, que en ese proyecto llevaban el título de Ingenieros de Pruebas.

P. S.

Pero esto está dejando de lado algo fundamental: el valor para el negocio, que se logra mediante el comportamiento, se exhibe y se materializa, también en una proporción muy mayoritaria de los casos, a través de la interfaz de usuario.

Por añadidura, la mayoría de los usuarios reales solo consideran que la aplicación les sirve cuando la ven a través de su interfaz de usuario. Ergo, si cada iteración debe terminar con un entregable potencial, solo debería considerarse terminada si lo que se entrega incluye la interfaz de usuario con su interacción y navegación probadas.

En definitiva, no se debería dejar sin especificar con pruebas la interfaz de usuario.

Para llevar la idea de TDD a este tipo de pruebas, se han hecho intentos de construir pruebas a partir de maquetas de interfaz de usuario y modelos, tales como en los trabajos realizados en el LIFIA⁸⁰ [Burella 2010]. En este sentido también destaca el trabajo de Meszaros y otros [Meszaros 2003].

No obstante, todavía falta, para considerar a TDD una técnica completa, que las pruebas de interacción sobre la interfaz de usuario puedan integrarse a las pruebas de comportamiento, a nivel de modelo de negocio.

Esto es, si tenemos una aplicación en capas, habitualmente separamos la lógica de la aplicación, o modelo de dominio, de la interfaz de usuario. El modelo es lo que suele ser más fácil de someter a pruebas automáticas de aceptación.

Sin embargo, hay algo que no se suele tener en cuenta: la interfaz de usuario también tiene comportamiento, a veces complejo, al punto que fue uno de los primeros usos de la programación orientada a objetos. Por eso mismo es que ha habido intentos de usar BDD para desarrollar interfaces web. El más destacado es la utilización del patrón denominado *Page Object* [Stewart 2011].

De todas maneras, aun cuando no haya una integración entre ambos tipos de pruebas, hacerlas por separado es un gran avance. De hecho, probar la interfaz de usuario sabiendo de antemano que el comportamiento está funcionando bien y ha sido probado es más tranquilizante, ya que seguramente nos vamos a encontrar con menos problemas en la prueba a través de la interfaz de usuario. Por ejemplo, hay herramientas, como Selenium, que sirven para probar aplicaciones web a través de esta interfaz.

Por supuesto, luego de las pruebas automatizadas de aceptación, hay que realizar pruebas a mano, las que hemos llamado exploratorias, que garantizan aspectos de calidad que solo el criterio humano puede discernir.

Formas de automatizar

Ahora bien, las maneras de automatizar no son siempre las mismas.

En primer lugar, depende de los medios que utilicemos para la interacción entre las pruebas y la aplicación. Una aplicación construida en capas admite que las pruebas se hagan usando a las distintas capas como interfaz del sistema a probar. Simplificando, hay dos maneras principales de interactuar con la aplicación a probar:

- Mediante la interfaz de usuario, simulando un usuario humano.
- Mediante una interfaz programática (*API*⁸¹).

En el primero, a veces son la única forma de probar en aquellos casos en que la *API* del programa no prevé acceso a toda la funcionalidad desde afuera.

La otra cuestión es la manera en que materializamos las pruebas:

- Si se construyen escribiendo código (los guiones o *scripts*, a la manera del patrón *Scripted Test* [Meszaros 2007]).
- Si se construyen grabando pruebas corridas ad hoc por un tester (según el patrón *Recorded Test* [Meszaros 2007]).

Si combinamos ambas clasificaciones podemos obtener cuatro posibilidades.

Las herramientas del tipo xUnit trabajan con pruebas basadas en guiones que se comunican a través de la *API* del sistema.

En el otro extremo, los robots que simulan interacción de usuarios, generan pruebas grabadas que se corren usando la interfaz de usuario.

Pero hay otras dos situaciones intermedias. Por ejemplo, a través de la *API* del sistema se pueden utilizar herramientas que graban interacciones con el sistema para correrlas en forma repetitiva, a través de una capa de servicios. Y a través de la interfaz de usuario podemos también hacer pruebas basadas en guiones.

Herramientas

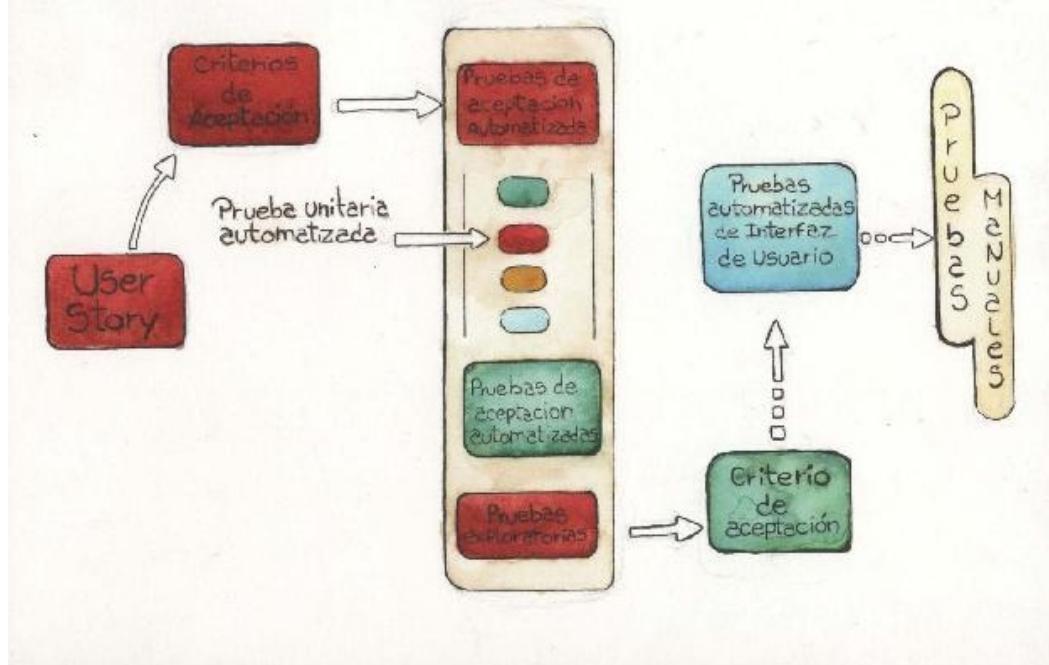
Cualquier lista de herramientas que hagamos va a resultar forzosamente incompleta, y se desactualizará día a día. Sin embargo, resulta ilustrativo ver algunas de las más relevantes en el momento en que estamos escribiendo estas líneas.

“Ver tabla comparativa en <http://unamiradaagil.com/tablas/>”

El ciclo de desarrollo automatizado

En definitiva, si automatizamos todos los tipos de pruebas, estaríamos ante un ciclo de desarrollo como el de la figura 8.3.

Figura 8.3. Ciclo de desarrollo automatizado.



Este ciclo de desarrollo se puede enriquecer aún más, incorporándole más actividades a realizar en forma automática, como veremos en el capítulo “Integrando el producto al instante”.

En resumen

Siempre es bueno que las computadoras nos ayuden en tareas en las que tienen ventajas naturales. Por eso, si una tarea es repetitiva, necesita resultados iguales para iguales estímulos independientemente de otros factores, y requiere ser ejecutada lo más rápido posible, debe automatizarse.

Muchas pruebas caen dentro de esta categorización, y por eso la planteamos. Puede que haya diferentes opiniones de hasta dónde automatizar, pero claramente hay situaciones en que la automatización no debería ser una opción.

Además, existen muchas herramientas que facilitan estas tareas, además de varios enfoques metodológicos que las respaldan.

Para ser útiles, conviene que corran rápidamente, que sean independientes unas de otras y que sirvan como soporte ante regresiones.

En el capítulo “Integrando el producto al instante”, trataremos otras automatizaciones al tratar los temas de integración y entrega continua.

⁶⁵ Con esto no estamos descartando a los usos que tiene y tendrá la Inteligencia Artificial. Solo estamos diciendo que lo que se esperaba de la misma es más que lo que luego se logró.

⁶⁶ Refactorización, como se explica en el capítulo “Arquitectura y diseño en emergencia”, es una práctica del desarrollo ágil de software que consiste en mejorar la calidad del código para facilitar su mantenimiento, pero sin alterar el comportamiento observable del mismo.

⁶⁷ A lo largo del día, los desarrolladores y testers cambian su nivel de atención. Incluso en distintas etapas de un proyecto, las personas están más o menos motivadas y alertas a problemas. Lo que se busca con la automatización es evitar que la evaluación sobre la calidad del producto quede en manos de personas cansadas o apresuradas.

⁶⁸ Recordemos que TDD nació como una manera de hacer diseño orientado a objetos.

⁶⁹ No se muestra la clase completa, ya que solo se pretende mostrar el estilo de escritura de pruebas en código. Por supuesto, las pruebas completas deben ser mucho más abundantes.

⁷⁰ Hablamos de módulos para mantener la neutralidad respecto del paradigma. Meszaros trabaja suponiendo el paradigma de objetos, y por lo tanto habla de objetos y clases, no de módulos en forma genérica.

⁷¹ La expresión en inglés que usa Meszaros es “Test Doubles”.

[72](#) Si bien Meszaros hace una distinción muy minuciosa entre todos los tipos de dobles, no entraremos en detalles, teniendo en cuenta que muchas personas definen en forma ligeramente diferente cada tipo, haciendo que las distinciones entre un tipo y otro no siempre estén claras.

[73](#) <http://jmock.org/>

[74](#) <https://code.google.com/p/mockito/>

[75](#) <http://easymock.org/>

[76](#) <http://nmock3.codeplex.com/>

[77](#) <https://code.google.com/p/moq/>

78 <http://hibernatingrhinos.com/oss/rhino-mocks>

[79](#) <http://rspec.info/>

[80](#) Laboratorio de Investigación y Formación en Informática Avanzada, de la Universidad Nacional de La Plata.

[81](#) “Application Programming Interface” o “interfaz de programación de la aplicación”.

Esto está listo

Vamos a describir dos situaciones que, aunque no son totalmente reales, fueron inspiradas por episodios de proyectos pasados.

Situación 1

Los desarrolladores y testers de un equipo se encuentran en una reunión. Están investigando la causa de unos defectos que se filtraron en una entrega al cliente. Los testers dicen que llegan demasiados defectos que consideran de sentido común y que eso les quitó tiempo para hacer las pruebas de regresión. Los desarrolladores argumentan que los testers reportan incidentes sobre casos que no estaban especificados. Cuando entran en detalle, se dan cuenta que todas las pantallas desarrolladas tienen problemas menores visuales y en muchas faltan validaciones o mensajes.

Situación 2

Llega el día de la demostración del producto y un integrante del equipo expone ante el cliente la funcionalidad comprometida. Se había pedido un módulo que imprimiera tickets para eventos deportivos. La reunión es todo un éxito y el sistema funciona correctamente. Sin embargo, al cabo de unos días de instalado en producción, empiezan a llegar pedidos para arreglar el producto porque no está funcionando bien. Al consultar sobre la falla, el cliente informa que el sistema está dando problemas con las impresiones. En la charla menciona que el área de tecnología cambió todas las impresoras el mes anterior para reemplazar las viejas, que tenían 5 años de antigüedad. A la vez, muchos errores reportados corresponden a problemas con la impresión de textos dado que la empresa decidió utilizar el software en su filial árabe donde los textos deben escribirse de derecha a izquierda. El responsable del proyecto explica que las pruebas se habían realizado con otra impresora y en castellano. El cliente responde que, dado que la unidad de negocios más grande está en Medio Oriente, no se puede considerar terminado un producto que no satisface su filial más importante. El equipo se resigna y se queja porque en los proyectos siempre pasan estas cosas.

Estas situaciones son muy comunes en los proyectos de desarrollo. Una parte cree que cumplió con el trabajo pero cuando la otra lo recibe no satisface las expectativas. Muchas veces, se asumen reglas implícitas que después resultan inexistentes. También es común que una parte considere que su trabajo está suficientemente bien como para ser entregado. La comunicación es uno de los elementos esenciales a gestionar durante un proyecto y muchos problemas pueden generarse por malos entendidos, suposiciones, por no aclarar algún punto, o por ser ambiguos.

Si analizamos la primera situación, lo que el equipo de desarrollo consideraba terminado, era algo inestable para testing. Por este motivo, los testers lo devolvían con gran cantidad de bugs reportados, generando varias idas y vueltas. Es decir, no había un acuerdo sobre la calidad mínima que la entrega debía tener para considerarse completa. Estas idas y vueltas dejaban menos tiempo para probar todo el sistema y, por consiguiente, el resultado final era una entrega inestable.

En la segunda situación tenemos varios problemas. En primer lugar, ni el equipo ni el cliente tienen claro el alcance de la entrega. Los primeros consideran que el sistema entregado está funcionando bien porque permite imprimir recibos en castellano pero el cliente esperaba que funcione también en árabe y esto último no fue contemplado.

Podríamos suponer que existió un problema en el relevamiento por parte del equipo, ya que no fue capaz de reconocer una necesidad que aportaba valor para esta empresa (la impresión de tickets en árabe). Pero, por otro lado, también hubo problemas para delimitar el alcance de la funcionalidad, puesto que como el cliente se quejó y el equipo no supo cómo responder más allá de la queja, queda en evidencia que nunca hubo un entendimiento común de lo que se esperaba de la funcionalidad.

Cuando hablamos de falta de entendimiento común, no es solo que ambos sepan que se va a hacer, sino que es fundamental que las dos partes estén de acuerdo *sobre cuando la pieza construida estará completa*⁸². Esto se puede ver bien en el ejemplo, porque aunque el equipo le dí la razón al cliente y arregle el software para que funcione en árabe, este luego puede volver a quejarse en unos meses porque la versión japonesa o china no imprime bien y una vez arreglados estos problemas, seguiríamos con otros sistemas de escritura cada vez más raros para nosotros. El problema no está en el software sino en que las partes nunca entendieron lo mismo sobre lo que había que hacer.

Entonces, más allá del problema de relevamiento, vemos que no dejar claro el alcance de una funcionalidad genera falsas expectativas, frustraciones, trabajo adicional y roces en la relación equipo-cliente. En este caso, el equipo tendría que haber utilizado alguna técnica para acordar de antemano qué puntos tienen que cumplirse para que se considere aprobada una funcionalidad y una entrega.

Para complicar más el tema, tenemos el problema de funcionamiento con las nuevas impresoras. El equipo empezó el desarrollo para un dispositivo que más adelante fue cambiado por un área no directamente vinculada con el proyecto. Lo que sucede en este caso lo podemos considerar como una modificación en el ambiente original y, aunque es muy poco probable que estas situaciones puedan evitarse, existen formas de mitigar riesgos *acordando las condiciones sobre las cuales se realizarán las pruebas o la aceptación del producto*.

Ambas situaciones surgen por diferencias entre lo que ambas partes entienden por una entrega de calidad aceptable. Ambas generan trabajo adicional, desvían el foco del proyecto y generan roces entre los participantes, pero la segunda situación tiene el inconveniente de ser una entrega al cliente y generalmente condicionar pagos, bonos, cerrar etapas, liberar parte del equipo para nuevos proyectos, iniciar nuevas oportunidades, etc. Por esto, definir qué significa que algo está listo debería ser considerado crítico para el éxito de un proyecto.

A mí me funciona

Podemos identificar varios problemas típicos que aparecen en el trabajo diario de un equipo y entre el equipo y su cliente. A continuación, algunos ejemplos:

El código no compila: este es un caso lamentablemente recurrente que, por suerte y gracias a las técnicas de integración continua, está desapareciendo. Es el típico caso en que un integrante no envía la totalidad de sus cambios al repositorio y no verifica si la nueva versión compila.

La entrega tiene funcionalidad incompleta: si bien a primera vista parece que todo anda bien, el sistema carece de algunas características menores solicitadas como el envío de notificaciones por mail o guardar datos en un registro de auditoría o comunicar cambios a algún sistema externo. En líneas generales, puede ser que una funcionalidad esté desarrollada pero no haga todo lo que tiene que hacer.

La entrega no tiene tests unitarios: el desarrollador no entrega las pruebas que validan que su código hace lo que tiene que hacer. Para tener conciencia de lo importante que son los tests unitarios, en algunos equipos, no entregar pruebas unitarias es equivalente a no entregar. Si alguien no puede asegurar que lo que desarrolló funciona es muy probable que otro tenga que verificarlo por él más tarde. Esto generará idas y vueltas innecesarias si es que faltaba contemplar algún punto. Por lo tanto, en estos equipos, funcionalidad entregada sin pruebas unitarias no se acepta.

Validaciones no realizadas: todo funciona siempre y cuando nos acordemos de completar todos los campos del formulario, de no poner letras cuando nos piden números, de acordarnos de ingresar las fechas en formato dd-mm-aaaa. Si nos olvidamos de cumplir con alguna de estas reglas, el sistema falla o peor, queda en un estado interno inconsistente.

Solo se probó el camino feliz: es bastante común encontrar desarrollos en los que sólo se considera esta vía. Por ejemplo, si hay que enviar un mensaje por red podría ser que no se hubiese considerado que no haya conexión o que el receptor me responda con un mensaje no esperado.

Solo funciona en la máquina de quien lo desarrolló: otro punto que es bastante frecuente encontrar, tiene que ver con funcionalidades que responden correctamente en el ambiente de desarrollo pero que por alguna diferencia de ambiente (configuración regional, software de base con otra versión, etc.) no funciona en el ambiente final. Esto también está disminuyendo con los servidores de integración continua pero aún ocurre.

No estaban definidas las pruebas: un caso frecuente es desarrollar una funcionalidad sin conocer bien los criterios con los que va a ser verificada. Esta situación genera que muchos errores evitables lleguen a instancias muy avanzadas (aceptación de la entrega) para volver a desarrollo porque no cumplen con las expectativas de quien recibe el producto.

No se cumple con las pruebas: puede darse el caso que estén definidas las pruebas pero que el desarrollador construya solo mirando la funcionalidad, sin validar que cumpla con las pruebas generadas. Como estas complementan la funcionalidad aportando casos especiales, muchas veces aparecen reglas de negocio que no fueron consideradas a pesar de que el desarrollador esté convencido de que terminó con su tarea.

Se entrega con errores: ningún software está libre de errores, pero cuando se realiza una entrega y esta contiene errores de una cierta criticidad, es esperable que vuelva rechazada. Si existen incidentes reportados para una funcionalidad y la cantidad o severidad de los mismos es considerable, es poco probable que podamos afirmar que el trabajo está terminado.

No cumple con los estándares acordados: el equipo definió que había que respetar la notación estándar de código, que se iban a seguir ciertos patrones de arquitectura y diseño, que todos los mensajes se guardarán en la base de datos pero, sin embargo, nada o poco se cumple en el código que se acaba de enviar al repositorio. A veces son impuestos por el cliente por normativa de la empresa o para asegurarse un código legible cuando quede en su poder. Un incumplimiento de un estándar obliga a reprogramar y a pasar por las instancias de testing y aceptación nuevamente de manera innecesaria.

No cumple con los criterios estéticos definidos: la entrega está impecable a nivel funcional, pero el color elegido para los mensajes no tiene nada que ver con los que se definieron para la aplicación. También pasa con diferentes tonalidades, tamaños, tipografías, ubicación de controles, manejo de espacios, orden de tabulación, etc. Puede ser que no invaliden totalmente una entrega, pero no deberíamos considerar completa una funcionalidad que no respeta los criterios estéticos.

Solo funciona en algunas plataformas: también es frecuente, por ejemplo en el desarrollo web, aunque no se limita a este tipo de proyectos. Se suele dar cuando un desarrollador programa con éxito una parte del sistema y la prueba en su navegador preferido. Luego, los usuarios utilizan otros navegadores y el sistema no funciona por incompatibilidad entre los mismos.

Como vemos, existen muchas categorías de problemas que surgen de la falsa sensación que tenemos al creer que terminamos de desarrollar la funcionalidad. Cada fuente de problemas es un aporte al mal funcionamiento del proyecto y un granito de arena para contribuir al derrumbe de las expectativas del cliente.

Terminé es que terminé

Si queremos evitar los problemas que ya mencionamos, tenemos que poder dejar en claro que entendemos cuando decimos “está listo” o “terminé”.

Para que todos tengan un mismo entendimiento, lo mejor que se puede hacer es generar un listado de puntos que el equipo considera necesario verificar antes de poder decir que la tarea está completa. Para este caso son muy útiles las listas de verificación. Con esta herramienta se pueden definir cuáles son los ítems que deben estar cumplidos para que alguien pueda decir que su parte está completa.

Por ejemplo, podríamos definir una lista de verificación que defina que una funcionalidad de un desarrollador está completa siempre y cuando cumpla con cada uno de los puntos del siguiente listado:

- En el ambiente de desarrollo:
 - ° El código compila.
 - ° El código tiene sus pruebas unitarias.
 - ° Las pruebas se ejecutan correctamente.
 - ° Si hay pantallas, las mismas tienen todas sus entradas con validaciones.
 - ° Las pruebas fueron definidas.
 - ° Todas las pruebas se ejecutan exitosamente.
- Al integrarlo con el resto del código:
 - ° El código del sistema compila.
 - ° El ambiente de integración ejecuta todas las pruebas exitosamente.

Hay autores⁸³ que proponen diferentes listas de verificación que tratan de considerar todos los puntos a tener en cuenta cuando se desarrolla. En nuestra opinión, esto genera condiciones demasiado estrictas y el listado se vuelve tan grande que pierde su utilidad. Además, genera el rechazo de quienes lo usan porque lo ven como un impedimento para avanzar con su tarea. Muchas veces, estas listas de verificación entran en demasiado detalle solo para asegurar que ningún aspecto queda fuera de control y olvida que el objetivo fundamental de estas herramientas es cumplir con las expectativas de los clientes, eliminar el trabajo innecesario y entregar más valor en poco tiempo.

Por eso es importante que cada equipo, a medida que avanza el proyecto, vaya definiendo e incorporando los puntos que considere necesarios⁸⁴. Siempre resulta un compromiso más difícil de rechazar si los mismos integrantes del equipo asumen ese compromiso, y el mismo no viene impuesto de afuera.

Quiero que me aprueben las entregas

Cuando tenemos que acordar los puntos que tienen que cumplirse para dar por aprobada una entrega recurrimos a los criterios de aceptación. Lo que se busca con estos es fijar las condiciones por escrito mediante las cuales la entrega se considerará aprobada.

Un ejemplo de criterio de aceptación podría ser:

Todas las pruebas de aceptación pasan exitosamente.

Existen 0 errores críticos, 0 medios y 4 menores.

A su vez, hay que definir sin ambigüedades a qué se considera error crítico, medio y menor.

Los criterios de aceptación no son solo un concepto ágil, sino que son utilizados ampliamente en la gestión de alcance de todo tipo de proyectos.

Ya hablamos de los criterios y casos de aceptación en el capítulo “Empezando por la aceptación”. Allí explicamos que los métodos ágiles nos permiten ir construyendo estos casos a partir de la definición de diferentes ejemplos de uso del sistema. Es así que si una entrega consta de un conjunto de funcionalidades, mis casos de aceptación constarán de diferentes ejemplos de uso para cada una de estas funcionalidades.

Los casos de aceptación los vamos acordando con nuestro cliente en la medida que vamos relevando el sistema. Es decir, contribuyen, por un lado, a entender mejor la funcionalidad y, por otro, nos ayuda a generar el material que se utilizará para verificar si la entrega debe ser o no aprobada. El punto importante a tener en cuenta es que los casos deben estar escritos, o por lo menos propuestos, *por el cliente* puesto que será él quien los utilice para validar las entregas.

Entrando en detalle, un caso de aceptación indica, para un posible uso de la funcionalidad que estamos desarrollando, el comportamiento que se espera del sistema. Pero es importante tener en cuenta que un caso de aceptación puede terminar expandiendo esta funcionalidad y requiriendo nuevos comportamientos en el resto.

Vamos a poner un ejemplo. Supongamos que estamos desarrollando un sistema para una empresa de tarjetas de crédito y tenemos la funcionalidad *Abonar con tarjeta de crédito*.

Los casos de aceptación que podrían surgir del intercambio con el cliente podrían incluir los siguientes:

Que cuando el usuario de tarjeta:

- Quiere pagar y tiene crédito, pueda realizar la compra.
- Quiere pagar y no tiene crédito no pueda realizar la compra.
- Pague con la tarjeta, disminuya el crédito en la misma cantidad que la compra realizada.
- Intente pagar con la tarjeta y como está denunciada, no pueda realizar la compra.
- Ingrese datos inválidos, el sistema informe que los datos son incorrectos.

También podrían ser más específicos, por ejemplo:

Dado que el cliente tiene una tarjeta con crédito, número *1111111111*, vencimiento *12/15* y estado *no denunciada*.

Cuando,

• Ingresa todos los datos de la tarjeta pero en el vencimiento ingresa *12/15*, entonces el sistema muestra el mensaje “los datos de la tarjeta no son correctos”.

• Ingresa una fecha de vencimiento previa a la fecha actual, entonces el sistema muestra el mensaje “probablemente la tarjeta no esté vigente”.

• No ingresa número de tarjeta, entonces el sistema muestra el mensaje “algunos datos no fueron completados”. Remarcando el campo del número de tarjeta.

Ahora bien, estos casos están directamente relacionados con la funcionalidad, pero podría haber otros que tengamos que tener en cuenta si es que ya existe otra funcionalidad a la que este nuevo desarrollo afecta. Siguiendo con nuestro ejemplo, supongamos que ya estaba desarrollado el resumen de cuenta del cliente y el informe diario de operaciones del comercio. En ese caso, es probable que el cliente aporte los siguientes casos de aceptación:

Dado que el usuario con una tarjeta con \$15 de saldo realizó una compra por una botella de agua que costaba \$5, cuando pague con la tarjeta,

- A fin de mes aparezca dicha compra en el resumen de cuenta y el saldo luego de esa compra sea de \$10.
- En el informe diario de operaciones del comercio donde se concretó la compra, se visualice la compra realizada por \$5.

¿Y si el usuario quiere garantizar algún tema relacionado con el rendimiento? En ese caso podría agregar casos de aceptación como los siguientes:

- Que cuando el sistema reciba 50 transacciones en un segundo procese las 50 transacciones.
- Que cuando el usuario pague con la tarjeta, el resultado de la operación se informe en menos de 5 segundos.

Si tenemos una entrega definida, su aceptación va a estar condicionada por el cumplimiento de todos los casos de aceptación que el cliente haya definido para las funcionalidades desarrolladas, las previas afectadas por esta entrega y las diferentes restricciones impuestas sobre las mismas. Todas estas cuestiones, como vimos, se pueden especificar mediante la técnica explicada arriba.

El otro punto a tener en cuenta, si recordamos la segunda situación al inicio del capítulo con el problema de la impresora, es sobre la definición del ambiente de aceptación. Es fundamental relevar las expectativas del cliente en cuanto al ambiente en el que espera ejecutar nuestro entregable.

En el caso del pago con tarjeta, se podrían incorporar ciertos casos de aceptación como los siguientes:

- Que cuando el pago se realice con las terminales T21, y Tz3 con las versiones de firmware 1.22, 1.23 y 2.0, el pago se registre correctamente.

Con este texto mínimo estamos limitando las posibilidades de rechazo de nuestra aplicación por ejecutarse en un ambiente no conocido. Al especificar los ambientes, el equipo de desarrollo sabe donde probar su trabajo, o al menos sabe que es lo que debe emular. Por otro lado, el cliente se asegura que el sistema que se está construyendo se ejecutará sin inconvenientes en estos dispositivos.

Lo que hay que evitar en estos casos son afirmaciones ambiguas del tipo: *deberá funcionar con la última versión del sistema operativo*.

La idea, en definitiva, es que este listado de casos forma parte de la entrega y el cliente pueda ir revisando caso por caso y marcando si se cumplen o no como si estuviera verificando una lista de verificación. De esta manera, se logra generar un entendimiento común de la completitud de la entrega y se establece una técnica relativamente sencilla de verificar que se cumple con todo lo que el cliente espera.

En resumen

Viendo la primera situación planteada, parece obvio que, si el equipo hubiera acordado una lista de verificación para determinar que una tarea estaba lista, las discusiones descriptas no se habrían suscitado.

Y si el cliente y el equipo del segundo ejemplo hubieran acordado los casos de aceptación, habría quedado claro como se esperaba probar la entrega y el cliente hubiera tenido la posibilidad de indicar que iba a probar con el idioma árabe. Esto habría dado oportunidades al equipo para considerarlo en el desarrollo. A la vez, mientras se especificaban los casos para fijar el ambiente de pruebas, el cliente podría haber indicado que las pruebas debían ejecutarse correctamente en las nuevas impresoras o por lo menos, si no lo mencionaba, quedaría claro que la necesidad de impresión sobre otros dispositivos adicionales a los definidos originalmente serían un añadido y no una falla del producto original.

Ahora sí, con la definición de “Listo” del primer ejemplo y utilizando la técnica de los casos de aceptación como mostramos en la solución del segundo caso, tenemos herramientas para favorecer un entendimiento común dentro del equipo y con nuestro cliente. En ambos casos *la construcción es conjunta* y requiere la participación tanto de quien genera el trabajo como de quien es el encargado de verificarlo ya que lo que se está haciendo es construyendo acuerdos. Ambas herramientas son relativamente sencillas de utilizar pero, junto a otras prácticas ágiles, brindan un gran poder para evitar conflictos, enfocar el trabajo y asegurar la satisfacción de quien recibe lo que hacemos.

[82](#) En Scrum se denomina Definition of Done (DoD) véase [ScrumOrgDoD].

[83](#) Un ejemplo puede verse en [Waters 2007].

[84](#) Una excelente descripción del proceso de creación de la definición de Hecho se puede encontrar en [Lacey 2008].

Integrando el producto al instante

Es viernes y Nicolás, que estuvo las últimas dos semanas trabajando en una nueva funcionalidad, termina de hacer las pruebas en su máquina. Al observar que no hay errores, envía todos los cambios al repositorio en el que está el resto del producto y parte a su casa. El miércoles de la semana siguiente Diego pretende hacer lo mismo, pero cuando va a integrar sus cambios, se encuentra con los fuentes modificados por Nicolás. Los descarga localmente y todo deja de andar. La aplicación ni siquiera compilaba. Pasaron varios días sin que nadie se diera cuenta que al producto le faltaban porciones de código. El problema es que Diego no sabe bien que es lo que falta porque los cambios fueron tantos durante esas dos semanas que le es muy difícil entender en donde está el problema. Finalmente, aparece Nicolás diciendo que se olvidó de agregar un archivo nuevo que era necesario para la compilación. Una vez solucionado el tema, funciona lo que debió funcionar desde el miércoles. Todavía falta que Diego entienda como incorporar sus cambios dentro del código que por diez días solo conoció Nicolás.

La etapa de integración

Hasta no hace mucho tiempo, uno de los trabajos más complicados dentro del proceso de desarrollo era la integración de un producto a partir del trabajo de uno o varios equipos. Muchos proyectos encaraban esta etapa luego de trabajar semanas, meses y hasta años en sus módulos, quizás queriendo copiar la manera en que se trabaja en otras industrias en donde es posible la construcción de varios componentes por separado para luego dar paso a su ensamblado en un producto final. Un caso frecuente, por ejemplo, en la industria aeronáutica o automotriz.

Quien haya vivido alguna etapa de integración como la mencionada puede estar de acuerdo en que eran muy complejas porque siempre existían problemas entre lo definido y lo construido. Las interfaces entre los diferentes módulos, aunque a veces se respetaban, en muchas ocasiones terminaban con un comportamiento que no coincidía con el esperado. Ante cada problema había que recodificar, volver a hacer la prueba del módulo y a probar todas las interacciones. Pocas veces se trataba de una etapa con final cierto. Las fallas aparecían todo el tiempo y los retrasos en las correcciones de un módulo impactaban en los tiempos de los demás. Por si fuera poco, los cambios podían hacer que un módulo funcione pero que otro dejara de hacerlo.

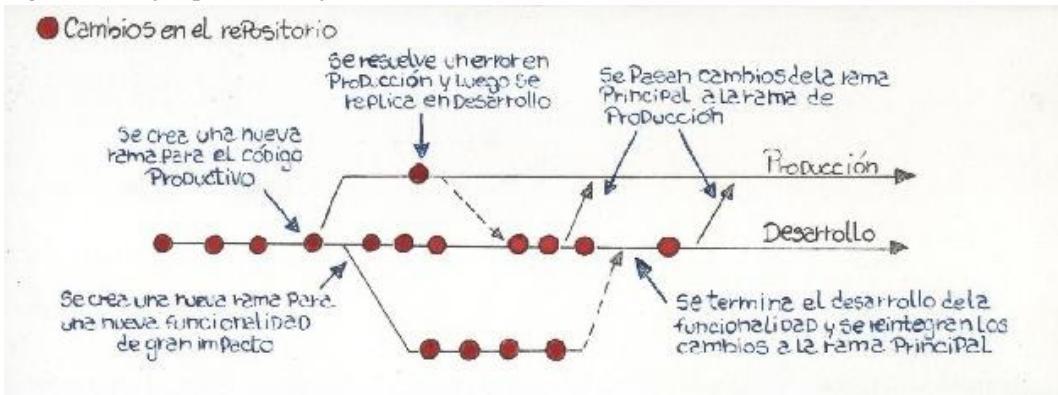
Esta forma de trabajar puede entenderse hace unos 30 años cuando el trabajo en equipo era mucho más complicado por las restricciones técnicas que había, pero desde hace bastante tiempo existen los sistemas de control de versiones (VCS en inglés, nosotros los llamaremos SCV). Los SCVs trajeron muchas ventajas, como por ejemplo:

- Que todo equipo pueda acceder a las diferentes versiones del código del producto.
- Poder revertir cambios.
- Asegurar y auditar el acceso al código fuente.
- Indicar cuando más de un integrante modifica la misma porción de código.

Las herramientas para gestionar estos SCVs también posibilitaron que un equipo pueda generar, a partir de una determinada versión del código, varias ramas paralelas de desarrollo y que una vez terminadas se reintegren a la rama principal. También permiten marcar cual es la versión que se encuentra productiva, o hacer mantenimiento de una versión mientras se desarrolla una nueva y poder trasladar los cambios entre las distintas versiones.

El control de versiones es una actividad que se enmarca dentro de la disciplina de Administración de la Configuración, conocida en inglés como Software Configuration Management.

Figura 10.1. Ejemplo de trabajo con ramas con un SCVs.



Las ventajas de utilizar los SCVs son evidentes y no solo en un entorno ágil⁸⁵. Si algo no funciona, podemos volver a la versión anterior que no tenía problemas, identificar y aislar el cambio en el código que generó la falla.

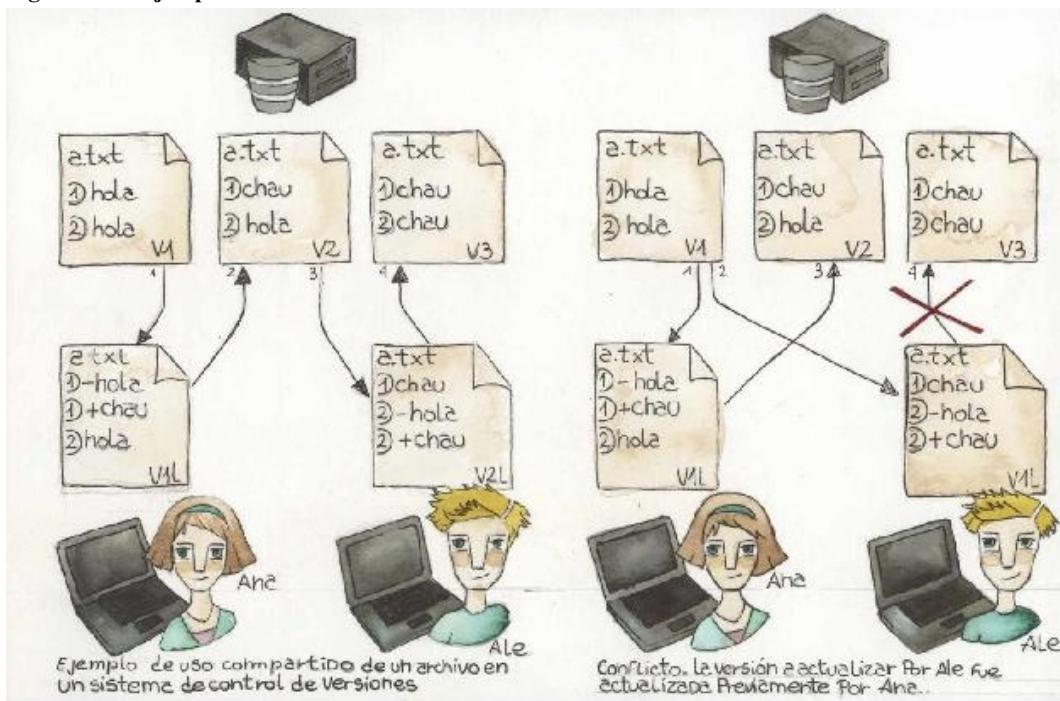
Cómo se usa un SCV

Para quien nunca trabajó con un sistema de control de versiones explicaremos con una breve descripción su funcionamiento. Un SCV es una herramienta de almacenamiento donde se van guardando las diferentes versiones de los archivos con los que se trabaja. Para simplificar las cosas, vamos a decir que existe un repositorio en un servidor al que todos los integrantes del equipo tienen acceso. Para trabajar con repositorios, al menos tenemos 2 lugares donde se encuentra el código: en un ambiente remoto y en nuestro espacio de trabajo, generalmente algún directorio o carpeta en nuestro sistema de archivos.

Cuando queremos trabajar, nos conectamos al servidor para actualizar nuestro ambiente local con los cambios que se pudieron haber hecho desde la última vez que estuvimos desarrollando. Luego trabajamos como siempre con nuestras herramientas generando cambios en el código existente (creando nuevos archivos, haciendo modificaciones y eliminando elementos que ya no necesitamos). Por último enviamos nuestras modificaciones locales al repositorio remoto. Si se detecta que desde que descargamos el código hasta que lo volvimos a enviar algún otro integrante del equipo modificó alguno de los archivos que queremos actualizar, puede generarse lo que se denomina un conflicto. Esto surge porque, cuando existen varios cambios simultáneos a un archivo, no siempre hay forma de saber

cual es la versión final que debe mantenerse. Para resolver el conflicto, debemos indicar el contenido que quedará como resultado de mezclar nuestro archivo con el que está en el repositorio y recién en ese momento podremos enviarlo al servidor. Ahora que completamos con éxito la actualización del repositorio remoto, cualquier integrante del equipo que esté autorizado tendrá nuestros cambios a su disposición.

Figura 10.2. Ejemplo de uso de un archivo con un SCV.



Una idea sencilla y genial

Basándose en la existencia de los sistemas de control de versiones, hace unos años algunos programadores, entre quienes se encontraba Kent Beck⁸⁶, comenzaron a cuestionarse la utilidad de una etapa de integración en los procesos de desarrollo y plantearon lo siguiente:

Si en vez de ser una etapa, la convertimos en un paso de nuestro trabajo diario, la integración, como la conocíamos antes, desaparece y nos aseguramos que al finalizar el día siempre tendremos en el repositorio la última versión del código integrado. Así dieron nacimiento a la práctica conocida como Integración Continua.

Para entenderla, vamos a describirla como una serie de pasos. Antes que nada vamos a suponer que todo el equipo cuenta con un repositorio común donde se integrará el producto. También supondremos que el equipo conoce el funcionamiento de un SCV, y que cada vez que un integrante escribe código, lo acompaña con sus pruebas automatizadas.

Los pasos son:

- Seleccionar una tarea a desarrollar.
- Actualizar la copia local.
- Codificar la tarea con sus pruebas.
- Ejecutar las pruebas localmente.
- Enviar cambios.
- Verificar la integración.
- Si hay problemas, arreglarlos.

Para poder integrar diariamente, nuestro trabajo tendría que poder completarse en un día o menos. Por este motivo, deberíamos procurar que cada tarea a integrar sea pequeña, idealmente que pueda resolverse en un día o menos. Si la tarea es más grande, sería bueno dividirla en subtareas más pequeñas.

Al actualizar nuestra copia local descargamos la última versión del SCV en el ambiente de desarrollo. Verificamos que el código del repositorio funciona correctamente. Si llegara a darse el caso que tuviese errores, debemos corregirlos antes de comenzar la tarea.

Al codificar, como dijimos arriba, también tenemos que escribir las pruebas unitarias que verifican que el código cumple su función. La integración continua está muy ligada a la escritura de pruebas automatizadas, porque con la existencia y ejecución de estas pruebas es que nos podemos dar cuenta si es que algo no anda bien. Es recomendable que las pruebas sean unitarias y de rápida ejecución, ya que en una aplicación grande puede haber miles de pruebas y deben ejecutarse en un plazo razonable (unos minutos en lo posible) para que la integración continua tenga sentido.

Una vez terminado nuestro trabajo de codificación y escritura de pruebas, nos conectamos al repositorio. Si hay nuevos cambios, los descargamos y nos aseguramos que localmente la aplicación siga compilando y que las pruebas se ejecuten sin errores. Cuando estamos seguros que todo funciona bien, enviamos los cambios al SCV.

Hasta este punto, el trabajo es el que haríamos con cualquier SCV. La diferencia es que cada vez que alguien envía un cambio al repositorio (o cada cierto tiempo), se dispara la ejecución de un proceso de integración que toma los cambios del SCV, y en un ambiente previamente definido y configurado, construye la aplicación y ejecuta las pruebas.

Figura 10.3. Esquema con los pasos de trabajo diario con Integración Continua

El proceso que se ejecuta puede ser un simple script creado por nosotros que descarga los cambios, compila y ejecuta las pruebas, o

también puede ser una aplicación específica de Integración Continua que se instala y se configura para que ejecute el build por nosotros.

Qué es un *build*

Es la acción de tomar todos los fuentes y demás componentes de un sistema y generar la aplicación ejecutable. Durante el build se ejecutan tareas de precompilación, compilación, enlace/ensamblado de objetos en bibliotecas, y también otras asociadas con la verificación del resultado como chequeo estático de código, ejecución de pruebas unitarias, etc. No todas las plataformas de desarrollo relacionan el build con la compilación ya que hay lenguajes que no se compilan como, por ejemplo, los interpretados pero el concepto es siempre generar un resultado ejecutable ya sea en un sistema operativo, máquina virtual o servidor de aplicaciones.

Si durante la ejecución de este proceso ocurre alguna falla, el script o la aplicación que ejecuta el build debe notificar al equipo que el código que está en el repositorio no está correctamente integrado. A partir de ese momento sus integrantes deben corregir los errores de inmediato.

Para que quede claro, *si el resultado del build es erróneo, el foco del equipo debe cambiar para trabajar en solucionar el error y lograr que la integración vuelva a funcionar bien*.

Puesta en práctica

Ahora que conocemos qué es la Integración Continua, intentaremos detallar la manera de implementarla en un proyecto.

Primero necesitamos:

- 1 sistema de control de versiones.
- 1 script con las instrucciones del build.
- 1 servidor de integración.

Para tener nuestro sistema de control de versiones tenemos varias opciones, tanto de código abierto como cerrado, podemos instalarlos por nuestra cuenta y configurarlos en algún servidor o podemos contratar alguna empresa que brinde servicio de control de versiones con lo cual nos evitamos las tareas técnicas y operativas.

El script de build, dependiendo de la tecnología con la que se trabaje, pueden ser archivos que están muy relacionados con la plataforma utilizada (por ejemplo los archivos .csproj que se usan con C# o C con sus makefiles), pueden ser un estándar adoptado por la comunidad (Java con sus build.xml o pom.xml) o puede ser que tengamos que crearlo a mano, escribiendo comandos en un archivo bash (.sh) o de procesamiento por lotes (.bat).

El servidor de integración lo necesitamos para descargar los cambios y ejecutar el build. Allí no debe haber herramientas de desarrollo (IDEs) aunque sí deben existir los comandos y herramientas que son invocadas por el script de build. Por ejemplo, los utilizados para generar el ejecutable a partir de los fuentes, ejecutar las pruebas, etc. Es importante que este servidor esté configurado en condiciones similares al productivo, en relación a como interpreta los datos regionales, sensibilidad del nombre de los archivos, versión del sistema operativo y de las aplicaciones de base (Base de Datos, Máquinas Virtuales, etc.) para evitar que un build exitoso en este servidor se transforme en una aplicación con errores por este tipo de problemas.

Al script de build tendremos que indicarle que se ejecute periódicamente. Si estamos haciendo todo a mano, una opción es crear uno nuevo, que podemos llamarlo de integración. Deberá mínimamente descargar los cambios desde el SCV, ejecutar el script de build e informar el éxito o fracaso (sobre todo el fracaso) del proceso. Una vez armado, podremos agregarlo al planificador de tareas del sistema operativo (como, por ejemplo, el cron en Unix/Linux o el administrador de tareas en Windows). También es posible integrarlo con el sistema de control de versiones para que se ejecute ni bien se realiza una actualización de los datos en el repositorio.

Si no queremos crear otro script, podemos utilizar alguna herramienta de integración continua. Estas existen en modalidad comercial o gratuita, de código abierto o cerrado, instalable o consumible como servicio desde la nube. Si se utilizan, varias de las tareas ya mencionadas pueden delegarse como, por ejemplo, la integración con el sistema de control de versiones o la notificación del resultado.

La ventaja de utilizar una herramienta de integración continua es que brinda muchas facilidades de configuración. Por ejemplo, desde un único lugar se pueden configurar las condiciones de integración de todos los proyectos de la organización o del área. Nos permite, además, integrar con diferentes herramientas que complementan el build como ciertos validadores, notificadores, generadores de documentación y reportes. También trabajan con los sistemas de control de versiones más conocidos gestionando la descarga, el manejo de credenciales de seguridad, la modalidad de chequeo de cambios en el repositorio, etc. Este tipo de software también facilita las configuraciones más avanzadas como, por ejemplo, el manejo de dependencias entre proyectos o módulos de un proyecto forzando la ejecución del build de una aplicación si se completó exitosamente la construcción de alguno de sus de sus componentes. Adicionalmente, generan un espacio homogéneo para controlar los proyectos de una organización dando visibilidad al estado de cada proyecto, dando una idea de la “salud” de cada uno.

Figura 10.4. Pantalla de monitoreo de proyectos de la herramienta Jenkins.

The screenshot shows the Jenkins dashboard. On the left, there's a sidebar with links like 'New job', 'People', 'Build History', 'Project Relationship', and 'check file fingerprint'. Below that is the 'Build queue' section, which is currently empty. To the right is a large table titled 'All' with columns for 'Name', 'Last Success', and 'Description'. The table lists several Jenkins projects:

	W	Name	Last Success
		aplica-le	9 days ago (#81)
		eHector	1 day 1 hr (#103)
		W description	50% Build stability: out of the last 2 builds failed
		Plaid2-hp	7 days 4 hr (#24)
		Plaid-RE	4 mo 22 days (#169)
		Plaid8	3 mo 7 days (#116)
		Plaid-re8	4 mo 12 days (#32)
		B-Hector-Alerta	23 min (#263)
		B-Hector-Alerta-PROD	5 hrs 49 min (#25)
		B-Hector-SONAR	24 min (#150)
		B-Hector-confluence	2 days 7 hrs (#100)

Algunas herramientas de uso común dependiendo la plataforma son:

Ahora que conocemos lo que significa la integración continua podemos ver que no es solo una práctica ágil y puede aplicarse en otro tipo de metodologías, pero la realidad es que el agilismo y la necesidad de entregar valor cada vez más frecuentemente lograron hacer de esta práctica una verdadera necesidad.

Extendiendo la integración continua

Pero vamos un paso más allá. En vez de tratar de tener todo integrado al final de cada día podríamos buscar que toda la cadena de entrega de valor sea un proceso continuo. Como queremos ser ágiles, lo que tenemos que buscar es minimizar el tiempo desde que se toma la decisión de incorporar una nueva funcionalidad hasta estar en condiciones de liberarla en producción.

Veamos entonces un ejemplo de cadena de entrega de valor. El cliente nos pide una funcionalidad, la prioriza y junto con él definimos en qué entrega aparecerá. Acordamos la construcción y las condiciones de aceptación, desarrollamos, automatizamos las pruebas e integramos nuestro desarrollo con el resto del producto. Luego pasamos a un ambiente donde se realizan las pruebas funcionales no automatizadas, más tarde a uno de aceptación en el que el cliente valida que la funcionalidad cumpla con las condiciones de aceptación y, si está todo aceptado, queda lista para incorporarse a la próxima entrega. Una vez que todas las funcionalidades que corresponden a una entrega están completas, podemos avisar que estamos en condiciones de liberarla en producción.

Observando la situación, tenemos al menos, 4 pasajes de ambiente: desarrollo a integración, integración a pruebas, pruebas a aceptación y de allí a producción. En otra época, hacer un pasaje de ambiente era algo relativamente poco frecuente, pero en un enfoque ágil podemos hacer este trabajo decenas o incluso cientos de veces por semana. En cada pasaje pueden generarse errores y, por otro lado, casi siempre estamos haciendo lo mismo. Así que bajo el lema “si algo se repite, se automatiza”, buscaremos programar un proceso que permita ir trasladando nuestras funcionalidades de ambiente en ambiente hasta que lleguen a producción.

Visualicemos las funcionalidades avanzando a través de los ambientes desde que se crean en la máquina del desarrollador hasta que llegan a producción. En cada uno de estos pasos atraviesan distintos chequeos hasta que finalmente pueden ser liberadas en el ambiente productivo. Por ejemplo,

- En el servidor de integración se ejecutarán las pruebas unitarias automatizadas y de integración básicas, también el verificador de estilos y el revisor estático de código.
- En el ambiente de testing se ejecutarán las pruebas funcionales de aplicación y se podrán realizar también las pruebas manuales.
- En el ambiente de aceptación, se cargará la aplicación con datos para que el usuario pueda verificar las condiciones de aceptación y que también puedan ejecutarse las pruebas de aceptación que hayan podido automatizarse.
- El ambiente de producción, se incorporarán solo los cambios que hayan pasado los chequeos anteriores.

A algunos lectores les puede parecer raro tener dos ambientes de pruebas. La cantidad de ambientes varía ampliamente de proyecto en proyecto. En muchas empresas se suele diferenciar el ambiente de testing destinado a las pruebas internas durante la construcción y el ambiente de aceptación donde se instalan las entregas candidatas a producción. En este ambiente son verificados por el cliente todos los criterios de aceptación intentando simular la carga y la configuración productiva.

Esta práctica, que llamaremos Entrega Continua (Continuous Delivery), debe aceitarse para poder alinearla con el concepto del desarrollo ágil. Por eso, Jez Humble y David Farley⁸⁷ proponen ciertos principios que deben tenerse en cuenta a la hora de implementar un proceso de entrega continua:

- Debe ser repetible y confiable.
- Debe automatizarse todo lo que sea posible.
- Debe mantenerse todo bajo control de versiones.
- Si algo genera estado de pánico en el proceso, moverlo hacia el inicio.

- La calidad debe formar parte del proceso.
- Que algo está completo significa que está en producción.
- Todos los integrantes de un equipo son responsables del proceso de despliegue.

A partir de estos principios, se desprenden consideraciones a tener en cuenta como, por ejemplo, el significado de tener todo bajo control de versiones. Este punto implica ahora no sólo el código fuente sino el conjunto:

Código Fuente + Datos de Configuración + Datos del Ambiente + Datos del Programa

Cada cambio en uno de estos componentes debe ser probado y el resultado de esa prueba debe indicar si puede avanzar a la siguiente etapa.

Para facilitar el entendimiento del proceso, Humble y Farley utilizaron el concepto de tubería (pipeline) donde el software va avanzando a través de la misma desde su creación hasta que queda productivo. Cada vez que hay un cambio se genera un build que avanza por la tubería intentando superar las diferentes pruebas y chequeos. En caso de no poder pasárselas, se genera una devolución que explica el motivo del fallo y permite al equipo saber qué cambiar. Una vez corregido el problema, un nuevo build se genera y vuelve a intentar atravesar la tubería hacia el ambiente de producción.

Pero para que todo esto funcione, cada eslabón debería tener un nivel similar de automatización. Si generamos nuestro producto automáticamente pero el pasaje a ambiente de pruebas requiere configuración manual, tendremos cuellos de botella que terminarán afectando toda nuestra línea de trabajo. Por este motivo cuando se hace mención a automatizar todo lo posible significa pensar en programar desde el cambio de archivos de configuración hasta el impacto de cambios en la base de datos, pasando por la generación de distintos tipos de pruebas automáticas (desde unitarias hasta de aceptación, de carga, concurrencia, etc.). Como dijimos, todo lo repetible, en principio, se automatiza.

Es cierto que hay determinado tipo de pruebas que no son automatizables y otras en las que el costo de hacerlo es muy alto. Un ejemplo son las pruebas exploratorias o las de usabilidad. En esos casos, no será posible automatizar, pero debe ser sencillo para el responsable de estas verificaciones indicar si lo que ha probado se comporta correctamente o tiene errores y que a partir de esta acción se pueda pasar a la siguiente fase o descartar el build.

Al tener todo (o casi todo) el proceso automatizado minimizamos la inefficiencia y la alta tasa de errores típica de la operación manual, por lo que se aceleran los tiempos de manera notable. Es tan así que muchas empresas que utilizan la práctica de entrega continua, pasaron de hacer despliegues productivos cada uno o varios meses a hacerlo varias veces *por día*, algo impensado en paradigmas anteriores.

Algunas reacciones

A raíz de la popularización de esta práctica, surgieron algunas dudas en la comunidad de desarrollo. Por ejemplo, a veces un pasaje puede durar varios minutos, o incluso algunas horas. Si cada vez que hacemos un pasaje tenemos que esperar horas y debemos poder hacer varios despliegues por día, mi aplicación estaría fuera de servicio gran parte del día. Por suerte existen técnicas, enfoques de arquitectura y de configuración que permiten desplegar en producción de manera continua sin tiempos muertos, aprovechando las facilidades que dan la clusterización y la virtualización⁸⁸.

Otro cuestionamiento se centró en la necesidad real de impactar los cambios tan frecuentes en producción y de automatizar todo, con el costo que ello implica. Impactar pocos cambios muy seguidos está basado en la idea de *fallar rápido* y es la misma idea que se usó en su momento para generar conciencia sobre la necesidad de realizar pruebas, o de integrar continuamente. Fallar rápido aisla el problema, evita que se construya sobre el código defectuoso, reduce el impacto de los cambios y vuelve el proceso más económico.

Si estamos acostumbrados a fallar rápido, aislar el problema es muchas veces trivial por lo que solo tenemos que concentrarnos en corregirlo. Si hubiéramos obviado probar frecuentemente, la aplicación habría fallado de todas maneras más adelante en el tiempo, quizás en unos días o semanas, con el inconveniente que el código de ese momento será el resultado de múltiples cambios. Al costo de corregir la falla habrá que sumarle el de encontrarla y el de verificar si el código construido posterior a la falla está libre de errores o también debe ser modificado.

Por otro lado, estaban quienes planteaban que este tipo de soluciones solo aplicaba a casos muy específicos y no en un entorno corporativo con aplicaciones de gran tamaño. Afortunadamente, al día de hoy existe evidencia tanto de pequeñas empresas con menos de 10 desarrolladores como de grandes organizaciones con áreas de cientos de empleados que han implementado la entrega continua con éxito⁸⁹.

Ahora bien, es cierto que en muchas empresas medianas o grandes existe un rol o área encargada de la configuración de los ambientes y los pasajes a producción y se podría considerar una restricción para aplicar esta práctica.

Aunque lo mejor sería intentar cambiar la manera de trabajar hacia una cultura de responsabilidad compartida, somos conscientes de que existen muchas industrias donde es muy común encontrar este tipo de barreras, algunas justificadas y otras no tanto. En estos casos, nuestro objetivo debe ser ayudar a minimizar el trabajo manual y reducir la cantidad de pasos para evitar errores y bajar los tiempos. Debemos trabajar con los responsables de estas áreas para que se sientan parte del proyecto y generar en conjunto la mayor cantidad de pasos automáticos de manera tal que solo quede por realizar manualmente lo necesario.

Por último, podemos mencionar a quienes critican la escalabilidad de la entrega continua. Puede ser cierto que implementar un proceso de entrega continua sea más sencillo en un equipo chico que en uno grande. Solo considerando los tiempos de compilación de una gran aplicación contra una pequeña podemos hablar de horas versus segundos. Para que un proyecto pueda crecer exitosamente manteniendo la entrega continua, es importante tenerla en consideración a la hora de definir la arquitectura del sistema. La aplicación debe poder soportar algún tipo de particionamiento como, por ejemplo, el particionamiento funcional, que permita aislar conjuntos de funcionalidades relacionadas, de manera que cada equipo trabaje esa unidad funcional como un módulo, exponiendo y consumiendo servicios a y desde los demás equipos. Es necesario definir una arquitectura que permita ir creciendo pero que, cuando se crece lo suficiente, permita dividir la aplicación para volver a un tamaño donde un despliegue lleve minutos, la cantidad de pruebas automatizadas sea manejable y donde además pueda minimizarse la parte no escalable de nuestro proceso como pueden ser las pruebas manuales.

Versiones vs despliegues

Una duda que puede surgir es cómo se lleva esta práctica de entrega continua de valor con la planificación de entregas. Por un lado,

estamos hablando de entregar regularmente funcionalidades que van pasando por los diferentes estados hasta llegar a producción y, por otro, tenemos un plan donde dice que determinado conjunto de funcionalidades debe salir antes que otro. Para aclarar el tema, sepáremos lo que es una entrega de un despliegue. Una entrega se acuerda a nivel comercial, no es una cuestión técnica. Lo define nuestro cliente, la cúpula de la empresa o el área de marketing, etc. Un despliegue, por el contrario, es un proceso para colocar nuestro producto construido en algún ambiente (en este caso, producción).

Es posible separar ambas. Por ejemplo, podemos entregar continuamente funcionalidades aceptadas a producción, pero diseñar nuestra aplicación para que mediante configuración podamos decidir cuáles estarán habilitadas en producción. Así, cuando la empresa o el cliente indique que liberemos la entrega, solo tendremos que activar las funcionalidades correspondientes. Están en producción desde el primer día en que fueron aceptadas pero solo las habilitamos cuando el negocio las requiere. Si queremos ir más lejos, en sistemas grandes podríamos habilitar selectivamente por país, tipo de usuario, etc. Algunas empresas solo habilitan ciertas funcionalidades a un grupo selecto de usuarios para obtener una devolución respecto a sus sensaciones con el sistema y luego deciden si extienden el uso al resto⁹⁰. Esta separación entre despliegue y entrega facilita la creación de nuestra tubería ya que simplifica muchas condiciones que pueden darse en la etapa previa a producción que están relacionadas con decisiones fuera del alcance del equipo.

Cuando algo no funciona

Así como se plantea una línea de trabajo para ir avanzando de etapa en etapa en la medida que esté todo en orden, es muy importante contar con un mecanismo de rollback automático por si algo falla. Por ejemplo, supongamos que se hace el despliegue en producción de una funcionalidad y que el sistema detecta una falla severa al iniciar el nuevo ejecutable. El proceso de rollback podría deshacer los cambios automáticamente y restaurar la versión original reportando al equipo cuál fue el error.

La entrega continua busca darle importancia al proceso de rollback para que sea tan automático como el de despliegue, pero completar este proceso no siempre es sencillo. Dependiendo del tipo de aplicación, del ambiente y de los cambios que queremos revertir, puede ser que la vuelta atrás de un despliegue sea un comando simple que haga una copia completa del estado del ambiente antes del despliegue (por ejemplo, si el ambiente se encuentra en una máquina virtual puedo hacer una copia snapshot) y lo restaure en caso de que haya problemas. Pero en muchos casos, cuando nuestro ambiente productivo ya se encuentra operativo y un nuevo despliegue altera datos de nuestro sistema, tendremos que tener en cuenta el impacto que estos cambios tendrán a la hora de intentar volver a la versión anterior.

Las aplicaciones están vivas y los datos están cambiando todo el tiempo. Por ejemplo, si en una versión se agregan ciertos estados posibles, al volver atrás debemos determinar qué se hace con los objetos a los que, entre el despliegue y el rollback, les fueron asignados estos nuevos valores. Si se modifican o eliminan, podemos perder información o tener inconsistencias pero tampoco pueden quedar los valores actuales porque tendríamos objetos que representarían estados inexistentes.

Cuando un despliegue implica cambios en el estado de la aplicación pueden generarse casos en los que no sea posible volver atrás automáticamente y que manualmente sea demasiado costoso. Sumado a ello, hay despliegues que pueden requerir ser realizados en paralelo con el de otros sistemas, o que deben lidiar con operaciones ya comenzadas y definir cómo deberán continuar en la nueva versión. Este tipo de situaciones pueden complejizar considerablemente el proceso de rollback.

Por estos motivos, automatizar el proceso que vuelve al estado anterior un despliegue puede ser un verdadero dolor de cabeza. Una correcta arquitectura facilitará el éxito en nuestro intento, aunque no lo garantizará. De todas maneras, independiente del grado de automatización, tener un mecanismo de rollback muestra un grado de madurez interesante del equipo de trabajo y permitirá resolver estos temas mucho más profesionalmente que quienes lo encaren de manera improvisada.

En resumen

Tanto la integración como la entrega continua son prácticas que materializan muy bien los conceptos del desarrollo ágil. Se pueden implementar utilizando herramientas o escribiendo los procesos a mano, pero lo que es realmente relevante es el cambio de mentalidad en el equipo enfocado en mantener el software integrado y que cualquier cambio pueda generar un build candidato a desplegarse en producción.

Siendo relativamente sencillas de implementar, sobre todo la integración continua, permiten organizar el trabajo de manera que los resultados no solo se notan rápidamente, sino que la nueva velocidad con la que los cambios llegan al cliente sorprenden hasta a los mismos desarrolladores. Es difícil que quien se acostumbre a esta forma de trabajar pueda volver a sentirse cómodo con el estilo de integración y entrega tradicional.

⁸⁵ Administración de la Configuración es un concepto que puede verse en otros contextos como, por ejemplo, en el modelo de madurez CMMi, donde es una de las veintidós áreas de procesos.

⁸⁶ La práctica está incorporada en su libro Extreme Programming [Beck 1999]. También puede leerse [Fowler 2006] para entender su origen.

⁸⁷ Para más información véase pág. 48 de [Humble Farley 2010].

⁸⁸ Existe una presentación muy interesante sobre Zero Downtime Deployment de Fabiane Bizinella Nardon que expuso en JFokus 2012 [Bizinella 2012].

⁸⁹ Jez Humble, en una entrevista para la GOTO Conference de 2012 [Humble 2012], menciona entre otros a Facebook y Amazon como casos exitosos de Implementación de Continuous Delivery. Por otro lado, varios autores participamos en proyectos implementando y utilizando esta práctica en empresas grandes, medianas y pequeñas.

⁹⁰ Jez Humble y Martin Fowler dan ejemplos en un podcast de Hansel Minutes [Humble Fowler 2012].

En retrospectiva

Estamos en una fecha cercana a finalizar la iteración 7 que tiene una duración de dos semanas. La revisión con el cliente está planificada para el próximo miércoles a las 9 de la mañana. Históricamente lleva cerca de una hora y media, y se espera lo mismo para esta ocasión. Recién terminó la reunión diaria. Pablo, que en la iteración pasada se definió como el próximo en coordinar la reunión retrospectiva, solicita colaboración del equipo para definir los objetivos y formato de la misma. Y propone los siguientes temas para los cuales encontrar mejores formas de trabajar:

- Documentación del código
- Reglas para la pizarra de tareas
- Comunicación con los usuarios

Para el desarrollo de la reunión sugiere hacer las siguientes actividades: checkin, radar, brainstorming, priorizar con puntos y objetivos SMART⁹¹

Nicolás, que estuvo investigando con Pablo algunas herramientas nuevas y cree que pueden ser de gran utilidad en el proyecto, sugiere cambiar la documentación por algo más abarcativo; y está de acuerdo con el resto. Marcio, que estuvo trabajando con un proceso muy complejo propone analizar como distribuir mejor el conocimiento de temas puntuales. Alejandro sugiere cambiar el 'checkin' que se hizo en la reunión pasada, por una actualización en los acuerdos de trabajo que desde el inicio no se modifican y alguna vez trajo algún conflicto.

Luego de esta conversación, que no se extendió más de 10 minutos, Pablo preparó la agenda y envió el siguiente mail a todo el equipo:

De: Pablo

Fecha: Lunes 09:47hs

Asunto: Retro Iteración 7

Para: Marcio, Carlos, Alejandro, Nicolás, Diego.

Estimados,

Les paso la agenda de la próxima retrospectiva del miércoles próximo de 11 a 13hs después de la revisión con el cliente (en la sala número 2):

Objetivos:

Encontrar formas para mejorar:

- Herramientas de desarrollo
- Reglas en la pizarra de tareas
- Distribuir conocimiento
- Comunicación con los usuarios

Agenda:

1) Repaso de la agenda y los tiempos, acuerdos de trabajo y su actualización -10 minutos

2) Poner información en común sobre los objetivos con un radar 15 min.

3) Brainstorming y causas raíz 45min

¿Qué se hizo bien y qué se puede mejorar y/o experimentar sobre los temas de los objetivos?

Meta a cumplir: 40 ideas entre todos.

4) Priorizar y decidir qué hacer, y quién 25min

Especificar planes (SMART). Definir responsables.

5) Cierre de la retrospectiva breve retro de la retro 15min

¿Qué se hizo bien y qué se puede mejorar del formato de la retrospectiva?

Definir el próximo moderador.

Las reglas de la reunión (que definimos en la primera retrospectiva) son:

- Focalizar la conversación en los objetivos.
- Celulares en silencio.
- Cumplir horario, ser puntuales.
- Aceptar la opinión de cada uno sin juzgar o criticar.
- Evitar buscar culpables.
- Todos los participantes deben cumplir estas reglas y hacerlas cumplir, no solo el moderador.
- Las etapas de 1 a 5 son de tiempo fijo.
- El moderador guarda los resultados de la reunión al finalizar.
- Definir el próximo moderador al terminar.

En la historia anterior podemos ver como se organiza una reunión para tratar los temas que se quieren mejorar, involucrando a todo el equipo, definiendo objetivos claros y estableciendo un tiempo de trabajo concreto, previo al comienzo de la nueva iteración. Se planificaron diferentes actividades para realizar en cada una de las fases de la reunión, y se acordaron reglas de trabajo que seguirán evolucionando.

Lo anterior fue un ejemplo de organización para llevar a cabo una reunión de mejora continua. Sobre estas ceremonias hablaremos en las secciones siguientes.

Introducción a las retrospectivas

La finalidad primordial de este capítulo es guiar a un equipo para buscar mejorar en su trabajo mediante reuniones retrospectivas. Haremos énfasis en su preparación y en especial en su proceso, porque es común que equipos con poca experiencia se reúnan improvisadamente solo con el fin de hacer mejor su trabajo. Así, se encuentran con reuniones sin preparación, aburridas, con acciones que no se llevan a cabo, perdiendo tiempo, preguntándose ¿qué pasó?, ¿por qué salió mal?, terminando en una decepción para los participantes y para el moderador.

En primera instancia vamos a introducir el concepto de retrospectiva⁹² como la forma ágil de aumentar el desempeño del grupo de trabajo en forma iterativa; básicamente es una reunión llevada adelante por el equipo, luego de finalizada una iteración o hito particular, en la que se busca cómo trabajar mejor. Y es importante remarcar que es el propio equipo el que se propone mejoras, y no una imposición de la dirección de la empresa o alguien ajeno.

Solo personas con voluntad, responsabilidad y compromiso con el trabajo se toman el tiempo para esto, lo que es destacable y respetable. Haciendo mención a palabras de *Ron Jeffries*, uno de los fundadores de Extreme Programming (XP), “Hay que mostrar respeto por los esfuerzos de la gente que quiere mejorar por cuenta propia”, por lo que, ante gente con esta buena predisposición, es recomendable que las empresas permitan y alienten este tipo de prácticas.

Las retrospectivas son una excelente oportunidad para lograr una mayor eficiencia en el corto plazo, y aunque requieren cierta preparación y dedicación, la relación costo beneficio de realizarlas es siempre conveniente mientras se hagan bien. Los casos en donde no se tienen buenos resultados se presentan al confundir el concepto y hacer retrospectivas para resolver cualquier problema: por ejemplo, no son para buscar culpables, ni para resolver inconvenientes personales entre miembros del equipo, ni para obligar a alguien a dar explicaciones o forzar a proponer ideas. Tienen otra finalidad. En casos de problemas personales entre miembros del equipo o de bajo desempeño de alguna persona en particular, se deben tratar en privado.

Norman Kerth, uno de los pioneros en el campo de las retrospectivas, propone que todos los participantes adhieran a lo que denominó Primera Directiva que dice así:

“Independientemente de lo que descubramos, entendemos y realmente creemos que todos trabajaron de la mejor forma que pudieron, dado lo que sabía cada uno en ese momento, sus habilidades y capacidades, los recursos disponibles, y la situación en cuestión” [Kerth 2001b].

De esta forma se busca eliminar los miedos a una reunión negativa con quejas y culpas, que claramente no ayudará al aprendizaje.

El marco de trabajo que propone Scrum⁹³ se basa en tres principios: transparencia, inspección y adaptación. En particular, estas últimas dos actividades son las que guían a una retrospectiva para aplicarse sobre la conducta del equipo, sus herramientas, prácticas y toda tarea relacionada con el proyecto y las personas involucradas. Este ritual de la reunión retrospectiva, como lo es en Scrum, llevado adelante correctamente, genera mucho más valor que el que se espera al responder las preguntas: ¿qué hicimos bien y qué podemos mejorar?⁹⁴

Las crisis y las retrospectivas

Durante el transcurso de un proyecto es muy difícil no pasar por problemas, conflictos o situaciones delicadas. Vamos a resumirlos con el término “crisis”, que aplica a todos ellos. Lo que uno generalmente desea luego de una crisis, es que no le vuelva a ocurrir ese malestar, las consecuencias, el sufrimiento o la dolencia; por eso, como una suerte de tercera ley de Newton aplicada –el principio de acción y reacción– ante una crisis, una persona busca una reacción opuesta para evitar que vuelva a ocurrir. En cambio, es poco habitual que uno analice acerca de lo que le pasó y cambie y adapte su comportamiento si no transitó por una crisis. Esa necesidad psicológica que se requiere para mejorar, el sentir la crisis, sentir que uno puede ser mejor, que puede superarse, no a otros, sino a uno mismo, con su equipo. Eso es lo que lo impulsa a uno al cambio.

Jeff Sutherland, cocreador de Scrum decía: “Necesitamos una ‘mente kaizen’⁹⁵, un sentido interminable de crisis en la compañía para impulsar constantemente a mejorar”.

Así también, es oportuno conocer al menos una parte de las palabras que se le atribuyen a Albert Einstein sobre las crisis:

“No pretendamos que las cosas cambien si siempre hacemos lo mismo. La crisis es la mejor bendición que puede sucederle a personas y países porque la crisis trae progresos. La creatividad nace de la angustia como el día nace de la noche oscura. Es en la crisis que nace la inventiva, los descubrimientos y las grandes estrategias. [...] Es en la crisis donde aflora lo mejor de cada uno, porque sin crisis todo viento es caricia. Hablar de crisis es promoverla, y callar en la crisis es exaltar el conformismo. En vez de esto, trabajemos duro. Acabemos de una vez con la única crisis amenazadora que es la tragedia de no querer luchar por superarla”.

Las retrospectivas son una buena forma de responder ante la crisis. Veamos entonces su origen para luego entrar en el proceso y poder llevarlas adelante con un mejor entendimiento.

El origen

¿Por qué son ágiles las retrospectivas? Para entender de donde surge la idea hay que remitirse y analizar el último principio del manifiesto ágil, que expone lo siguiente:

“A intervalos regulares el equipo reflexiona sobre como ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia”.

Hagamos un breve análisis. Comparemos primero la idea de los intervalos regulares con los que tradicionalmente se hacen, en donde las reuniones de lecciones aprendidas se llevaban a cabo al finalizar un proyecto, con el fin de incorporar los resultados a una base de conocimiento y tenerse en cuenta para futuros proyectos dentro de la organización. De esa forma, las mejoras detectadas no permitían a un equipo crecer durante el proyecto. Recién se aprendía a mejorar cuando ya se había concluido el trabajo y, en muchos casos, desintegrado el equipo, lo cual es problemático en el caso de empresas orientadas a proyectos⁹⁶. Tampoco tiene sentido dedicarle tiempo a analizar el comportamiento del grupo específico de personas en donde hay pocas garantías de que se mantenga unido.

El enfoque ágil es diferente: las personas son lo más importante, cada equipo de trabajo tiene su propia eficiencia y necesita analizarse dentro de ese contexto; esto no puede pasarse por alto. Es irracional tener que esperar a otro proyecto para incorporar nuevas

prácticas, herramientas, habilidades, métricas, mejores formas de comunicación, otras maneras de hacer más eficiente o divertido el trabajo. El proyecto está activo, es hoy, hay que actuar en el corto plazo. No podemos perder la oportunidad de trabajar mejor, de entregar mayor valor, de ser más eficientes, de trabajar más orgullosos.

Tenemos que dedicarle tiempo a analizar como se está trabajando y poder mejorar de manera iterativa, porque lo que es conveniente hoy no necesariamente lo es mañana.

Comparemos y pongamos de ejemplo un proyecto de 6 meses. Tradicionalmente se realizaba una única sesión de lecciones aprendidas donde ya no se tendría la posibilidad de aplicar lo aprendido al proyecto post mortem; contra 12 retrospectivas de un proyecto ágil, si se hacen iteraciones de 2 semanas.

Asimismo, un estímulo para la motivación de cualquier persona es trabajar en un entorno que le permita buscar e implementar sus propias formas para mejorar y superarse, propias del equipo y no impuestas por otro sector, una gerencia o dirección.

El proceso

Como ya comentamos, si conducimos una reunión retrospectiva sin preparación y solo con idea de que los participantes respondan qué hicimos bien y qué podemos mejorar, los resultados y el desarrollo de la reunión no serán los mejores.

Para organizar una retrospectiva, consideramos acertados los lineamientos que proponen Esther Derby y Diana Larsen en su libro *Agile Retrospectives, making good teams great* [Derby Larsen 2006]. Básicamente es un marco de trabajo compuesto por cinco partes, sobre las cuales nos apoyaremos para dar nuestra visión, opinión y otras alternativas que nos resultaron convenientes:

1. Establecer el escenario.
2. Reunir datos.
3. Generar entendimiento.
4. Decidir qué hacer.
5. Cierre.

Establecer el escenario

Este primer paso es para poner a todos los participantes en contexto (no en uno de información, esa es la segunda parte, sino en un contexto de focalización). Al no ser una tarea fácil, habitualmente, es una etapa que no se hace y saltea. Así se comete un error que se paga en las fases siguientes y se potencia a medida que es mayor la cantidad de integrantes.

Hacer foco significa dejar de lado distracciones (celular, notebooks, charlas con el compañero), concentrarse en ese tiempo y ese lugar para trabajar de manera sinérgica y cumplir con los objetivos de la reunión. Para esto es recomendable, en esta etapa de arranque, repasar la agenda de la reunión, las partes de la retrospectiva con sus tiempos, las técnicas y herramientas a utilizar, las reglas del equipo y algún ejercicio para romper el hielo. Es el puntapié inicial para lograr un clima grato en el que todos sientan ganas de participar.

En esta primera etapa se recomienda repasar las reglas autoimpuestas por el equipo que se deberán respetar durante la reunión. Si no existen, se pueden crear, y si las tienen, se pueden actualizar. Luego del acuerdo, no son opcionales. Todos deben cumplirlas y hacerlas cumplir, no solo el moderador. Un ejemplo puede ser el siguiente:

- Ser puntuales, cumplir con el horario planificado.
- Si bien se espera participación de todos, nadie está forzado a hablar; si alguien tiene mucho para contar, tampoco debe dominar la sesión.
- Focalizar la conversación en los objetivos.
- Celulares en silencio, monitores apagados.
- Aceptar la opinión de cada uno sin juzgar ni criticar.
- No buscar culpables.

Es recomendable imprimirlas o escribirlas en un costado de una pizarra para repasarlas al inicio de la reunión y que todos las puedan ver durante las retrospectivas: esto hace más fácil su cumplimiento.

Una de las tantas actividades que se pueden utilizar para entrar en clima y animar a todos a participar es el “checkin”. Consiste en que cada uno responda una pregunta en una o dos palabras, con el fin de entrar en confianza y permitir e incentivar la participación de todos, como se postuló en las reglas básicas. Por ejemplo: en muy pocas palabras, ¿qué esperas de esta retrospectiva? o ¿cómo te sentiste al finalizar la iteración pasada? o, para cada objetivo de la retrospectiva, ¿qué palabra describe el estado actual de cada uno? Otra posibilidad para responder es seleccionar entre opciones, por ejemplo: feliz, mejor, enojado, triste, disconforme. Tanto la pregunta como las opciones deben prepararse previamente a la reunión.

El “checkin” especialmente en equipos grandes resulta muy eficiente, se hace muy rápido, y da ánimo para empezar a participar. Crea un clima de grupo y permite entender el estado de ánimo al escuchar las voces de cada uno. Solo con este ejercicio se perciben las diferentes sensaciones de los participantes sobre el tema. Se mantiene la atención en el que está hablando y esperando el propio turno, haciendo foco en la reunión. Ese es el objetivo que queríamos cumplir.

Pero hay diversas actividades que se pueden utilizar en esta fase e incluso alguno con espíritu innovador pueden inventar alguna.

Ahora sí, repasamos la agenda, la estructura de la reunión, los tiempos, las reglas básicas y un ejercicio de participación; ya podemos pasar a la segunda parte.

Reunir datos⁹⁷

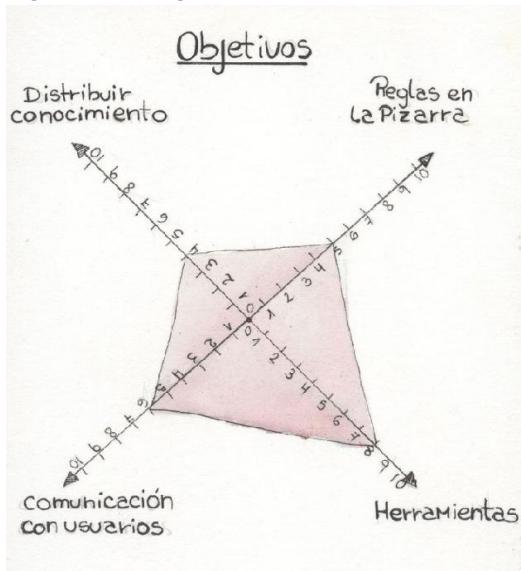
Esta etapa puede parecer innecesaria porque uno tiende a creer que todos saben lo mismo sobre la iteración que acaba de finalizar. Pero no es así. Pongamos un ejemplo, en un equipo que recién terminó de hacer una revisión de una entrega importante con el cliente, el facilitador puede estar contento, y suponer que el resto también lo está, porque el cliente estuvo muy satisfecho con los incrementos en el producto, pero otro miembro del equipo puede estar molesto porque no hizo más que desarrollar código para la capa de presentación que no es de su agrado, y otro disconforme porque el software para el seguimiento de las tareas no es rápido ni provee toda la información necesaria para un mejor control.

Por lo tanto, no hay que suponer que todos cuentan con la misma información. Es clave, para mejorar, recolectar los hechos y los sentimientos, compartirlos y trabajar sobre todo el conjunto.

Se pueden hacer comentarios, escribir sobre la pizarra, utilizar papeles autoadhesivos, u otros métodos para recolectar información. Utilizaremos, a modo de ejemplo, la técnica del radar. Siempre haciendo foco en los objetivos. La técnica de “checkin”, cuando se hace sobre los objetivos, proporciona de forma muy rápida una visión cualitativa sobre lo que piensa cada participante sobre los temas que son propósito de la reunión. Por ejemplo, ¿qué te pareció la metodología en una palabra? y las respuestas: “bien”, “mejor”, “inadecuada”, “útil”. En cambio, la técnica del radar nos da información de manera cuantitativa, más precisa, cuánto mal: “4”, cuánto mejor: “8”, cuánto inadecuada, “3”, cuánto útil “10”. Primero se dibuja el radar en blanco, con los temas a analizar en cada línea, se explica la escala a utilizar, por ejemplo: de 0 a 10, números enteros, donde cero es lo mínimo, el peor valor, y 10 el máximo, la mejor puntuación.

Se puede hacer de dos formas. La primera, como una dispersión de puntos sobre el radar, donde cada persona dibuja un punto alrededor del número que elija para cada una de las líneas de temas. Funciona para equipos pequeños y medianos. En la segunda forma se tiene en cuenta solo el promedio. Se recolectan los valores de todos los participantes para cada uno de los temas y luego se marca sobre el radar el promedio para compartir los resultados y evaluar la medición conjunta. Esta forma es anónima, cada persona debe escribir en un papel el puntaje para cada tema y entregarlo al moderador que saca el promedio y anota los resultados. Esta segunda opción funciona para cualquier cantidad de personas aunque es especialmente recomendable para grupos numerosos. Es conveniente disponer de una calculadora y un colaborador que dicte los valores para sacar el promedio.

Figura 11.1. Diagrama radar.



Una vez marcado el promedio, se pueden unir esos puntos con líneas y pintar el interior para resaltar más el área. Los valores visualizados son los que surgen del equipo en conjunto que deberá proceder a analizar, enfatizando y celebrando los puntos fuertes y el área cubierta, y detectando posibilidades de mejora para las regiones no cubiertas.

Si el equipo no es muy numeroso y el tiempo para la retrospectiva no es muy ajustado, para complementar esta etapa, cada participante puede agregar notas adhesivas con información sobre cada tema para generar una imagen global diversificada que contenga todos los puntos de vista.

El radar fue solo un ejemplo porque existen numerosas actividades que se pueden hacer en esta fase. Otra técnica, mucho más simple, es preguntarse ¿qué hicimos bien? Y ¿qué podemos mejorar? Hay que destacar que en la segunda pregunta se hace foco en la mejora y no se formula desde una visión negativa. Por eso es importante que el moderador explique que esos temas hay que verlos como algo positivo porque el equipo pretende modificar para ser mejor. Se debe tener en cuenta que en esta fase hacemos foco en el “qué”, ya que el “por qué” y el “cómo” se analiza más adelante.

Lo importante que queremos destacar aquí es poner la información en común, para poder trabajarla de fondo entre todo el equipo en la fase siguiente.

Generar entendimiento

En esta etapa el equipo debe evaluar los datos de la fase anterior y preguntarse por qué esos temas no anduvieron bien y así llegar a encontrar las causas raíz de los problemas y proponer cambios. También se trabaja sobre los riesgos y se busca reconocer patrones que llevaron a hechos positivos y negativos.

Varias son las actividades que se pueden realizar en esta fase y no es el objetivo entrar en detalle. Lo importante es comprender la esencia de esta etapa, por lo tanto, solo veremos unos ejemplos.

La actividad más común es el Brainstorming. La intención es generar muchas ideas sobre los objetivos de la reunión. Una buena forma es poner una meta de mínimo (por ejemplo: 40 ideas en el tiempo planificado).

Se puede hacer en un formato libre, en donde cada uno participa a medida que lo desee, o de manera ordenada comenzando por una persona seguida por el contiguo. El formato libre es más común, pero puede dejar fuera a gente con menor tendencia a participar en público, y así perder importantes ideas. En ese caso, el moderador puede invitar a participar a quienes no lo han hecho o cambiar al otro formato.

Es una buena práctica también comenzar con unos minutos para pensar en silencio. De esta manera cada uno va anotando sus ideas, para luego empezar a exponerlas sabiendo que todos al menos disponen de algunas.

Las ideas se escriben resumidas en notas adhesivas y se pegan en una pizarra, donde cada persona la explica brevemente y la coloca a la vista de todos. De acuerdo con la cantidad de gente o conveniencia, estas últimas tareas las puede hacer el moderador.

El moderador o los mismos participantes pueden agrupar las ideas según tengan una afinidad, relación o patrón. Esto también va a servir para priorizar mejor en la fase siguiente.

Uno de los beneficios de esta técnica es la posibilidad de construir nuevas ideas sobre ideas de otros. De no ser así, muchas nunca van a ocurrir. Es aconsejable también, previo al inicio de la actividad, que el moderador mencione y anote en la pizarra las reglas del Brainstorming. Un ejemplo podría ser:

- Mínimo 40 ideas.
- Todas las ideas son buenas.
- No filtrarlas.
- Construir sobre ideas de otros.
- No criticar.
- Se espera participación de todos.

Otra técnica que funciona como complementaria es la de los cinco por qué⁹⁸. Consiste en preguntarse varias veces el por qué algo está ocurriendo para así llegar a la causa raíz en la quinta respuesta como máximo. Se puede utilizar sobre los resultados de alguna actividad de la fase anterior o sobre el Brainstorming, especialmente en los puntos a mejorar. Pongamos un ejemplo: “esfuerzo adicional por cambios frecuentes en la interfaz de usuario”, nos preguntamos:

1) ¿Por qué hay cambios frecuentes en la interfaz de usuario? Porque cuando el cliente ejecuta el sistema en la demo casi siempre pide cambios.

2) ¿Por qué no propone los cambios cuando se valida la funcionalidad antes de su desarrollo?

Porque las historias de usuario y los criterios de aceptación en general están bien, siempre pide cambios en tipos de controles, tamaños, imágenes, navegación.

3) ¿Por qué no puede validar los controles, tamaños, imágenes?

Porque no tiene prototipos para hacerlo, solo el sistema al terminar la iteración.

4) ¿Por qué no hacemos prototipos de pantallas? Porque no los tuvimos en cuenta en las tareas ni en la estimación en la reunión de planificación. ¡Incluyámolas para la próxima iteración!

Ya identificados los problemas con sus causas raíz, podemos pasar a la próxima fase y establecer acciones concretas para mejorar.

Decidir qué hacer

En esta etapa hay que determinar las tareas, acciones o experimentos resultantes de la etapa anterior que se van a llevar a cabo durante la próxima iteración, quién y cómo lo va a hacer. En el caso particular de que sea una retrospectiva de fin de proyecto serán cambios para documentar e implementar en nuevos proyectos.

Es una fase que consta de una primera parte de priorización y una segunda de planificación de los pasos a seguir para cada acción de mejora. En la priorización, el equipo debe decidir que propuestas se consideran más importantes y las que más se desean llevar a cabo, teniendo en cuenta la relación costo beneficio.

Una actividad para votar las preferencias es la de priorizar con puntos que cada participante dispone para asignar a las propuestas. La cantidad de votos debe ser un número fijo, que en general va de un tercio a la mitad de la suma del total de las ideas a escoger. Cada uno va asignando puntos a su gusto sobre la pizarra sin pasarse de esa cantidad. Se puede hacer una restricción adicional; por ejemplo, no poner más de dos votos sobre una misma idea.

A la hora de tomar las propuestas que se identificaron como las prioritarias en orden para ejecutar, es importante no sobrecargar al equipo, haciendo solo algunas. Si se seleccionan demasiadas difícilmente se puedan cumplir. Hay que tener en cuenta que estas tareas se sumarán a todas las de la propia iteración para seguir construyendo sobre el producto. Si se seleccionan demasiadas actividades de mejora y luego solo se pueden llevar a cabo algunas pocas, el equipo en cierta forma se frustra y se genera una sensación de desconfianza en las retrospectivas. Por eso hay que elegir pocas o alguna estrategia que permita realizarlas, por ejemplo, una para cada integrante.

Ahora bien, una vez que tenemos la lista priorizada, no conviene quedarse solo con las ideas de mejora, sino que hay que llevarlas a acciones concretas. Esto va a ayudar a que sea más probable que se cumplan. Por ejemplo, utilizando la técnica de objetivos SMART (por sus siglas en inglés) donde se busca que los objetivos sean: específicos, medibles, alcanzables, relevantes y oportunos. Por ejemplo, no es lo mismo quedarse con la propuesta: “hay que automatizar pruebas funcionales”, que definir en conjunto con el equipo: “Diego automatizará las pruebas del módulo 1 y Carlos las del módulo 2 durante la semana próxima. Se utilizará la herramienta Selenium. El martes se realizará una demostración interna para evaluar los resultados”.

Finalmente, las acciones priorizadas y detalladas se deben planificar al comienzo de la siguiente iteración al igual que se hace con las *user stories*. Las acciones remanentes con menor priorización no se deben descartar, sino que se pueden tener en cuenta en la próxima retrospectiva⁹⁹.

Cierre

La fase de cierre se centra básicamente en hacer un muy breve análisis de la retrospectiva, guardar los resultados de la reunión y agradecer el trabajo realizado.

La actividad más utilizada para evaluar la retrospectiva consta en que el equipo debe indicar qué cosas estuvieron bien y las que se pueden mejorar para la próxima reunión. Pueden ser críticas constructivas sobre el formato, los tiempos, las actividades realizadas o cualquier otro factor. El ejercicio se puede hacer sobre una pizarra dividiéndola en dos, y el equipo va agregando las opiniones para estas dos categorías en forma ordenada o libre.

Al ser el cierre de la etapa final, y por lo tanto, la candidata a sufrir algún recorte de tiempo si alguna de las fases anteriores se extendió, se puede practicar una alternativa en el caso de que el equipo sea relativamente pequeño, llevando a cabo el ejercicio de pie rodeando la pizarra. En general, de esta forma se acelera el procedimiento y el equipo se siente cómodo por ser una práctica utilizada en las reuniones diarias. Es importante que el moderador anote las sugerencias.

Para finalizar y luego de los agradecimientos por el esfuerzo y la participación a todo el equipo, no hay que olvidar tomar una fotografía de las pizarras con los gráficos y tareas resultantes de todo el proceso.

Recomendaciones complementarias

Si bien en las secciones anteriores fuimos describiendo prácticas que consideramos recomendables, a continuación complementaremos algunas ya vistas y aconsejamos otras que ayudarán a lograr una mejor retrospectiva.

Ambiente seguro

Para obtener una buena participación, y por lo tanto, buenos resultados, es necesario propiciar un ambiente seguro y cómodo pues durante la reunión retrospectiva no se están haciendo evaluaciones de desempeño. Tampoco se hace para reportar a una gerencia o dirección. Es un momento y un lugar en donde el equipo puede discutir libremente.

Lugar cómodo y diferente

Es recomendable salir del puesto de trabajo habitual para escapar a la rutina y crear un ambiente más recreativo. En general para las retrospectivas no es necesario utilizar grandes mesas, sino que es más conveniente orientar la disposición de las sillas en forma semicircular en donde todos tengan acceso a la pizarra, ver al resto de los participantes y escucharse sin dificultades.

Comida

No estamos hablando de un gran banquete que invite a que la gente se disperse, pero unas galletitas acompañadas de café o alguna otra infusión a gusto, son un buen comienzo para llevar adelante una agradable y positiva reunión.

Alternar el moderador

Es una buena práctica rotar al moderador, no siempre tiene que ser el que tiene el rol de facilitador del equipo, siempre y cuando haya otros voluntarios. Es un factor motivante tener la posibilidad de tomar el rol para quien le interese y, por otro lado, hace más divertida la reunión.

Nuevas actividades

Es aburrido ir a reuniones predecibles donde siempre se hace lo mismo. Por eso, no es recomendable repetir siempre el mismo formato más de dos o tres veces con el mismo equipo de trabajo. En las secciones anteriores se comentaron algunas actividades que se pueden hacer en cada una de las cinco fases de una retrospectiva¹⁰⁰. Hay que animarse a probar nuevas alternativas.

Ayudar a la imaginación

Dependiendo del tipo de retrospectiva y la dinámica con que viene trabajando, puede ser conveniente que el equipo llegue con una mínima preparación a la reunión, una lista de temas o ideas pensadas y anotadas. Luego de una larga iteración, o si no se hacen retrospectivas a menudo, a veces se olvidan los temas que fueron sucediendo y no se recuerdan rápidamente durante la reunión. Por ese motivo se puede ayudar a recordar si se utiliza una lista de categorías o áreas en las que se puede hacer foco, por ejemplo: herramientas, comunicación, documentación, cliente, proceso de desarrollo, testing, hardware, calidad, riesgos, alcance, metodología, capacitación, configuración. El listado se debe enviar a todos los participantes en conjunto con la agenda, en especial si el objetivo es libre.

La mejora es día a día

La retrospectiva no debe ser un espacio para la resolución acumulada de problemas. Ante la aparición de un inconveniente o impedimento, e incluso mejoras, se debe tratar de resolver a tiempo. En caso contrario, el impacto puede ser mayor. Por otro lado, si surgen muchos problemas que resolver, seguramente no se solucionarán todos en la próxima reunión retrospectiva. La mejora continua no sólo se da en estas reuniones, sino que se construye día a día.

Moderador externo

Hay ocasiones en que es conveniente que el moderador no sea parte del equipo de trabajo, ya que puede generar influencias o estar influenciado por otros participantes. Esto puede ocurrir cuando forman parte de la reunión personas con diferentes puestos. Por otro lado, la participación de un moderador externo puede resultar motivante y entretenida en casos de proyectos con muchas iteraciones, en donde esa tarea siempre la hicieron las mismas personas.

Usar retrospectivas anteriores

Como parte de la agenda, se puede destinar un tiempo para ver el estado y los temas de retrospectivas anteriores. Casi nunca se llegan a resolver todos los puntos de acción en la iteración siguiente en que se detectaron, y ese trabajo no hay que desaprovecharlo.

Preparar la reunión

Organizar la reunión, preparar los materiales, enviar la agenda. Todo esto es un buen signo de respeto por el tiempo de los demás. Asimismo, una mejor preparación va a dar mejores resultados.

Trabajo en equipo

Los objetivos se deben preparar en conjunto con el equipo de trabajo antes de enviar la agenda. Consultar a los participantes si les parece bien el formato y las actividades propuestas es parte de la tarea del organizador. Desde ya que pueden surgir cambios y hay que estar dispuesto a recibirlas y adaptarlos.

Resumen

Las retrospectivas representan una instancia de mejora continua, la más formal y poderosa, en el sentido de que es una reunión que requiere preparación, sinergia de todo el equipo y debe ejecutarse entre iteraciones.

Para lograr resultados excelentes es esencial entender cada una de las partes que conforman la reunión y preparar las actividades a realizar, sin saltar etapas.

Es importante tener en cuenta varios factores para organizar y llevar adelante la reunión: las características del equipo y su aporte en la preparación, las emociones, el tamaño de las iteraciones, un ambiente seguro, no buscar culpables, no criticar ideas, rotar el moderador, definir reglas; son algunos de los ingredientes básicos.

⁹¹ Del inglés: Specific, Measurable, Attainable, Relevant, Timely.

⁹² El término ‘Retrospectiva’ fue inicialmente utilizado por Norman Kerth en Project Retrospectives: A Handbook for Team Reviews [Kerth 2001a] y se hizo popular y aceptado por la comunidad con el uso de Scrum.

[93](#) Véase Apéndice: La riqueza de la diversidad en el cual se explica en más detalle Scrum.

[94](#) En [Schwaber 2004] donde Ken Schwaber explica Scrum y define una reunión retrospectiva, indica que se deben responder esas dos preguntas de la última iteración o Sprint.

[95](#) Kaizen: “cambio a mejor” o “mejora” en japonés, el uso común es “mejora continua”.

[96](#) [PMBOK4] pag. 28, sección “2.4.2 Estructura Organizacional” se definen tres tipos: Funcional, Matricial y orientada a Proyectos. En estas últimas las personas no tienen un lugar al que pertenezcan una vez terminado el proyecto, sino que quedan a la espera del siguiente. A diferencia de las funcionales, en donde, por ejemplo, un área de Marketing, permanece más allá de un proyecto.

[97](#) Una buena práctica es aprovechar este momento para evaluar las acciones de mejora comprometidas en la retrospectiva anterior y evaluar si se hicieron o no y cuales fueron los resultados obtenidos.

[98](#) Esta técnica de análisis de causas raíz puede complementarse con un diagrama de espinas de pescado o diagrama de Ishikawa.

[99](#) Puede considerarse y administrarse como un backlog de mejoras, o incluso incorporarse al backlog del proyecto.

[100](#) Para profundizar en las actividades y detalle de cada una véase *Agile Retrospectives, making good teams great* [Derby Larsen 2006]

Reuniendo al equipo

Son las 10 de la mañana menos un minuto, un miércoles en plena iteración de un proyecto en curso. El equipo, incluyendo al líder de proyecto y el representante del cliente, están juntos para la reunión diaria.

Diego, el líder de proyecto se dirige al equipo:

– ¿Estamos todos aquí?

– Falta Alejandro –contesta un miembro del equipo– Llamó hace un rato para avisar que estaba enfermo, me comentó de su avance de ayer, y dijo que más tarde intentará conectarse para ver si puede hacer algunas tareas desde su casa.

– Gracias –responde Diego– pero tratemos de no molestarlo salvo alguna pregunta muy puntual; es mejor que descansen y se recupere. Comencemos.

Uno a uno, de pie frente a la pizarra con el estado del proyecto, en sentido horario, los miembros del equipo comentan sobre su avance de ayer, lo que planean realizar hoy, y los impedimentos que están encontrando.

Cuando llega el turno de Carlos, comenta sobre un impedimento técnico y José le responde:

– Yo tuve un problema similar en mi anterior proyecto. Lo que encontré es que...

Diego lo interrumpe:

– Si les parece, podrían juntarse durante el día para ver esto. ¿A alguien más le parece que puede aportar y dar una mano?

Marcio comenta:

– Yo vi algo parecido; me sumo y lo vemos.

– Bárbaro, gracias a los tres. Sigamos.

Así todos comentan su avance. Diego toma nota de los temas que surgieron y de los que el equipo debe encargar se, incluyendo el hecho de la ausencia de Alejandro. Luego ingresará estos ítems en la planificación para analizar el impacto. Todo el equipo está ahora al tanto de la situación de los demás, y quienes tienen algún problema para avanzar ya han informado sobre ello. Son las 10.15 y la reunión ha terminado.

En este breve ejemplo podemos ver algunas características esenciales de reuniones exitosas: se comenzó y terminó a tiempo, se abordaron los temas previstos; también surgieron temas adicionales que no afectaban a toda la audiencia a los que se les asignaron responsables y se acordó abordarlos fuera de la reunión. Además, hubo un orden implícito y simple: cada uno sabía sobre qué y cuándo debía hablar y la información fluyó hacia todos los integrantes del equipo.

En este capítulo veremos algunas características de las reuniones de proyectos ágiles y algunas prácticas que pueden aplicarse a todo tipo de reuniones.

Pautas básicas para reuniones ágiles

Time Boxing

Así como los proyectos ágiles se dividen por iteraciones, con una duración fija, las reuniones deben tener un tiempo establecido y no sobrepasarlo. Para esto es clave la puntualidad, cosa no siempre fácil por razones culturales. Por eso conviene prestarle atención al tema del tiempo.

Para lograr esto resulta fundamental que los temas a tratar en una reunión sean claramente conocidos por todos y que los participantes estén atentos para evitar desvíos en los objetivos que, de producirse, llevarán a que la reunión se extienda.

En un proyecto ágil, cada integrante es responsable de administrar su tiempo y cumplir con sus objetivos. Tener al equipo en reuniones largas e improductivas (o al menos, poco eficientes) puede interpretarse, y no sin razón, como una falta de consideración a las tareas de los integrantes. Si esto se hace habitual, no tardarán en surgir malestares en el equipo.

Cuando surge un tema importante que extenderá la reunión, lo mejor es agendar otro momento para tratar ese tema en particular, de manera de cumplir con los objetivos originales. Si las reuniones son periódicas, finalizarlas al cumplirse el tiempo establecido, se haya terminado o no, ayuda a aprender a ejecutarlas mejor la siguiente vez.

Si la reunión llega a su tiempo límite y no se cubrieron todos los temas planificados, no se debería extender como si no hubiera pasado nada. Lo mejor es preguntar a los asistentes si prefieren continuar o concertar una nueva reunión. No hay que dar por supuesto que quienes están en la reunión pueden quedarse más allá de lo planificado: estaríamos despreciando implícitamente el valor de su tiempo.

Estructura de las reuniones

Las reuniones que se realizan en forma periódica tienen una estructura y objetivos bien definidos. Esto se debe explicitar a los participantes para que sean más efectivas.

Un buen momento para explicar la estructura es durante la reunión de inicio del proyecto. Luego, en la primera ocurrencia de cada reunión en particular, se debería repasar la estructura en mayor detalle.

Hay reuniones que, por definición, pueden tener una estructura que cambia cada vez que se las realiza. Un ejemplo de ello son las retrospectivas. En este caso se deberán reservar unos minutos al principio para explicar la dinámica o repasarlas en caso de haber enviado una agenda previa con una explicación de las actividades.

Lo más importante en cuanto a la estructura de las reuniones en un proyecto ágil es que deben ser simples. Nuestro objetivo es construir software de calidad, no tener reuniones lo más ordenadas y documentadas posibles, donde la gestión se convierte en un fin en sí mismo.

Documentación

En muchos proyectos es común que luego de cada reunión se prepare una minuta. Si bien esta es una práctica establecida, la minuta no es un artefacto propio del desarrollo ágil. Esto no significa que no se pueda usar, sino que no siempre tiene que existir si hay otro elemento que la sustituya.

Un ejemplo son las reuniones cuyo objeto queda plasmado en un documento: en una reunión de planificación, el resultado es el plan para la iteración y muchas veces con eso alcanza para documentar lo hecho.

Cuando es necesario documentar lo hablado y acordado, debe primar la utilidad de esta documentación para el equipo. Muchas veces

un correo electrónico con una estructura fija pero mínima (tanto en el asunto como en el cuerpo), detallando temas tratados, acciones comprometidas, responsables de las mismas y acuerdos realizados, es más que suficiente. Una buena práctica es colocar todo esto en el cuerpo del email, o si alguna norma externa obliga a colocar un adjunto, al menos mencionar en el cuerpo del email los temas principales. Es más probable que lean el texto del cuerpo a que abran el adjunto.

En las reuniones técnicas es común que se realicen diagramas en una pizarra y luego en lugar de transcribirlos a formato digital, simplemente se tome una foto de la pizarra donde fueron dibujados.

Tipos de reuniones en el desarrollo ágil

Reunión diaria (daily standup meeting)

Esta es quizá la reunión emblemática del agilismo y, por definición, la que más veces se realiza durante un proyecto. De hecho hay muchos proyectos que se creen (equivocadamente) ágiles porque se reúnen a diario y todos comentan su avance.

En estas reuniones el equipo se reúne y cada integrante usualmente responde a 3 preguntas respecto a su trabajo:

¿Qué hice ayer?

¿Qué haré hoy?

¿Qué impedimentos tengo para realizar mi trabajo?

Dada su frecuencia, es imperativo que esta reunión sea corta. Al estar reunido todo el equipo, el tiempo que puede ahorrarse tiene un efecto multiplicador enorme.

Hay quienes recomiendan hacer estas reuniones de pie y no sentados (de hecho en muchos casos se las llama Daily standup), tanto para ayudar a que la reunión sea corta, como para aprovechar la energía asociada al estar todo el equipo parado.¹⁰¹

Es habitual realizarlas por la mañana así se tiene fresco en la memoria (y con algo de perspectiva) el avance del día anterior y al tener que explicitar los objetivos de la jornada, se arranca con foco en las tareas a realizar.

Pueden existir motivos para hacerla en otro momento. Es posible que quien tiene el rol de facilitador tenga una idea del estado del proyecto al final de la jornada. Esta no parece una decisión en la dirección correcta: el objetivo del proyecto no es reportar el avance, sino desarrollar software. El reporte de avance es necesario, pero debe amoldarse a las necesidades del equipo, aunque le implique un esfuerzo mayor.

En estas reuniones no es necesario tener todo lo dicho plasmado en una minuta. Muchas veces lo más importante es documentar¹⁰² los impedimentos para poder tomar acciones, los acuerdos y las nuevas reuniones que surgen de lo tratado.

Reunión de planificación

Estas suelen ser las reuniones más largas, y en el caso de proyectos de iteraciones con longitud mayor a dos semanas pueden incluso superar las dos horas. En estos casos es recomendable tomar un descanso cada tanto para no perder efectividad en las actividades que se realizan. Los descansos no tienen que prefijarse (no es bueno interrumpir el flujo del trabajo cuando se está concentrado), sino más bien que cualquiera tenga la libertad de sugerir hacer una pausa cuando se siente cansado o nota cansancio en el equipo.

El resultado se plasma en el plan de iteración y la modificación de los demás planes. No debería ser necesario una minuta al final de esta actividad.

Revisión de la iteración

Esta es la más formal de las reuniones. Involucra a todo el equipo, incluyendo al cliente y en muchos casos a los stakeholders. Es una instancia importante del proyecto, pues se muestra el trabajo hecho al cliente.

Es muy importante que el objetivo sea claro, de ser posible detallar de antemano los temas a tratar, de forma que no haya sorpresas. Es también de las pocas reuniones en las que es recomendable practicar de antemano lo que se mostrará y verificar el tiempo necesario para realizarla.

Conviene repasar la estructura de la misma en su comienzo, en especial si hay participantes que no son del equipo. El anfitrión debe asegurarse que quien haga la presentación la realice correctamente, y que las preguntas y discusiones sean en forma ordenada.

En este caso puede haber una minuta que contenga los comentarios de quienes reciben la presentación. El contenido debería estar presente en otro documento (al que se puede hacer referencia) como, por ejemplo, las notas de la versión.

Retrospectiva

Este es un caso de reunión en la cual puede variar la estructura cada vez que se realice para refrescar la dinámica del grupo. En estas reuniones es imperativo explicar el desarrollo en los primeros minutos (a veces primero se realiza un ejercicio para romper el hielo y luego se comienza la reunión).

Estas reuniones se han visto en detalle en el capítulo “En retrospectiva”, pero no está de más recalcar que deben tener un límite definido, no sólo para la reunión en su totalidad sino para cada parte de la misma. Dada la naturaleza de las retrospectivas, es muy fácil que se extienda sin llegar a un final, a menos que se la guíe correctamente.

Programación de a pares

Esto no es estrictamente una reunión, pero hay recomendaciones para las demás reuniones que se pueden aplicar. Tener el objetivo claro es fundamental en la programación de a pares, así como también el respeto por el tiempo del otro. Nadie debe sentirse inhibido de tomar un descanso cuando lo necesita, o de excusarse para hacer algún llamado o tarea personal importante.

No son necesarias minutias en una sesión de programación de a pares, ya que el código generado es la documentación del trabajo hecho.¹⁰³

Reuniones con miembros remotos

En muchas ocasiones ocurre que uno, varios o todos los citados a una reunión no pueden concurrir. En estos casos la tecnología actual nos asiste para minimizar los problemas que esto trae aparejado.

Es importante hacer uso de todos los elementos a nuestra disposición, como videoconferencia, posibilidad de compartir la pantalla de la computadora y de dar el control a otro, y cualquier otro elemento que facilite la interacción. Uno de los pilares del agilismo es la cooperación e interacción entre los miembros del equipo, y esto siempre se ve disminuido al no estar todo el equipo presente.

Por todo esto, siempre que sea posible, hay que esforzarse para hacer reuniones presenciales, por lo menos, algunas veces durante el

desarrollo del proyecto (y cuanto antes mejor). En numerosas ocasiones hemos visto que a partir de una reunión presencial con miembros de equipos que no podían estar en la misma oficina a diario, la relación y la interacción experimenta un cambio positivo y rotundo.

Más allá de estas recomendaciones, lo más importante es evitar siempre que sea posible situaciones donde el equipo no puede trabajar junto a diario. Nada puede sustituir el trabajo cara a cara para un equipo ágil.

Juntos pero separados

Una mala costumbre que hemos observado con frecuencia es un equipo donde los integrantes trabajan en forma aislada, aun estando todos en la misma oficina. Equipos que estando a unos pocos metros unos de otros realizan reuniones diarias, cada uno sentado (de por sí un error) en su escritorio y conectados a una sala de reunión telefónica o virtual.

No hay ninguna excusa para esto. Si no hay sala de reunión disponible, pueden reunirse en un escritorio. Si allí molestan, en la sala de descanso. Si no, frente a los ascensores o en la puerta del edificio donde se reúnen los que fuman. No hay excusas para no reunirse cuando se está en la misma oficina y menos para una reunión de 15 minutos que, en muchos casos, se recomienda hacerla de pie.

Las reuniones son un medio, no un fin

Muchas personas en esta profesión, y especialmente quienes tienen responsabilidades de liderazgo, se quejan de que su día parece ser una serie de reuniones de principio a fin. Esto es en parte inevitable en algunas empresas y para algunos cargos. Pero quienes lideren un equipo ágil deben hacer el mayor esfuerzo por proteger a su equipo de estos vicios organizacionales.

En el caso del representante del cliente muchas veces es difícil evitar que ocupe su tiempo en reuniones, en especial si es alguien que está acostumbrado a estructurar su día utilizando su agenda. Una forma de evitar que al equipo se le dificulte el avance por tener poca disponibilidad de esta persona, es agendar reuniones con el equipo. Así, el representante ve clara y gráficamente su aporte, y puede tomar conciencia del impacto que tiene su ausencia, al ver la cantidad de tiempo que el equipo necesitaba de él y que ha rechazado.

Otra persona importante que es requerido erróneamente para reuniones fuera del proyecto es el de los líderes técnicos. En este caso es crítico que el líder del equipo evite esto todo lo posible, en especial cuando su presencia se requiere, por ejemplo, para comentar por enésima vez el avance de los detalles técnicos, a un grupo de gerencia que no tiene los conocimientos para entender de lo que se les habla. El tiempo de los líderes técnicos es mejor utilizarlo para analizar y resolver problemas complejos, enseñando a otros miembros del equipo o haciendo programación de a pares.

En resumen

Muchas prácticas ágiles implican una reunión de todo o parte del equipo. Es importante ser cuidadoso en la forma de realizar las reuniones para que sean efectivas y eficientes. Controlar tiempos, objetivos y reducir el trabajo improductivo es fundamental ya que, mientras más personas participan, cualquier ineficiencia tendrá un efecto enorme sobre el tiempo disponible para desarrollar en el proyecto, que es, en definitiva, el objetivo final.

[101](#) Martín Fowler explica esta y algunas otras recomendaciones interesantes sobre reuniones diarias en su artículo: It's Not Just Standing Up: Patterns for Daily Standup Meetings (<http://martinfowler.com/articles/itsNotJustStandingUp.html>).

[102](#) Con documentar no nos referimos siempre a documentos formales, puede ser en una herramienta de seguimiento de tareas, una nota sobre la pizarra del equipo, etc.

[103](#) Esta práctica la volveremos a ver en el capítulo “Construyendo con calidad día tras día”.

Irradiando información

Son las 9:30, hora de la reunión diaria. Todo el equipo se pone de pie y se acerca a la pizarra de tareas que está próxima a los escritorios de trabajo. Carlos pide iniciar la reunión.

– Ayer terminé con el segundo proceso y todos sus tests –dice mientras toma la tarjeta que estaba en la columna Desarrollo y la pasa a la de Testing Funcional y agrega– hoy voy a continuar con la pantalla de búsqueda y la tarea que surgió de la retrospectiva de investigar sobre herramientas para la generación de gráficos.

Toma ambos papeles, uno de la columna Retrospectiva y otro de Pendientes, las pasa a desarrollo y concluye:

– Ayer a última hora surgió un problema con el servidor de integración continua que está caído, quizás Nicolás lo pueda ver.

Nicolás, que tiene el rol de facilitador, agrega:

– Está bien, yo me encargo.

Anota el problema y en la pizarra lo ubica en la columna de Impedimentos generales. Diego, que está al lado de Carlos, continúa:

– Ayer estuve con el armado del nuevo ambiente en el servidor externo que hoy tengo que terminar –señala en la pizarra una tarea de desarrollo que tiene asignada– pero tengo un problema que no pude resolver, al parecer de permisos para crear la base de datos externa. Voy a necesitar ayuda de Carlos con esto. –y etiqueta la tarea con una marca que indica que tiene un impedimento– Hoy voy a comenzar el desarrollo de los perfiles. –Toma esa nueva tarea de las pendientes y la pasa a desarrollo.

Carlos comenta:

– Sí, Diego, después de la reunión nos juntamos y vemos lo de la base de datos.

– Ayer estuve con el desarrollo de los servicios web, me falta poco, y hoy voy a continuar con las notificaciones por email. No tengo impedimentos –acota Marcio y toma el papel de *notificaciones por email* de Pendientes y lo pasa a Desarrollo.

Alejandro, que es el responsable del producto, sugiere:

– Marcio, si no tenés inconvenientes, te pediría por favor si podemos terminar los prototipos para la funcionalidad de Reportes, ya que mañana me reúno con Jorge, el principal usuario. Me llamó hace unos minutos, y me gustaría llevarlos.

– De acuerdo, Ale, no hay problema, veamos primero los prototipos.

Y pasa la tarea de prototipos de Pendiente a Desarrollo. Enseguida, Nicolás comenta:

– Acordate que el límite de trabajo por persona es de dos tareas a la vez: ese fue el cambio que acordamos en la última retrospectiva.

– ¡Es verdad, gracias! Voy a agregar el límite de

WIP como nota a la pizarra si están todos de acuerdo. Todos aprueban la idea.

– Entonces, dejo la tarea de las *notificaciones por email*.

Y vuelve el papel a la columna de Pendientes. Pablo, que hace el testing funcional, continúa con su parte:

– Ayer terminé de automatizar todos los casos de prueba para la interfaz de usuario de la funcionalidad de registros, que es muy compleja, y no encontré ningún error, así que la paso a Finalizada.

– ¡Buenísimo!, ¡excelente! –todo el equipo festeja. Y sigue:

– Hoy voy a probar...

Veamos todo lo que pasó en esta reunión.

El equipo organizó en pocos minutos su trabajo del día con el soporte de la pizarra de tareas. No fue el facilitador (o desde un punto de vista no ágil, el gerente de proyecto) el que indicó qué tareas debía realizar cada uno en base a su calendario. La pizarra de tareas fue la herramienta que integró a todo el equipo (incluido el cliente) en la planificación, su actualización y mejora. Cada uno se autoasignó actividades, surgieron cambios, e incluso mejoras al proceso como, por ejemplo, visualizar el límite de tareas.

Todos pudieron ver el flujo del trabajo y tener una visión del estado general, en cuanto a qué hay que hacer, quién hizo y quién hará determinada tarea, qué progresos hubo, qué problemas hay que resolver y quién necesita ayuda. Hasta temas que el grupo se comprometió a mejorar de la retrospectiva anterior. Y no se tienen que acordar de nada. Todo está en la pizarra. No hay que acceder a ningún sitio web o software sofisticado de planificación, ni a ningún sistema de seguimiento de tareas. Solo hay que levantar la mirada.

El proceso es claro y transparente, y se ve en las columnas del tablero que se destaca por su sencillez. Todos lo entienden, incluso cualquier persona externa al proyecto que pasa por delante y lo ve.

El equipo comparte los progresos y la finalización de tareas, como también los impedimentos o potenciales problemas que emergen y quedan expuestos en el tablero y, a la brevedad y mediante la colaboración de todos, se resuelven.

La gestión visual aportó al seguimiento y control, a la organización, a los riesgos, a la calidad, a la integración, a la comunicación y a la colaboración del equipo. De esto se trata. Bienvenidos a este capítulo: Irradiando Información.

Gestión visual e irradiadores de información

La utilización de elementos y técnicas visuales para la organización del trabajo se conoce como gestión visual. Están aceptados en la comunidad ágil por su influencia positiva y por ser un gran complemento de otras técnicas de organización, como por ejemplo, Scrum.

Alistair Cockburn, uno de los primeros impulsores del movimiento ágil de desarrollo de software y participante de la escritura del Manifiesto, definió a cada uno de estos elementos visuales como un *irradiador de información*, que es un indicador que tiene las siguientes características [Cockburn 2008a]:

• Está ubicado en un lugar donde la gente lo puede ver mientras trabaja o camina por la zona, en general en la oficina del equipo de trabajo.

• Casi siempre está sobre papel, muy rara vez en una página web donde se accede frecuentemente.

• Contiene información importante.

• Se actualiza fácilmente y en forma periódica.

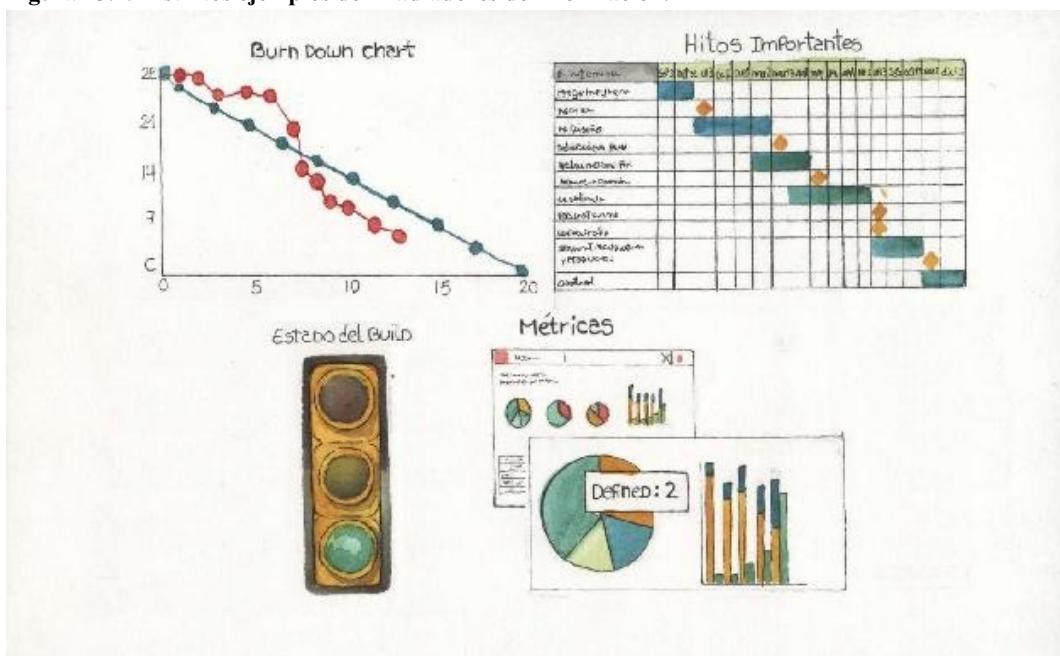
• No es necesario tener que preguntar qué significa, ya que se entiende a simple vista.

Algunos ejemplos son el gráfico de burndown, el indicador del estado de los ‘builds’, una lista de impedimentos, métricas de errores, los riesgos activos de mayor exposición.

Estas representaciones colocadas en las paredes¹⁰⁴ sirven para mantener al equipo organizado, ya que no se pierde el estado actual de los temas importantes. Solo requiere levantar la vista para encontrar la información más reciente del proyecto. No hay necesidad de hacer ningún esfuerzo, y se evitan pérdidas de tiempo por ubicar información. También son una forma de mostrar transparencia, ya que no se están ocultando datos entre pares del equipo ni tampoco hacia el cliente, generando así una mayor confianza.

Mediante el uso de irradiadores de información y su actualización frecuente, el equipo mantiene su progreso a la vista, anticipándose a los problemas, detectando mejoras en sus procesos, aumentando la colaboración y logrando así fortalecerse y autogestionarse.

Figura 13.1. Distintos ejemplos de irradiadores de información.



El gráfico de burndown

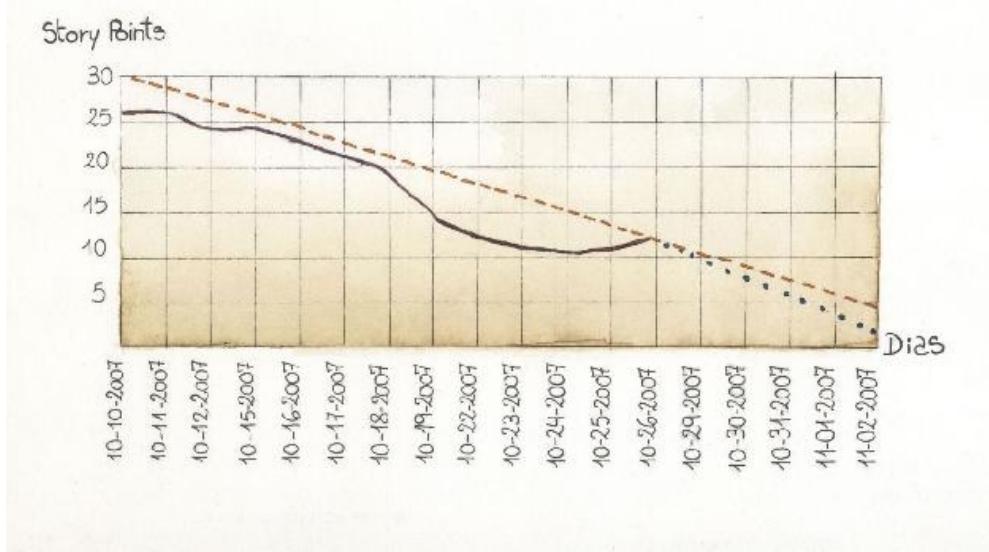
Este gráfico es un irradiador de información muy común en la gestión ágil y muchas veces se lo suele hacer en forma manual en el tablero del proyecto.

En el eje horizontal se representa la dimensión tiempo y en el vertical, el trabajo remanente. Las unidades utilizadas para cada una de estas dimensiones pueden variar dependiendo de lo que se pretenda mostrar.

En algunos casos se utiliza este gráfico para ver el avance durante una iteración. En tales casos, en el eje vertical se representan los story points remanentes y en el horizontal, los días de la iteración.

En el siguiente gráfico se muestra el burndown correspondiente al día 12 de una iteración donde el trabajo remanente es de 12 story points (línea oscura continua). También se aprecia una línea de guiones que representa una proyección basada en el ritmo actual de trabajo del equipo. Finalmente la línea punteada muestra el ritmo de trabajo que debería tener el equipo para llegar a completar el trabajo remanente al final de la iteración.

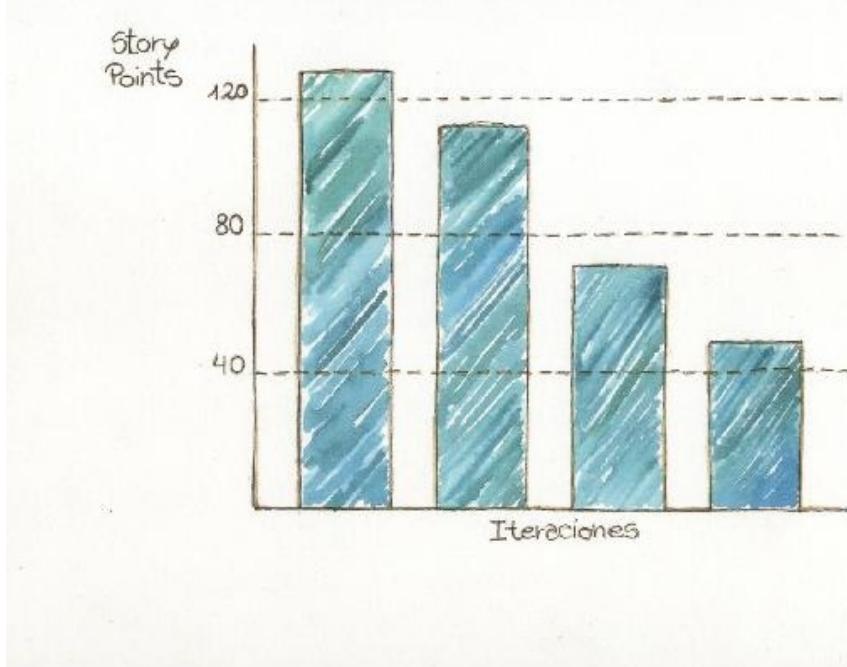
Figura 13.2. Gráfico de burndown de una iteración.



El gráfico de burndown también puede utilizarse para tener una visión de la evolución del proyecto a más alto nivel que la iteración

actual. En este caso, se representan iteraciones en el eje horizontal y los story points correspondientes a las user stories aún no completadas en el vertical.

Figura 13.3. Gráfico de burndown de proyecto.



Cuando las iteraciones son de una semana, el gráfico de burndown de iteración no agrega mucho valor, a diferencia del burndown de proyecto que siempre agrega valor.

Un punto importante del gráfico de burndown es que no solo nos muestra una foto, sino que también muestra tendencia, lo cual nos sirve para ajustar nuestro plan de trabajo.

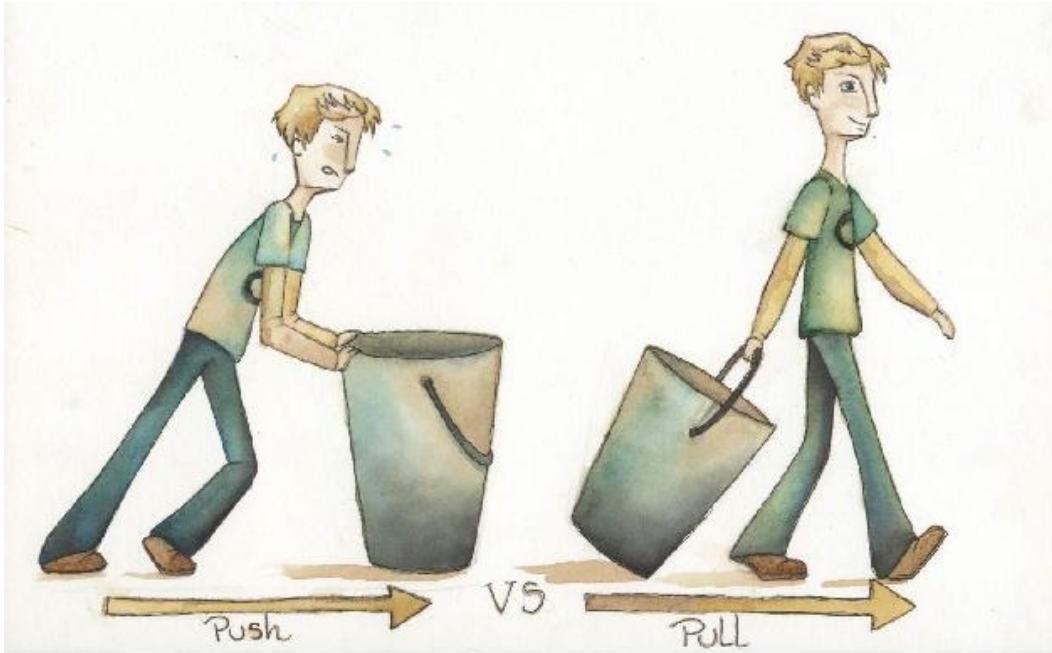
Los tableros

En la actualidad son muy utilizados los tableros en proyectos de desarrollo ágil, pero la idea tiene sus inicios en la década de 1950, a partir del sistema de producción de Toyota, basado en los principios de *Lean manufacturing* (o manufactura esbelta) y *Just in Time*¹⁰⁵. El principal soporte de este sistema eran las tarjetas (o señalizaciones visuales) denominadas *kanban*.

Años más tarde, en 2001, Mary Poppendieck escribe un artículo [Poppendieck 2001] sobre el parecido entre las metodologías ágiles, las ideas Lean y el sistema de producción de Toyota. En 2003, ya describía el uso de los tableros de tareas ágiles tipo Kanban en su libro *Lean Software Development*¹⁰⁶ [Poppendieck2003] que se basa en los siete principios que revolucionaron la industria manufacturera, pero en este caso aplicados al desarrollo de software.

Los tableros nos permiten implementar un esquema de trabajo de tipo *pull* (tirar) donde cada uno va tomando la tarea que considere más conveniente, a diferencia de los sistemas *push* (empujar) en donde el trabajo es impuesto. De esta forma, en colaboración con todo el equipo, se puede ir autogestionando y decidiendo la próxima tarea a encarar o los cambios que conviene hacer al plan que está visible a todos y al alcance de sus manos.

Figura 13.4. Pull versus Push.



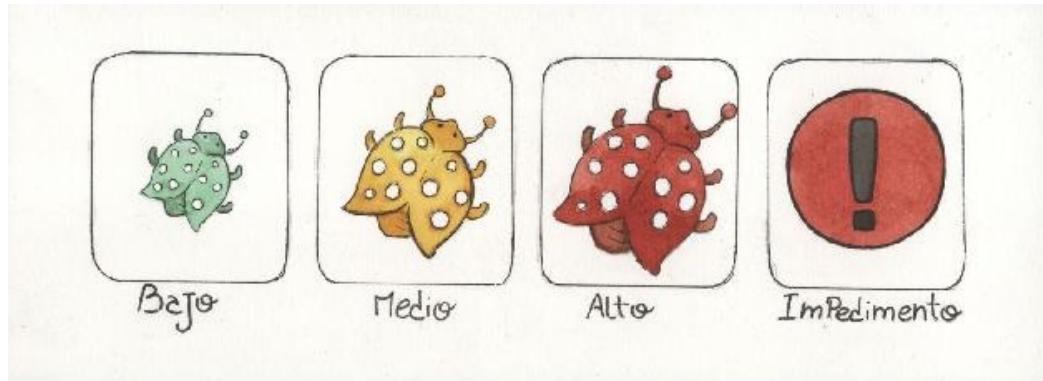
Existen innumerables formas de armar un tablero de tareas, según el proyecto, la necesidad, los gustos, la cultura, entre otros factores. No existe un formato estándar que se ajuste a todos los proyectos, sino que cada equipo debe decidir lo más conveniente e ir mejorándolo con el tiempo. Lo más sencillo es el tablero de tres columnas: pendiente, en progreso y finalizado:

Figura 13.5. El tablero más sencillo.

Pendiente	En Progreso	Finalizado
	<ul style="list-style-type: none"> [Tarea 1] [Tarea 2] [Tarea 3] [Tarea 4] [Tarea 5] [Tarea 6] [Tarea 7] [Tarea 8] [Tarea 9] [Tarea 10] 	<ul style="list-style-type: none"> [Tarea 1] → [Tarea 2] → [Tarea 3] [Tarea 4] → [Tarea 5] → [Tarea 6] [Tarea 7] → [Tarea 8] → [Tarea 9] [Tarea 10] → [Tarea 11]

También se pueden armar opciones más completas que incluyan, por ejemplo: una columna para las tareas de la retrospectiva anterior, datos de la iteración (rango de fechas desde y hasta, número, estado actual), fotos de los miembros del equipo, columna para los impedimentos generales, límites de *WIP* (explicado más adelante) por persona o por columna, referencias a los tipos de tareas según los colores de las tarjetas, calcomanías para representar tareas particulares con impedimentos o defectos, etc.

Figura 13.6. Calcomanías para impedimentos y diferentes severidades de defectos.



O como propone Mike Cohn en lo que denomina *Scrumboard* [Cohn 2003], en donde cada fila representa una user story que es explotada en tareas, las cuales van avanzando por el tablero con su progreso.

Figura 13.7. Un tablero más completo. En este caso, las tareas de testing las realizaban dos personas y se usaron líneas horizontales para saber quien había hecho el desarrollo.

Por otro lado, existen los tableros Kanban que proponen, entre otras prácticas¹⁰⁷, limitar el trabajo en progreso (también conocido con la sigla WIP de *work in progress*). Lo que se pretende es mantener baja la cantidad de trabajo en curso para ayudar a reducir la sobrecarga y el *task switching*, como también detectar problemas y cuellos de botella.

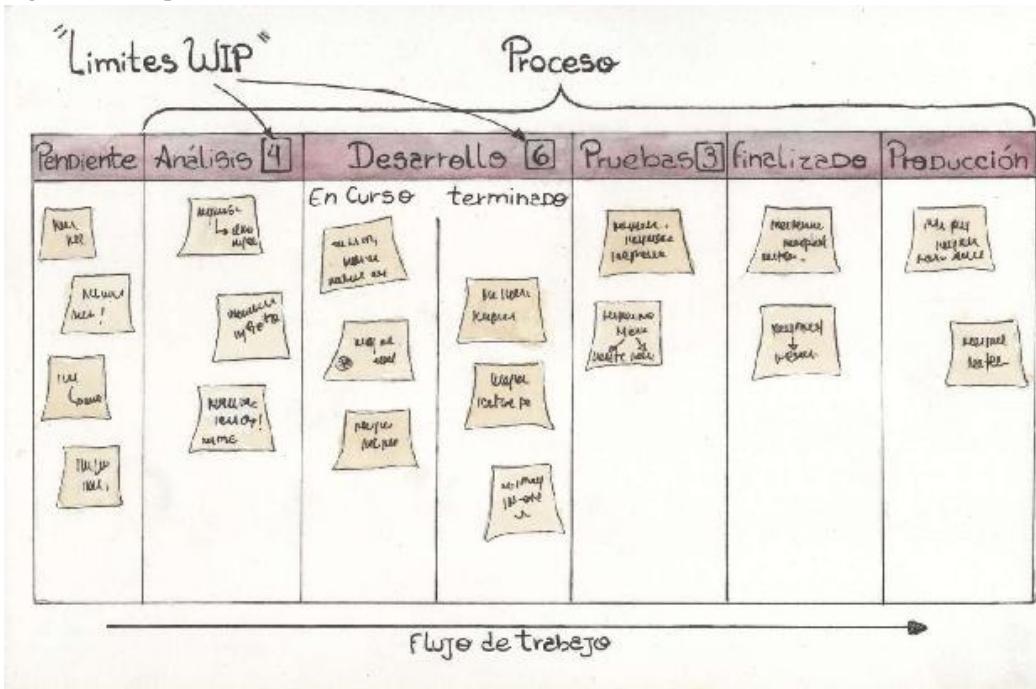
Si el equipo acumula muchas tareas en progreso, va a ser difícil detectar en el corto plazo si hay problemas con alguna. Un ejemplo es el de comenzar nuevas tareas cuando aparecen impedimentos en las actuales, y en vez de resolverlos se toman nuevas, acumulando así tareas empezadas, sin terminar ninguna. De esta manera, cuando no quedan tareas por comenzar, saltan a la vista los problemas que hay que resolver, pero ya no queda mucho tiempo para tomar acciones preventivas o correctivas que resulten eficientes.

Figura 13.8. Un Scrumboard.

User Story	Pendiente	En Progreso	Finalizado
<p>Navegar:</p> <p>recorrer net, leer información.</p>	<p>recorrer net información</p> <p>Navegar net, leer</p> <p>Navegar net, leer</p> <p>Navegar net, leer</p>	<p>recorrer net, leer información</p> <p>Navegar net, leer</p> <p>Navegar net, leer</p> <p>Navegar net, leer</p>	<p>recorrer net, leer información</p> <p>Navegar net, leer</p> <p>Navegar net, leer</p> <p>Navegar net, leer</p>
<p>Recuperar datos:</p> <p>recorrer net, leer información.</p>	<p>recorrer net, leer información</p> <p>Navegar net, leer</p> <p>Navegar net, leer</p>	<p>recorrer net, leer información</p>	<p>recorrer net, leer información</p> <p>Recuperar información</p> <p>Recuperar información</p>

Otro ejemplo puede ser el de un tester que está sobrecargado de tareas, pero con la ayuda de los límites de trabajo el equipo lo detecta y puede colaborar antes de seguir construyendo y dificultando más el flujo.

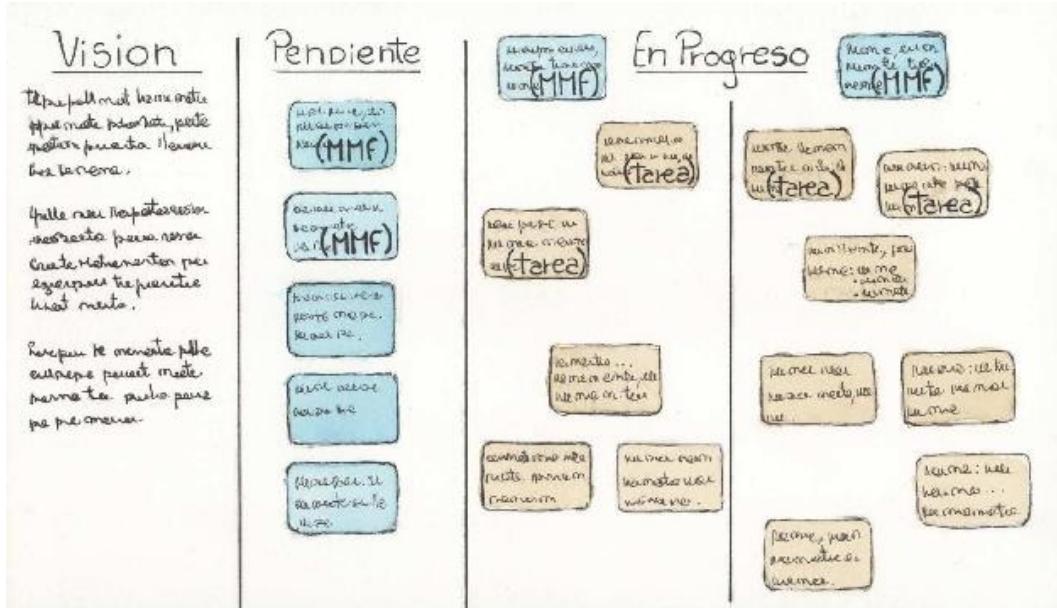
Figura 13.9. Aspecto de un tablero Kanban.



En todos los casos se busca que el flujo de las tareas sea continuo y en paralelo. Algunas funcionalidades se pueden analizar mientras unas están en desarrollo y otras en pruebas, dependiendo de como esté organizado el proceso (representado por las columnas del tablero).

La última variante de tableros que vamos a ver es la pizarra de detective [108](#) que propone James Shore para implementar el flujo de una pieza, basado en el sistema de producción de Toyota [109](#). Lo que se busca es que el equipo complete una sola funcionalidad (MMF [110](#)) por vez, y de esta forma minimizar el trabajo en progreso y eliminar las colas de espera que generan un falso flujo continuo y aumentan el tiempo del ciclo (que es el período desde que se inicia el proceso hasta que termina, en este caso, es el tiempo en completar un MMF).

Figura 13.10. Pizarra de detective para flujo de una pieza.



En la pizarra de ejemplo se muestran dos columnas para los dos MMFs en progreso, ya sea porque tamaño del equipo lo amerita o por si existen otras consideraciones que lo requieran.

Para implementar este tipo de tablero se requiere que el equipo sea multifuncional, que trabajen todos en un mismo lugar físico, y utilizar prácticas de XP con fases simultáneas (testing, desarrollo, diseño, despliegue, análisis, planificación) a diferencia de Kanban en donde las fases están separadas y son las diferentes columnas del tablero. Es una versión de XP (*eXtreme Programming*) sin la fase de planificación ya que se necesita generar un flujo continuo y se hace utilizando MMFs en vez de iteraciones.

Recomendaciones complementarias

- Realizar las reuniones diarias frente a los tableros para poder actualizarlos y reflejar el estado de las tareas e impedimentos de la iteración. Es una buena práctica hacer todas las modificaciones entre columnas en el momento de la reunión, así todo el equipo está al tanto de los movimientos y de los últimos progresos e impedimentos. Para poner etiquetas, indicar defectos u otras actualizaciones no es necesario esperar a la reunión.
- Al armar el tablero, utilizar cinta eléctrica para las divisiones de las columnas, papeles autoadhesivos de diferentes colores para las tareas, marcador tipo fibrón para escribir (no usar bolígrafo) y hacerlo siempre en letra mayúscula imprenta para que sea legible. En el blog de Xavier Quesada [Quesada 2009] se pueden encontrar más recomendaciones de este tipo y muchos ejemplos de tableros.
- Para equipos distribuidos, un acercamiento a estas prácticas se puede encontrar utilizando tableros online. Existen diferentes opciones gratuitas y pagas en la web, y tienen como ventaja adicional que si se quieren sacar métricas, puede requerir menor esfuerzo. Pero si el equipo trabaja en el mismo lugar, es conveniente el uso de tableros físicos y la interacción cara a cara, con lo que se va a lograr una mejor colaboración que ante cualquier software de medio.
- Probar las diferentes ideas que el equipo vaya detectando como potenciales mejoras. Por ejemplo, la limitación del trabajo en progreso es una buena práctica no solo en Kanban, sino también en cualquier tablero de tareas, para detectar problemas y cuellos de botella.
- Es conveniente llevar en la pizarra las tareas y acciones de la retrospectiva para hacerles un mejor seguimiento, ya sea en el mismo tablero o en otro dedicado.
- El uso de tableros no implica reemplazar un sistema de seguimiento de tareas y/o errores, que son más completos en cuanto a los datos a cargar sobre cada elemento de trabajo. Además proveen métricas, mantienen la trazabilidad y permiten guardar estas evidencias. Para muchas organizaciones se requiere su uso, en especial si se trabaja con un proceso de desarrollo certificado. En particular para el caso del manejo de errores, hay equipos que prefieren utilizar solo el software de seguimiento para los defectos menores, y los importantes tratarlos en la pizarra directamente para darle un flujo más rápido. Una buena opción, sobre todo, si se trabaja con testers, es hacer marcas con colores sobre las tareas, para indicar que tienen errores de determinada criticidad.
- Inicialmente definir las columnas de la pizarra en base al proceso actual de desarrollo, luego ir mejorando la estructura iteración a iteración. Por ejemplo, si el proceso actual para un determinado proyecto consta en hacer un fuerte análisis de los requerimientos, desarrollar, luego realizar un testing funcional interno, para luego hacer la entrega de la iteración al cliente; se podría comenzar con las columnas: pendiente, análisis, desarrollo, testing, finalizado.
- Los tableros de tareas se reinician al comenzar una iteración para colocar las tareas del nuevo plan. Ese es un buen momento para aprovechar y hacer las modificaciones de estructura, si es que se detectaron mejoras a experimentar, originadas a partir de una retrospectiva o por consenso de todo el equipo. Por ejemplo, agregar o eliminar columnas, cambiar tamaños, marcar nuevos límites de trabajo en progreso, etc.

En resumen

La agilidad pone el foco en las personas y su interacción, por lo que resulta muy conveniente que ciertos datos y elementos claves del trabajo estén disponibles para todos los miembros del equipo, de modo tal de facilitar la comunicación y la colaboración.

En este capítulo vimos que utilizando herramientas de gestión visual como radiadores de información y tableros de tareas, se logra una real integración del equipo, generando mayor confianza y ayudando a su autogestión, consiguiendo así llegar más rápido y mejor a los objetivos propuestos.

[104](#) [Beck 1999] hace referencia a ellas como *Big Visible Charts* (gráficas visibles y grandes).

[105](#) Esta política industrial consiste básicamente en aumentar la eficiencia manteniendo el inventario al mínimo posible, produciendo sólo lo estrictamente necesario, en el momento solicitado y en las cantidades justas.

[106](#) Véase sección de Lean en el Apéndice: “La riqueza de la diversidad”.

[107](#) Véase la sección de Kanban en el Apéndice: “La riqueza de la diversidad”.

[108](#) Del inglés: *detective blackboard*. Lo denominó así por como se ve en las series de televisión en donde el detective coloca toda la información del caso en la pizarra.

[109](#) [Pawlak 2009] The Toyota System, One Piece Flow.

[110](#) *Minimum Marketable Feature*, véase el capítulo “Planificación Constante”.

¿Quién manda a quién?

Introducción

Como vimos en capítulos anteriores, uno de los principales cambios que trae una actitud realmente ágil es una nueva mirada sobre la gente. En particular, hay dos grandes cambios sobre los roles tradicionales:

- Cambio en el liderazgo (encarnado tradicionalmente en el líder de proyecto, gerente de proyecto o líder de equipo).
- Cambio dentro del equipo, en el que los roles tradicionales (analista, desarrollador, arquitecto, tester, etc.) se desdibujan para ser reemplazados por habilidades distintas que los integrantes aportan al conjunto para lograr los objetivos planteados.

En un proyecto ágil, el liderazgo está distribuido en tres áreas principales:

- *Equipo*: el equipo de desarrollo, autoorganizado y responsable solidario por el producto y por el proceso de trabajo.
- *Responsable de producto*: responsable por la visión y el retorno de inversión del producto/proyecto.
- *Facilitador*: responsable de maximizar el potencial del equipo para lograr sus objetivos.

Con respecto a los roles, el cambio más significativo es que todos los tradicionales se consideran incluidos en el rol *Equipo*. La principal razón para este cambio es la necesidad de autoorganización, puesto que es el equipo mismo quien define hacia su interior cómo organizar mejor el trabajo. Por lo tanto, los roles no deben estar rígidamente predefinidos por el proceso o la organización¹¹¹. También en el flujo de la comunicación hay un gran cambio, ya que está orientado hacia el interior del equipo de desarrollo en lugar de que toda la información pase a través de la gerencia.

En las siguientes secciones desarrollaremos algunos aspectos del liderazgo y los roles específicos de un proyecto ágil, sin adentrarnos en la correspondencia de roles ágiles con los tradicionales. Esta correspondencia está tratada en más detalle en los capítulos “Empezando por la aceptación” (analistas y testers), y “Arquitectura y diseño en emergencia” (desarrolladores y arquitectos)¹¹².

Liderazgo

La función de liderazgo integra múltiples aspectos necesarios para mantener un ambiente de trabajo satisfactorio y que el equipo logre sus objetivos. Como vimos en la introducción, en un ambiente ágil esa función se comparte y tiene múltiples exponentes. En esta sección, desarrollaremos algunos de sus aspectos centrales en forma general, para luego aplicarlos a los roles en un entorno ágil.

Las principales responsabilidades de liderazgo son la motivación, la visión y la transformación.

Motivación

Daniel Pink [Pink 2009] propone tres factores clave en la motivación en el trabajo moderno: autonomía, maestría y propósito.

La autonomía es producto de la confianza cuando a los equipos se les permite encontrar su propio camino¹¹³. La búsqueda en sí misma, con la libertad y responsabilidad que trae aparejadas, actúa como elemento motivador.

La maestría es atractiva porque implica crecimiento personal a través del aprendizaje y la mejora continua, y es un elemento motivacional intrínseco del trabajo. En particular, tiende a ser más interesante cuanto más interesante es el trabajo en sí.

Por otro lado, la maestría en nuestro trabajo define nuestra propia identidad mediante la pertenencia social al grupo de aquellos con los que trabajamos y quienes definen implícitamente lo que es “hacer bien el trabajo”. Por ejemplo, aprendemos a ser analistas cuando hacemos el trabajo con otros analistas, mientras que cuando tomamos clases sobre el tema solo aprendemos cómo debíamos hacer análisis. Esta relación aprendiz-maestro puede ser explícita o implícita. En algunos casos solo sabemos que hacer mediante la imitación de otros que hacen la misma tarea¹¹⁴. Este proceso de dominio de nuestro trabajo consiste en “aprender a ser” alguien capaz de hacer el trabajo en lugar de “aprender a hacer”.

El propósito hace que nuestro trabajo tenga sentido, depende de los objetivos que nos ponemos, de la visión que nos orienta y enfoca nuestros esfuerzos, y del desafío que implica lograr lo que nos proponemos. Es el elemento motivador intrínseco por excelencia, muchas veces asociado íntimamente al placer que nos produce.

Las formas tradicionales de motivación, basadas en estructuras de premios y castigos, son incapaces de motivar a los equipos a producir resultados excelentes. El dinero, por ejemplo, actúa como elemento motivador solo mientras es una necesidad insatisfecha o durante un corto período (así es como algunas personas cambian con frecuencia de trabajo para negociar sus salarios manteniendo su atractivo como elemento motivador). Incluso, los modelos de premios y castigos pueden ser muy destructivos para el desarrollo de equipos y organizaciones porque tienden a promover el comportamiento premiado y a vaciar de contenido el resto de las políticas de la organización¹¹⁵.

Un ejemplo perfecto de esta actitud es una célebre tira cómica de Dilbert, en la que este le pide a Ratbert (una rata) que baile sobre el teclado dado que la compañía para la que trabaja le paga diez dólares por cada defecto (*bug*) que corrija, y por lo tanto desea que haya muchos defectos para corregirlos y ganar mucho dinero¹¹⁶. La tira termina, como era de esperarse, con Dilbert frustrado porque Ratbert programó un navegador web.

El problema puede generalizarse para todos los casos en que un equipo es observado y controlado. En palabras de Tobías Mayer, “las métricas deberían usarse para medir la realidad, no para medir el éxito o el fracaso. Solo las medidas de verdad son confiables puesto que no incitan a los equipos a usar soluciones rápidas pero temporarias”¹¹⁷.

Visión

La visión es una de las principales herramientas para alinear al equipo y guiar el trabajo en pos de objetivos comunes. En muchos casos, la función de liderazgo está directamente asociada al mantenimiento y comunicación de la visión. En Scrum, por ejemplo, mantener la visión y asegurarse de que el producto construido esté alineado con ella son algunas de las principales responsabilidades del Dueño de producto (*Product Owner*). La visión también provee foco al equipo mediante la definición de objetivos y una dirección para sus esfuerzos.

Transformación

Es una responsabilidad del liderazgo ayudar a los demás a crecer y mejorar continuamente, incluyendo desarrollar habilidades de liderazgo. Entonces, el líder es un maestro de líderes, y por lo tanto ayuda a lograr los objetivos concretos, además de renovar y hacer

crecer la organización.

Hace ya muchos años, cuando trabajaba en un banco multinacional, tuve un jefe excepcional, Osvaldo Carrizo. Cuando él hablaba de nosotros, los integrantes de su equipo, decía “él trabaja conmigo”, nunca “él trabaja para mí”. Ese simple gesto se quedó para siempre marcado en mi memoria, y cuando me tocó estar en ese lugar, me ayudó a ver un camino mejor para liderar equipos de desarrollo.

D.F.

Facilitación

El líder tiene la responsabilidad de facilitar el trabajo de aquellos a los que sirve, sin hacer el trabajo él mismo. Debe ayudar a:

- Remover obstáculos.
- Mantener la disciplina en la forma de trabajo recordando los acuerdos construidos al respecto.
- Identificar y acompañar en la resolución de conflictos.
- Mejorar continuamente.

Confianza

El líder es responsable de construir con el equipo un ambiente seguro y de confianza, donde todos puedan dar de sí lo mejor y arriesgarse a la excelencia, explorando nuevos caminos.

La confianza requiere una historia compartida, devolución (*feedback*) oportuna y compromisos cumplidos. Como explica Tom De Marco, la confianza debe entregarse antes de ser merecida, y por eso es que tenemos que arriesgarnos a confiar en otros para construir equipos. Luego, por supuesto, los demás deben cumplir con sus compromisos para mantener esa confianza inicial.

En la próxima sección, una recorrida por algunos tipos y estilos de liderazgo nos servirán de marco para entender más en detalle cómo se ejercen los roles en un proyecto ágil.

Tipos de liderazgo

• *Líder servil*: el líder cree que es su responsabilidad servir a la gente que trabaja con él. Una de sus mayores prioridades es asegurarse de que el equipo y la organización constituyan un muy buen ambiente de trabajo para sus miembros (esto incluye desde el entorno físico hasta las relaciones interpersonales, como veremos en el capítulo “No hay como un buen ambiente”).

• *Líder transformador*: el líder que trabaja con el equipo para mejorar su capacidad y satisfacción, para transformar a los miembros del equipo en líderes (esta idea fue concebida originalmente en el campo de la política).

• *Líder transaccional*: el líder lida con el equipo a través de premios y castigos, tiende a gerenciar en lugar de dar soporte¹¹⁸.

Estilos de liderazgo

De acuerdo a Daniel Goleman [Goleman 2000], un estudio global sobre ejecutivos permitió identificar seis estilos de liderazgo. Su recomendación es que deben usar estos estilos de acuerdo a la situación¹¹⁹.

• *Integrador*: “La gente primero”¹²⁰, este estilo ayuda a crear armonía y favorece la consolidación del equipo.

• *Visionario*¹²¹: “Vengan conmigo”, el líder mantiene la visión y el foco, pero puede tener problemas para dejar hacer al equipo¹²².

• *Democrático*: “Podemos encontrar el camino”, el líder busca consenso y participación en las decisiones. Este estilo promueve la innovación dando más libertad pero requiere fuerte disciplina de los involucrados.

• *Mentor*: “Puedo ayudarlos a ser tan buenos como pueden ser”. Corresponde al líder facilitador y transformador, se enfoca en la gente antes que en el producto.

• *Coercitivo*: “Hagan lo que yo digo”, el estilo tradicional de mandar y controlar. Se corresponde perfectamente con un líder transaccional.

• *Ejemplar*¹²³: “Seamos excelentes”, este estilo desafía a ser excelentes, pero requiere acuerdo entre el ritmo que marca el líder y la capacidad del equipo. Desafiarlo a un ritmo más rápido puede ayudarlos a mejorar, pero estirarlos más allá de lo que pueden tolerar puede quebrarlos.

Es interesante notar que el estudio encuentra que una mezcla de liderazgo visionario, integrador, democrático y mentor resulta ser la óptima para crear buen ambiente organizacional y mejorar el desempeño. Esa mezcla se corresponde muy bien con los roles en un entorno ágil.

Roles

En esta sección vamos a desarrollar en detalle los roles de un proyecto ágil, usando como marco las consideraciones sobre liderazgo de la sección anterior.

El cliente

Los representantes del cliente en el proyecto tienen dos responsabilidades fundamentales: promover que el producto construido aporte el mayor beneficio posible, y aportar la información y el conocimiento del negocio necesarios para el trabajo del equipo.

Los roles en el entorno del cliente son:

• *Responsable de producto*: responsable por la visión y el retorno de inversión del producto/proyecto. Es el único rol requerido de parte del cliente, aunque en condiciones ideales quien lo ejerce es también un Experto del negocio¹²⁴.

Características:

- Define, mantiene y comunica la visión del proyecto.
- Tiene conocimiento del negocio.
- Tiene poder de decisión con respecto a las características del producto.
- Prioriza los requerimientos.
- Mantiene el plan de versiones¹²⁵ del producto.
- Integra y resuelve conflictos de intereses entre los distintos interesados dentro de la organización cliente.

Como ya vimos, detenta el liderazgo en cuanto al producto, definiendo la dirección estratégica basada en la visión, y aportando el

foco necesario para materializar esa visión en el producto construido. Este foco se ejerce mediante las reuniones de lanzamiento de proyecto, de planificación (ajustando prioridades), revisión de producto (realizando comentarios sobre el producto parcial), interacciones cotidianas de intercambio de información, definición de criterios de aceptación, definición de fechas de lanzamiento¹²⁶, etc.

- *Patrocinador (Sponsor)*: garantiza el presupuesto del proyecto y es el responsable último de los resultados del proyecto.
- *Experto del negocio*: es la persona que conoce en detalle el negocio y tiene experiencia en el mismo. En general, es alguien que sabe “cómo” se hace el trabajo y no simplemente sabe “sobre” el trabajo.

Este rol debe ocuparlo alguien con experiencia pero que también pueda retirar su atención de la operación del negocio diaria para dedicarse al proyecto de desarrollo. Cuando su dedicación a la operación diaria del negocio no se ve afectada, puede considerarse un síntoma claro de falta de compromiso y un riesgo alto para el proyecto¹²⁷.

El equipo

Los principales roles en el entorno del equipo de desarrollo son:

- *Desarrollador*: es responsable de la construcción del producto. También ayuda a elaborar los escenarios y ejemplos que luego sirven para definir el producto y las pruebas técnicas.

- *Tester*: es otro de los participantes en la elaboración de los escenarios y ejemplos que luego sirven para probar la aplicación. Su foco fundamental está en lograr un producto de calidad participando de todos los aspectos del proceso.

- *Analista (o Analista de negocio)*: debe ser un facilitador de intercambio de conocimiento. Ayuda a construir el backlog, definir criterios de aceptación y escenarios de prueba. También mantiene la documentación necesaria para el proyecto. Como ya vimos, el rol tradicional era el de “traductor” entre los clientes y el equipo de desarrollo. En un equipo ágil, no hay traducción sino una interacción directa entre el cliente y el equipo (los integrantes que lo requieran), con el resultado de que este, siendo el responsable de realizar el trabajo, tiene acceso eficiente a la información que necesita¹²⁸.

- *Arquitecto/Diseñador*: ayuda al equipo a tomar las decisiones técnicas y de diseño más apropiadas, a partir de su experiencia y de su comprensión del contexto de negocio. Participa codo a codo con los desarrolladores del proceso de construcción.

- *Especialista*: aportan habilidades específicas al equipo, del que pueden no formar parte integral si su participación es muy esporádica. Ejemplos son especialistas en seguridad, bases de datos, redes, servidores, diseño de experiencia de usuario, etc.

El facilitador

El rol de facilitador (también conocido como *coach o scrum master*¹²⁹) es el más novedoso respecto a otros métodos de desarrollo de software, por cuanto su única responsabilidad es maximizar los resultados acompañando al equipo y al responsable de producto a mejorar constantemente su proceso y su práctica cotidiana. Este rol es un ejemplo vivo del concepto de controlar dejando hacer (*control by release*), puesto que no es ni jefe ni dueño, sino acompañante de un equipo al que no pertenece.

Una forma de ver a este rol en contexto es tener presente que tanto el producto como el equipo son resultados del proyecto¹³⁰. Con esa mirada, el facilitador y el responsable de producto tienen responsabilidad divida sobre los resultados del proyecto, el primero por el equipo, el segundo por el producto. Alan Cyment describe tres aspectos de este rol:

*Facilitador*¹³¹: es un acompañante neutral que ayuda al equipo a tomar las decisiones apropiadas para obtener los mejores resultados. En este sentido, es el encargado de:

- Remover obstáculos (obtener recursos de hardware o mobiliario necesarios, asegurar la dedicación de los miembros del equipo, etc.).
- Dar apoyo a la realización de las prácticas específicas del proceso (la reunión diaria o la retrospectiva).
- Ayudar al equipo a lidiar con los conflictos personales (que tienden a ser negativos) o sobre la tarea (que tienden a ser positivos).
- Servir de embajador del equipo ante la organización.

Coach: es el responsable de que el equipo maximice su potencial y produzca resultados excelentes. En la mayoría de los casos, no aporta las respuestas sino que ayuda al equipo a encontrar las propias.

Debe echar mano de varias herramientas, entre ellas:

- *Pregunta*: la pregunta busca que el otro desarrolle su propio conocimiento a partir de su experiencia y sus lecturas. El facilitador debe ser capaz de ofrecer las preguntas apropiadas para incomodar¹³², hacer pensar, motivar y ayudar a aprender. La pregunta no busca respuesta, ni siquiera requiere que sea tomada en cuenta.

- *Devolución (feedback)*: el propósito de la devolución es proveer al otro información sobre algo concreto que puede mejorar (o mantener). El valor está en que la mirada del facilitador puede hacer notar algo que pasó desapercibido, o para lo que el otro no encuentra un mejor camino. Los pasos de la devolución, como propone Diana Larsen [Larsen 2009], deben ser:

- a. *Pedir permiso*: esto formaliza el encuentro y asegura que hay disposición del otro a recibir la devolución.

- b. *Referir lo sucedido*: debe hacerse referencia concreta a lo ocurrido, que preferentemente debe haber ocurrido hace poco tiempo para estar fresco en la memoria del otro.

- c. *Explicar las consecuencias*: es necesario aclarar las consecuencias y el impacto que tiene lo sucedido, puesto que es la razón por la cual vale la pena hacer la devolución y buscar una mejora.

- d. *Proponer acciones concretas*: hay que dejar en claro qué se espera del otro, para ayudarlo a mejorar.

- *Retrospectiva*: es la oportunidad perfecta para la reflexión y la discusión sobre la forma en que el equipo está realizando el trabajo. El facilitador puede proponer actividades o señalar que “el rey está desnudo” aunque nadie lo vea.

Mentor: es, al mismo tiempo, maestro y guía del equipo y el responsable de producto. Muestra un mejor camino o ayuda a encontrarlo. En general, es un referente del método y las prácticas básicas a utilizar (tener experiencia aplicando métodos y prácticas ágiles) y debe poder guiar al equipo en su camino de mejora continua.

Una de sus principales funciones es enseñar los valores, el proceso y sus actividades, las prácticas y los roles. En muchos casos, esta enseñanza sirve para establecer un “lenguaje común” para reflexionar y discutir el trabajo. En el caso de equipos con integrantes que ya

tienen experiencia y otros que no, ayuda a homogeneizar una visión del trabajo compartido. También ayuda a alinear los valores y las prácticas ágiles a personas que hayan adquirido experiencia en otros contextos o aquellos que puedan haberse alejado de la filosofía ágil sin darse cuenta.

Hay un delicado equilibrio entre estos tres aspectos. Un facilitador puede caer en una actitud de coach extrema, utilizando solo la pregunta como herramienta y nunca aportando sus conocimientos y experiencia. En otro extremo, puede intentar controlar como el equipo hace su trabajo. Finalmente, puede también abandonar su responsabilidad de maestro y dejar que cada persona del equipo aprenda como pueda.

Cualquiera de estos extremos puede ser muy dañino, en algunos casos generando inefficiencia en el proceso de mejora (porque hace más difícil lograr acuerdos) y en otros puede afectar los resultados (si promueve las horas extras en forma sostenida, arriesgando la sustentabilidad). El facilitador debe evitar manipular y perder la confianza del equipo, aunque a veces deba ayudarlos indirectamente a dejar de lado sus propias costumbres y formas de trabajo. Por ejemplo, cuando una persona no acepta la responsabilidad de autoorganizarse y asignarse tareas, el facilitador puede solicitar a sus compañeros que no tomen para sí ciertas tareas molestas durante algunos días para que queden “disponibles” para esa persona y hacerle ver lo incómodo que puede ser no autoorganizarse.

La estructura del equipo

Debe reunir entre sus miembros todas las habilidades necesarias para cumplir con las responsabilidades que se le encomiendan. En particular, debe tener una buena mezcla de habilidades en el equipo de desarrollo, testers, desarrolladores, arquitectos, analistas, especialistas, etc., en proporciones acordes a las características del proyecto, y de representantes del cliente, tanto expertos como responsables de tomar decisiones. Además, debe tener un facilitador para acompañarlo. Lo ideal es que los miembros estén dedicados a tiempo completo, pero dependiendo de las características de la organización, el proyecto y el equipo mismo, las dedicaciones y proporciones pueden variar.

En resumen

En este capítulo recorrimos las cuestiones de liderazgo y los roles en un proyecto ágil. La principal diferencia de la mirada ágil es que el liderazgo es compartido entre el equipo de desarrollo, el responsable de producto (en general, representante del cliente) y el facilitador.

También recorrimos en detalle las características que deben tener quienes ejercen esos roles en la práctica, y mostramos como cada rol requiere cierto aspecto de liderazgo en sus tareas, tomando responsabilidad por el proyecto, en lugar de cumplir con la realización de ciertas actividades. Estos roles, en algunos casos más o menos ideales, deben estar siempre anclados en la experiencia y las necesidades concretas del proyecto.

¹¹¹ Otra razón para evitar la asignación de roles fijos es permitir ajustar la capacidad del equipo para realizar cierta tarea (por ejemplo, que los desarrolladores ayuden a hacer pruebas si ese es un cuello de botella).

¹¹² Para ver un análisis detallado de la correspondencia entre el rol tradicional de jefe de proyecto y otros roles ágiles véase [Adkins 2010], capítulo 1.

¹¹³ Véase el capítulo “No hay como un buen ambiente”.

¹¹⁴ Véase [Brown 2000], cap. 5. Allí los autores destacan los aspectos sociales del aprendizaje.

¹¹⁵ Véase [Poppdick 2004] para un desarrollo detallado e incisivo de las desventajas de las estructuras de premios y castigos en ambientes de desarrollo ágil.

¹¹⁶ Esta tira se puede encontrar en <http://dilbert.com/strips/comic/1995-11-14/>.

¹¹⁷ El original en inglés es “metrics should be used to measure truth—not to measure success or failure. Only measures of truth can be trusted not to incite quick-fix behavior in a team”, tomado de [Mayer 2009]. Reemplazamos el término “verdad” por “realidad” porque describe mejor el sentido original.

¹¹⁸ Un líder así tiende a actuar como un gerente tradicional y no es apropiado para un equipo de desarrollo. Véase [Humphrey 2005], pág. 13, “Leading versus Managing”.

¹¹⁹ Esta idea es coherente con la filosofía ágil de dividir aspectos del liderazgo en distintos roles.

¹²⁰ Los textos entre comillas están tomados de [Goleman 2000], la traducción es nuestra.

¹²¹ El artículo original dice “Authoritative” que puede traducirse como “autoritario”, sin embargo, preferimos una traducción más precisa y menos literal usando el término “visionario”.

¹²² Véase el capítulo “No hay como un buen ambiente” donde se describe el concepto de controlar dejando hacer.

¹²³ “Pace-setting” en el inglés original.

¹²⁴ En particular, Scrum requiere que exista un único responsable de producto (Product Owner) que cumpla este rol para resolver los conflictos entre los interesados dentro de la organización cliente.

¹²⁵ En inglés, “Release plan”.

¹²⁶ En inglés, “Release dates”.

¹²⁷ Véase en [Al-Rawas 1996] un estudio de campo detallado sobre la comunicación de requerimientos.

¹²⁸ Coplien y Harrison describen esta estructura como un patrón organizacional que denominan “Work flows inward” (“El trabajo fluye hacia adentro”), véase [Coplien 2004].

¹²⁹ Elegimos el término Facilitador para evitar las traducciones de Coach y la grandilocuencia de Scrum Master.

¹³⁰ Véase el capítulo “No hay como un buen ambiente”.

¹³¹ Como Alan Cyment escribe sobre Scrum en [Cyment 2012], se refiere al rol como Scrum Master y el aspecto como facilitador. En nuestro caso puede resultar confuso porque llamamos al rol igual que al aspecto.

¹³² En sentido de “sacar fuera de su zona de confort”.

No hay como un buen ambiente

“Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.”

Agile Manifesto

Introducción

El Ford Taurus fue un producto que marcó un hito en la historia de la compañía, tanto por su innovador diseño aerodinámico que cambió el mercado en el que se insertó como por su éxito de ventas con casi 7 millones vendidos a través de 21 años de producción.

Cuando Ford quiso repetir este éxito¹³³, se encontró con que todo el conocimiento de la organización sobre ese proyecto se había esfumado con la partida de los integrantes del equipo. Es decir, no quedaba nadie de aquellos que habían sido responsables.

Lo mismo nos pasa a nosotros en situaciones similares, proyectos difíciles pero exitosos que dejan claro para el equipo que mejor sería trabajar en otra parte. En todos esos casos, la organización se queda con un proyecto exitoso, pero sin la gente que lo hizo posible y sin la capacidad de repetir la experiencia.

La filosofía ágil hace foco en las personas, como elemento fundamental del trabajo intelectual y creativo, en oposición a otras concepciones deshumanizantes que conciben a las personas como piezas intercambiables de una maquinaria. Este nuevo foco modifica no solo la concepción que tenemos del equipo y de la organización, sino también del espacio laboral y de muchas prácticas ampliamente aceptadas en la cultura moderna del trabajo.

Este capítulo presenta nuestra mirada sobre los equipos de desarrollo y cómo estos se vinculan con las organizaciones a las que pertenecen. Una forma de pensar esta relación es entender que tanto los productos como los equipos que los construyeron son el resultado del proyecto llevado a cabo por la organización¹³⁴.

El equipo ideal

Los métodos ágiles se enfocan en el trabajo de equipos desarrollando software, porque la mayoría de los sistemas modernos son construidos por equipos, no por individuos, dado su tamaño y complejidad. En esta sección recorreremos los aspectos más significativos del trabajo en equipo al estilo ágil. Un equipo ideal debe:

- Autoorganizarse.
- Tener responsabilidad solidaria.
- Colaborar.
- Controlar dejando hacer.
- Ser armonioso.
- Ser multidisciplinario y multifuncional.

Autoorganizarse

Esta es una de las características centrales de todo equipo ágil, se menciona explícitamente entre los principios del Manifiesto ágil [Beck 2001]:

“Las mejores arquitecturas, requisitos y diseños emergen de equipos autoorganizados¹³⁵. ”

Un equipo ágil es un equipo autoorganizado, lo cual significa que son los propios miembros quienes organizan su trabajo atendiendo a las restricciones del contexto. Esto implica que:

- No hay una persona que indique a cada miembro qué ni cómo hacer estas tareas necesarias para cumplir con los compromisos que han asumido, sino que los propios miembros del equipo las identifican y distribuyen.
- El equipo no solo se organiza para llevar adelante el trabajo, sino que también monitorea su desempeño y adapta su forma de trabajo para mejorar sus resultados.

Si bien estas recomendaciones pueden resultar chocantes para gente acostumbrada a contextos de comando y control (donde el jefe de proyecto asigna las tareas), la realidad es que los que hacen el trabajo saben hacerlo mejor que sus jefes. Además, como son tareas complejas, con dependencias entre sí y aspectos intelectuales o tecnológicos que requieren conocimiento detallado del tema, no es fácil encontrar jefes con la capacidad adecuada. Aún en casos de aquellos con la experiencia y las habilidades necesarias, es difícil organizar a un grupo de gente exitosamente. En particular, porque requiere conocer las sutiles habilidades de cada uno, su actitud y energía emocional en el día a día y las formas óptimas de comunicación. Además, es común que sea difícil coordinar el trabajo para lidiar con las dependencias y que otros aspectos técnicos de la solución afecten también a la organización.

Por otra parte, ¿cómo esperar que alguien asuma seriamente un compromiso si no se le da la libertad de planificar el trabajo necesario para llevarlo a cabo? [Humphrey 2001]. La forma más fácil de probar esta idea es evaluar el resultado de imponer plazos arbitrarios a un equipo contra permitirle al equipo definir sus propias fechas. En el primer caso, es probable que el equipo acepte las fechas impuestas, porque no tiene otro remedio, pero su motivación para cumplirlas será menor. En el segundo, donde se apropiá de esas fechas porque es el encargado de definirlas, el equipo se comprometerá más profundamente con el resultado.

Es importante no malinterpretar la autoorganización: que el equipo sea autoorganizado no significa que haga lo que le parezca. No estamos diciendo que no hay lugar para la gerencia en un entorno ágil, sino que interviene de otra manera, guiando al equipo, orientando los comportamientos que surjan de su libre interacción, proveyendo foco, removiendo obstáculos y facilitando la interacción con el resto de la organización¹³⁶. La autoorganización tampoco implica que el equipo trabaje aislado¹³⁷. Es importante que mantenga una comunicación fluida con la organización en general y con la gerencia en particular, para que esta tenga visibilidad constante¹³⁸ del avance del proyecto. Esto le permitirá al equipo obtener feedback y generar mayor confianza para cumplir con su responsabilidad.

Tener responsabilidad solidaria

Todos los miembros del equipo deben asumir una responsabilidad compartida por los resultados. El impacto principal de este principio es que los miembros deben dar de sí todo lo posible para lograr los resultados, en oposición a proyectos en los cuales cada uno hace solo lo que le fue asignado. La responsabilidad solidaria asume la autoorganización, puesto que si las tareas son asignadas por

un jefe de proyecto o gerente, entonces este tiene la responsabilidad por el proyecto completo y solo delega las tareas¹³⁹. Una situación cotidiana en la que se observa claramente esta responsabilidad compartida es al final de cada iteración, cuando se muestra el producto construido. En ese momento, todo el equipo participa de la reunión de revisión y da la cara por los resultados obtenidos.

La relación con la gerencia y el apoyo de esta es una condición necesaria pero no suficiente para que un equipo pueda trabajar de forma autoorganizada y con responsabilidad solidaria. Existen otras condiciones internas que deben darse para que esta organización sea posible: el equipo debe compartir un objetivo y una visión, y sus miembros deben estar convencidos de que la mejor forma de alcanzarlo es trabajando colaborativamente asumiendo la responsabilidad por la planificación, la ejecución, el control y la adaptación al contexto de proyecto a medida que este avanza.

Colaborar

Rob Austin y Lee Devin definen la colaboración como la capacidad para reconcebir¹⁴⁰ nuestras propias ideas a la luz de las ideas de los demás. Esta definición es un poco más precisa que la idea que tenemos de ayudar a otros o trabajar en conjunto. Requiere soltura¹⁴¹, es decir, la capacidad de no aferrarse a las propias ideas, de modo tal de poder cambiarlas al escuchar las de los demás.

La colaboración tiene como objetivo generar en el grupo ideas mejores que la mejor de las ideas de los miembros individuales. Para comprender este concepto puede servir imaginar un ejemplo de interacción entre los miembros de un equipo. Supongamos que cada persona propone una idea, y luego todos votan la mejor de las ideas propuestas. La idea elegida solo puede ser tan buena como la mejor idea individual. En los casos en que hay fuertes restricciones, puede ocurrir que haya un empate y no haya acuerdo sobre cuál es la mejor idea. En ese caso, supongamos que se utiliza una solución de compromiso, en la cual cada una de las ideas más votadas resalta algún aspecto en favor de otra idea con la misma cantidad de votos. Con esta forma de acuerdo, lo más probable es que la idea resultante sea incluso peor que las ideas más votadas a partir de las cuales fue concebida.

Por el contrario, en el trabajo colaborativo la idea creada por el equipo es mejor que la mejor de las ideas individuales, porque se nutre de los aspectos buenos de las otras ideas, pero mantiene la coherencia propia de haber sido concebida y reconcebida en lugar de recortada. Las ideas fruto de la colaboración tienen la cualidad curiosa de no ser atribuidas a ninguno de los integrantes del equipo, sino que emergen de su interacción.

Controlar dejando hacer¹⁴²

Como vimos en la sección de autoorganización, la gerencia necesita dar al equipo la libertad para hacer su trabajo de la mejor manera posible. Este mecanismo, como opuesto al mandar y controlar, se conoce como controlar dejando hacer. Mediante el acto de liberar al equipo, la gerencia logra el equipo dé más de sí que ejerciendo presión.

Daniel Pink [Pink 2011] propone que la motivación intrínseca del trabajo proviene de tres aspectos: autonomía, desafío y maestría. La autonomía está satisfecha por la autoorganización y la responsabilidad solidaria. El desafío de crear el producto se incrementa con el desafío de organizar el trabajo, y la maestría implica la búsqueda constante¹⁴³ de mejorar su capacidad para hacer su trabajo.

Esta actitud no aparece solo en la relación del equipo con la gerencia. También implica soltura en el propio equipo, dejar el ego y el miedo para afrontar las posibilidades y llegar más lejos de lo que parecía posible. En un equipo, la soltura se expresa en:

- Gente dispuesta a escuchar activamente.
- Gente dispuesta a cambiar de opinión (que no es lo mismo que ceder).
- Poca tensión puesta en la tarea¹⁴⁴.
- Superiores que confían en el equipo y le dan libertad de maniobra.

La falta de soltura se muestra en:

- Jefes que miran por encima del hombro de las personas del equipo¹⁴⁵.
- Personas que siempre creen tener la razón.

La soltura es una condición fundamental de los equipos ágiles (y de los colaborativos), porque es necesaria para su interacción armónica.

Ser armonioso

Para ser efectivo, un equipo colaborativo debe trabajar en armonía. Esto no significa que no haya conflictos: en un equipo armónico hay y debe haber conflictos sobre el trabajo y la mejor forma de encararlo, pero no debe haber conflictos personales. Los conflictos sobre el trabajo promueven la aparición de nuevas ideas y reducen la probabilidad de estancamiento en sus antiguas formas de trabajo. Los conflictos personales, en cambio, drenan la energía en disputas sin valor y limitan la capacidad de los miembros para tomar decisiones apropiadas al mezclar egos, celos, envidias y otras disfunciones en el trabajo diario.

Una condición fundamental de la armonía es la confianza, tanto interna como en la interacción con el exterior. La confianza se construye a partir historias en común, de recuerdos compartidos y de haber experimentado como el otro es coherente entre su decir y su hacer [Larsen 2009]. Sin embargo, cuando se constituye un equipo nuevo no contamos con esa historia y es necesario trabajar en conjunto. En esos casos, es necesario asumir que el otro es confiable hasta que se demuestre lo contrario [De Marco 2002], y otorgarle nuestra confianza antes de que la hayan merecido. Recíprocamente, solo podremos pedir la confianza de otros si estamos dispuestos a dar la nuestra. Por ejemplo, si nuestro gerente confía en nosotros lo suficiente para aceptar que nuestras estimaciones son honestas, y acepta las fechas de nuestros compromisos por la misma razón, es natural que nosotros confiemos en que va a ayudarnos a quitar los obstáculos que surjan en el camino. Si, por el contrario, duda de nuestro compromiso, es poco probable que podamos exponernos a pedirle ayuda en una situación de riesgo.

La experiencia de trabajar en un equipo consolidado, con una historia común y la capacidad de funcionar armónicamente, será muy satisfactoria para sus miembros y promoverá que hagan su mejor esfuerzo para lograr los objetivos del proyecto.

Ser multidisciplinario y multifuncional

Como en los equipos ágiles la responsabilidad por el proyecto es solidaria, es necesario contar con todas las habilidades necesarias para completarlo. Un ejemplo concreto de esta idea es la participación de los responsables de las pruebas como parte del equipo. El principio es simple, sino se tienen todas las capacidades para realizar el trabajo correctamente, entonces no se puede tener la responsabilidad asociada. En el ejemplo de las pruebas, si el equipo realiza un producto que no pasa las pruebas, entonces no ha

terminado el trabajo y no debe entregarlo.

Por multifuncional entendemos un equipo que integra todas las actividades del proceso de desarrollo: requerimientos, arquitectura, diseño, pruebas y construcción. Por multidisciplinario, que integra las habilidades de diferentes personas: clientes, analistas, diseñadores gráficos, arquitectos, programadores, testers, etc.

La organización para el equipo ideal

El espacio de trabajo

Pasamos alrededor de un tercio del día en nuestro espacio de trabajo, lo cual es demasiado tiempo como para no estar cómodos. Este solo hecho debería ser suficiente justificación para prestar cierta atención al mobiliario de nuestro espacio de trabajo. En un contexto ágil este tema merece aún mayor atención, debido a que hay un mayor foco en las personas. Como hemos mencionado, la comunicación juega un rol muy clave en un contexto ágil y nuestro espacio de trabajo puede facilitar tanto la comunicación como también dificultarla.

Si esperamos que la información fluya, que la propiedad del código sea compartida, que haya confianza entre los miembros del equipo y que el diseño sea emergente, entonces resulta importante que el equipo esté sentado junto, trabajando en la misma habitación. Esto puede parecer obvio para algunos, pero en algunas organizaciones suele ocurrir que hay miembros de un mismo equipo trabajando en salas distintas, o dentro de una misma pero sentados en forma dispersa mezclados con miembros de otros equipos. En [Davies 2009] se recomienda que lo ideal sería que el espacio sea exclusivo del equipo.

Prácticas extremas

Extreme Programming va un poco más allá de las prácticas aquí mencionadas y propone algunas adicionales. Por ejemplo, que el cliente trabaje con el equipo de desarrollo en forma continua, lo cual lo obliga a tener un puesto de trabajo permanente junto al equipo.

Otra práctica de XP es la semana de 40 horas¹⁴⁶, la idea es que el equipo pueda mantener un ritmo de trabajo sostenible, para un equipo puede que sean 40 horas, para otro puede que sean 35 o 45, pero 60 horas semanales no es un ritmo sostenible.

Algunos artefactos de mobiliario típicos de ambientes ágiles suelen ser:

- Pizarra: para que el equipo pueda realizar esquemas a la hora de debatir un diseño, hacer diagramas, anotar recordatorios, etc. Esta pizarra debe estar acompañada de marcadores de varios colores y un borrador.
- Tablero¹⁴⁷: es otra pizarra o solo un espacio reservado en una pared, donde el equipo pega las notas autoadhesivas con sus tareas, story cards, impedimentos, etc. En general, está dividido en columnas donde cada una representa un estado: pendiente, en progreso, completo. Siempre debe estar acompañado de un mazo de notas autoadhesivas. Este tablero se actualiza diariamente a medida que se completan las tareas. Para más detalle sobre el uso y la forma de organización de este tablero, ver el capítulo “Irradiando información”.
- Semáforo de integración: es un indicador del estado del build de integración continua. En el caso más simple, suele ser un monitor que muestra el tablero del servidor de integración continua; en algunos equipos suelen ir más allá y utilizar diversos artefactos para reflejar esta información: sirenas, luces, etc.
- Sillas adicionales: para que puedan sentarse el/los clientes, cuando vengan a trabajar con el equipo.

Figura 15.1. Espacio de trabajo apropiado para un equipo ágil.



Horarios de trabajo

Algunos equipos de trabajo suelen enfrentarse a situaciones del tipo: “esta funcionalidad tiene que estar lista esta semana cueste lo que cueste”. Situaciones de este estilo, en que los compromisos¹⁴⁸ son impuestos y no gestionados por el equipo suelen ser contraproducentes¹⁴⁹ y llevan a que la gente deba trabajar horas adicionales. No está mal hacerlo en forma excepcional, pero si se convierten en una cuestión habitual, entonces estamos en problemas. El tiempo extra en el trabajo reduce el tiempo dedicado a cuestiones personales, lo cual muchas veces dispara diversos inconvenientes, que a la larga también impactan en el proyecto. Los

ambientes ágiles ponen un especial foco en la persona, prestan atención a cuestiones de índole humana como el balance trabajo-vida personal. En este sentido, la práctica ya mencionada de 40 horas semanales propone que bajo ninguna circunstancia un equipo trabaje extra dos semanas seguidas. El trabajo extra en forma continua genera situaciones que resultan insostenibles en el tiempo para cualquier equipo. Si este necesita trabajar extra dos semanas seguidas, entonces el proyecto tiene un problema bastante más grave que va más allá y debe ser resuelto de otra forma (mediante la renegociación de compromisos¹⁵⁰).

Otra cuestión común es el horario de entrada. Mientras que en algunas organizaciones se pide a los empleados llegar a determinada hora, otras adoptan políticas más flexibles. Más allá de lo que la organización requiera o proponga, en un ambiente ágil es conveniente que los miembros del equipo trabajen en la misma banda horaria, ya que posibilita la utilización de ciertas prácticas como programación de a pares y las reuniones diarias entre otras. En general, al comenzar un proyecto el equipo define un horario para la reunión diaria y todos los miembros se comprometen a estar en la oficina en el horario establecido.

En resumen

Uno de los factores principales de éxito en un proyecto de desarrollo de software son las personas. Como vimos, las condiciones para constituir equipos efectivos no son triviales, ni dependen solo de sus miembros. La generación de condiciones favorables de trabajo, desde las instalaciones y el mobiliario hasta los horarios y reglas de convivencia son fundamentales, y requieren el apoyo de la gerencia. Además, las interacciones entre los miembros del equipo, incorporando diferentes miradas y habilidades, deben tejerse cuidadosamente para lograr la colaboración y armonía necesarias para producir los excelentes resultados que se esperan.

¹³³ La historia está tomada de [Brown 2000], pág. 122, quienes a su vez la toman de Davenport y Prusak.

¹³⁴ Rob Austin y Lee Devin llaman a esta característica “Play”, juego de palabras en el inglés original asociado al teatro, donde el producto es la obra (*play*) y la interacción creativa que desarrolla el equipo se concibe como un juego (*play*).

¹³⁵ La versión original en inglés es “The best architectures, requirements, and designs emerge from self-organizing teams”.

¹³⁶ En el capítulo “¿Quién manda a quién?” tratamos en detalle el tema del liderazgo.

¹³⁷ Como muchas veces ocurre en entornos donde el concepto de empowerment se aplica sin criterio orgánico ni actitud colaborativa. Véase [Coplien 2004], pág. 294.

¹³⁸ Véase el capítulo “Ir radiando información”.

¹³⁹ Como expresa la sabiduría popular: “la responsabilidad no se delega”.

¹⁴⁰ El término “reconcebir” fue acuñado por Lee Devin. Véase [Austin 2003], pág. 101.

¹⁴¹ “Release” en el inglés original, véase [Austin 2003]. Preferimos soltura a la traducción alternativa “liberación”, que también es válida. Véase la sección “Controlar dejando hacer”.

¹⁴² En inglés “Control by release”.

¹⁴³ Vale aclarar que este es un buen punto a considerar a la hora de contratar, no algo que todas las personas compartan en la misma medida.

¹⁴⁴ Como en la premisa budista de “atención sin tensión”.

¹⁴⁵ En los equipos colaborativos, por lo menos, la mirada del dueño no engorda el ganado.

¹⁴⁶ Semana de 40 horas es la traducción de los autores para 40 hour-week.

¹⁴⁷ Algunos equipos utilizan como tablero una pizarra magnética. La realidad es que la herramienta no es relevante, sino que lo importante es que la información esté visible.

¹⁴⁸ Véase el capítulo “Planificación constante” para un desarrollo más preciso de cómo recomendamos establecer compromisos.

¹⁴⁹ De acuerdo a Mary Beth Chrissis, Watts Humphrey solía decir que podía conocer en media hora el nivel de madurez de cualquier organización preguntando a la gente que hacía el trabajo cómo establecía sus compromisos [Chrissis 2012].

¹⁵⁰ Véase el patrón organizacional “Recommitment Meeting” en [Coplien 2004], pág. 60.

La fantasía de evitar los cambios

En un proyecto de desarrollo de una aplicación de gestión de importaciones que está en una etapa avanzada, el patrocinador del proyecto, Pedro, se acerca al responsable comercial de la empresa que estaba construyendo el sistema, Pablo.

—Hola, Pedro, ¿qué sucede? Se te ve cara de preocupación —dice Pablo.

—Es que tengo muy malas noticias. Hubo un cambio inesperado en la legislación que regula las importaciones y esto trae muchos cambios respecto a definiciones que establecimos al inicio del proyecto.

—Sí, entiendo, trabajando en este rubro es lógico que surjan cambios que están fuera de tu control —responde Pablo.

—Es que siento que vamos a tener que arrancar de nuevo. Este cambio es transversal a muchos procesos, así que tenemos que arrancar la planificación nuevamente —se lamenta Pedro.

—En la forma en que trabajamos está previsto que esto pueda suceder, dejame que te explique como seguimos. ¡Carlos [el representante del cliente en el equipo] está al tanto de esto? —pregunta Pablo.

—Sí. Está estudiando el tema en este momento; para mañana por la tarde ya va a estar empapado en el asunto.

—Bien, ahora voy a hablar con Diego, el facilitador, que está ayudando al equipo, ya que esto es un cambio importante. Hoy es miércoles y los viernes son los días en que se revisa lo hecho en la semana y se planifica el trabajo de la próxima —explica Pablo, intentando utilizar lo menos posible la jerga técnica, que Pedro no entiende del todo— Diego va a hablar con el equipo para que se reserven un tiempo mayor al habitual para la reunión de los viernes, así hay tiempo de revisar el impacto de todo y replanificar lo que sea necesario.

—Por suerte —continúa Pablo— venimos trabajando hace dos meses en el proyecto, y el equipo ya tiene bien afinado el proceso de estimación y está conociendo bien el negocio. Además, trabajan muy bien con Carlos en esto. Para última hora del viernes vamos a tener una buena idea del impacto del imprevisto. Nuestra forma de trabajo presupone que va a haber cambios, no vamos a necesitar salir del ritmo de trabajo para atender este tema.

Pedro ya mostraba alivio en su expresión, pero le quedaba una duda aún:

—Pablo, gracias por tomar este tema y aclararme como se tratará, pero me preocupa un poco como afecta al contrato...

—Te propongo algo: el viernes voy a trabajar con el equipo y con Carlos sobre este tema. Ya con toda la información y el análisis del impacto, nos reunimos a almorzar el lunes y vemos los temas del contrato.

—Me parece bien —responde Pedro, ya más tranquilo— ¿en el lugar de siempre? Pero esta vez pago yo...

—Eso ya lo veremos —responde Pablo, contento de haber cambiado el humor de su cliente.

El cambio es una realidad

Es común en los proyectos de desarrollo de software (incluso en algunos que pretenden trabajar en forma ágil) hacer un gran énfasis en tener el alcance detallado y acordado con el cliente lo más temprano posible. En parte esto viene de las prácticas y metodologías donde la primera etapa era definir el *qué* y luego nos ocupamos del *cómo*. En parte, también tiene que ver con tener un compromiso a cumplir (un contrato) y una cantidad de recursos definidos para cumplirlo (precio, equipo y tiempo).

La realidad ha mostrado una y otra vez que esto rara vez funciona. O bien las necesidades detalladas del proyecto no son como las piensa el cliente, y se cae en cuenta de como debe ser recién al tener una parte desarrollada, o bien el contexto cambia y es necesario un cambio en las especificaciones originales para que el desarrollo no pase a ser irrelevante.

En este capítulo veremos como el agilismo maneja los cambios. Si bien esto ha sido un tema tratado por diferentes metodologías desde los inicios del desarrollo de software como disciplina (formal o informal), ahora la necesidad de adaptación al cambio de proyectos relacionados con nuevas tecnologías lo hace un tema cada vez más relevante. Como ejemplo tenemos las redes sociales y aplicaciones para dispositivos móviles, que en el tiempo que transcurre desde que se está escribiendo este libro hasta su publicación ya habrán sufrió cambios de relevancia.

El agilismo ante los cambios

El agilismo propone, desde sus comienzos y como una parte fundamental de su esencia, tomar el cambio como un hecho que va a ocurrir dentro de los proyectos y no como una excepción.

A diferencia de otros métodos de desarrollo, donde se intenta controlar el cambio (y esto se convierte, explícita o implícitamente, en un intento de minimizarlo), el agilismo se propone como misión responder positivamente a los cambios, entendiendo que el éxito del proyecto depende de incorporarlos y no de evitarlos.

Para esto incorpora en sus prácticas para trabajar en forma ordenada en un entorno de constante cambio, cuidando que este no provoque interrupciones en el trabajo, sino definiendo precisamente donde y como incorporarlo dentro de las prácticas.

Prácticas para integrar el cambio

Backlog vivo

El artefacto clave que nos guía en el alcance en un proyecto ágil es el backlog, el cual contiene (entre otras cosas) las user stories y la estimación de esfuerzo (medida en story points, por ejemplo) de cada uno de ellos. Esta simplicidad los convierte en elementos fácilmente intercambiables.

Por ejemplo, al darse cuenta el cliente que necesita introducir una nueva funcionalidad para una fecha de lanzamiento determinada, puede buscar una o varias user stories que sumen la misma cantidad de story points y reemplazarlas. Si quiere adelantar una funcionalidad planeada para más adelante, puede intercambiarla con otras que sumen la misma cantidad de story points.

Esta versatilidad nos permite ofrecer al cliente una herramienta donde vea el impacto de cada cambio sin tener que entrar en una negociación cada vez que se pide uno.

Por supuesto que esto no puede hacerse en cualquier momento. Introducir cambios durante el desarrollo de una user story en particular no es buena idea, ni tampoco lo es estar cambiando el Backlog todos los días. Para esto hay momentos definidos, relacionados con otros artefactos y prácticas, que ordenan la forma de tratar los cambios.

Reunión de planificación en cada iteración

Así como en todas las metodologías el momento de definir el alcance es durante la planificación, también lo es en el desarrollo ágil.

La característica principal es que en la reunión de planificación (de iteración o de lanzamiento) el usuario tiene la posibilidad de intercambiar user stories, introduciendo cambios en el proyecto. Al hacerlo, la replanificación se hará más compleja, pero es el momento indicado ya que todo el equipo se encuentra en un momento de replanificación y reestimación, y por ende con la mente abierta a nuevas definiciones.

En los proyectos ágiles, es crítico que se respete el momento establecido para introducir cambios. Para mantener al equipo trabajando en forma eficiente, no se deben interrumpir los procesos mentales creativos que requiere el desarrollo de software. Por eso, durante la iteración se trabaja para crear lo ya planificado, siendo un trabajo de detalle, y durante los momentos de planificación se trabaja sobre cambios y nuevas funcionalidades.

En aquellos casos en los que un cambio no puede esperar al final de la iteración, y esto afecta o hace irrelevante parte del trabajo que se está desarrollando en esa misma iteración, puede interrumpirse por completo la iteración y comenzar con el trabajo de planificación. Es importante que esto sea algo excepcional y que se involucre a todo el equipo, no separar a unos pocos para replanificar y que el resto siga con sus tareas. Separar al equipo rompería las sinergias y sería más disruptivo que realizar una replanificación general.

Cliente presente en el equipo

El tener al cliente (o alguien que conozca y represente sus intereses) como parte del equipo es fundamental para que los cambios aparezcan lo más temprano posible durante el proyecto. Con el cliente viendo el proceso día a día, validando las user stories a medida que son desarrolladas y haciendo revisiones integrales al proyecto, se facilita la detección temprana de necesidades de alterar o agregar funcionalidad al sistema.

En este sentido, es necesario crear un ambiente de colaboración en el equipo. El cliente debe entender que los cambios podrán tener consecuencias en el desarrollo del proyecto y los desarrolladores no deben mostrar una actitud negativa hacia los cambios propuestos. A su vez, el equipo deberá ser sincero y abierto en explicar las consecuencias de introducir los cambios que el cliente solicite.

Iteraciones cortas

Para que un proceso de desarrollo ágil sea iterativo y por incrementos, es preferible que las iteraciones sean relativamente cortas. Plantear iteraciones de dos meses en un proyecto de medio año de duración, haría que para el final de la primera iteración haya pasado ya un tercio del total. De esa forma, ante cualquier cambio de importancia, lo más probable es que haya consecuencias enormes.

Las iteraciones cortas permiten introducir cambios y reordenar prioridades a intervalos regulares. De todas maneras, las iteraciones no deberían ser tan cortas que el desarrollo se vea interrumpido y no llegue a un ritmo de trabajo eficiente.

Consideramos que en proyectos medianos a grandes, lo ideal son iteraciones de dos o tres semanas, siendo las de una semana reservadas para proyectos pequeños o en entornos de alta tasa de cambio (por ejemplo el desarrollo y mantenimiento de juegos sociales y sitios públicos masivos).

Revisión de la iteración

Este es un hito clave para validar con el usuario y otros referentes del proyecto que realmente se está construyendo lo que necesitan. Hacer la demo al final de cada iteración facilita que los cambios necesarios se puedan detectar y plantear al equipo tempranamente.

Una particularidad de este hito es la posible participación de actores que no son parte del equipo. Por ello, los pedidos de cambio que aparecen pueden ser disruptivos respecto al trabajo del día a día. Es importante evitar tomar esto en forma negativa, ya que es una forma de validación externa al equipo fundamental para asegurar construir el producto correcto.

Los cambios pedidos o sugeridos en las demos se deben analizar en la reunión de planificación siguiente. Si el impacto del cambio es grande, quizás amerite un análisis de impacto previo a la reunión de planificación, para validar si se decide o no implementar, en especial si fue propuesto por alguien que no participa de la dinámica habitual del proyecto.

Spikes

Como vimos en el capítulo “Estimar no es predecir”, cuando se solicita al equipo un cambio cuyo impacto no puede estimarse, por ejemplo porque implica una implementación con una complejidad técnica poco conocida, es recomendable plantear un spike para investigar esto durante la iteración.

Con esta herramienta se evita estimar algo desconocido, posponiendo la definición hasta tener mayor información sobre el tema. La incertidumbre es muchas veces un motivo por el cual se evitan los cambios importantes, que pueden a su vez serlo para el proyecto desde el punto de vista del cliente. Los spikes nos dan una forma de administrar y explicitar esta incertidumbre e incorporarla al flujo del proyecto.

Cambiar temprano y no tomar decisiones apresuradas

Una de las claves de adaptación a los cambios en un proyecto ágil es detectarlos en forma temprana. Es importante que el equipo vea el cambio como parte normal del proyecto y ser conscientes de no intentar evitar los referidos a las necesidades del negocio.

A su vez, para ser flexible al cambio, es fundamental no tomar decisiones tempranas en el proyecto que impidan realizarlos, sino posponerlas hasta que se cuenta con la información necesaria para tomar una decisión acertada. Prever cosas que no son parte del alcance de la iteración en curso, puede implicar, además de estar haciendo un trabajo que nunca llegará a ser necesario, construir elementos que luego haya que deshacer para implementar cambios que aparecen en las siguientes iteraciones.

Para esto, una recomendación es posponer las decisiones que tienen grandes consecuencias hasta el último momento prudente. Esto implica posponer las decisiones hasta el momento en que posponerlos aún más provoque problemas en el desarrollo del proyecto.

Muy lindo lo de aceptar los cambios, pero el contrato dice otra cosa

Claro que los cambios implican no solo un impacto en el proyecto sino que, en muchos casos, tienen impacto en los compromisos ya acordados. Aquí el trabajo recae en el comercial y el cliente, quienes deberán hacer llegar al equipo las decisiones tomadas.

En muchos casos, la decisión de incluir un cambio implica unos pasos administrativos por fuera del trabajo del equipo. En estas situaciones, es recomendable desacoplar este proceso de las iteraciones del proyecto. Si el cambio aparece durante una iteración, sería importante tener el resultado del proceso para la siguiente reunión de planificación. Si surge durante la revisión o la reunión de planificación, llegar a la decisión del cambio puede ser una tarea más de la iteración, que involucrará al líder y al cliente, y quizás alguien que realice un análisis del impacto desde el punto de vista técnico. En este último caso, deberá preverse como una tarea más de la iteración y reservarse tiempo para ella.

Es común que los cambios en un proyecto se quieran evitar por la complejidad que implican en la administración del proyecto. Los

métodos ágiles nos dan herramientas para incorporarlo dentro del trabajo, pero no podremos evitar la burocracia propia de las organizaciones y los contratos. Para esto se deberá limitar el impacto, aislando al equipo de esta burocracia todo lo que se pueda, y dejándolo liberado para concentrarse en desarrollar software en forma eficiente.

En resumen

El cambio es algo inevitable en los proyectos de software. Dicho de otra manera, es algo con lo que siempre podemos contar. El desarrollo ágil provee herramientas para administrar un proyecto tomando los cambios, no como una excepción, sino como parte normal del flujo de trabajo.

Para esto, lo incorpora donde más naturalmente se asimila, en el de iteración. En este sentido, requiere disciplina en el equipo para tener en cada caso la mentalidad correcta. Durante la iteración se implementa lo planificado y acordado, y entre iteraciones (en la reunión de planificación) se debe estar con la mente abierta para permitir cuestionar lo que se está desarrollando y el orden en que se va a realizar.

De esta forma, el agilismo se presta para el entorno cada vez más cambiante de los proyectos de software, donde la velocidad y adaptabilidad no solo son importantes, sino esenciales para el éxito de un proyecto.

Formalizando compromisos

En una reunión de inicio de un proyecto, el gerente de desarrollo presentaba al equipo la naturaleza del proyecto y el negocio. Cuando llegó la parte de la presentación referida a la organización del trabajo, afirmó:

—En el proyecto seguiremos la metodología [nombre de una conocida metodología ágil], porque creemos que desde la agilidad vamos a maximizar el valor que agregamos al cliente.

La presentación siguió con otros detalles y sobre el final de la misma se escuchó decir al gerente:

—Tengamos en cuenta que es crítico cumplir con cada uno de los puntos establecidos en el alcance, y con la mayor exactitud posible. El incumplimiento de cualquiera de los puntos puede traer graves consecuencias económicas... además, necesitamos evitar cualquier tipo de cambio. La negociación del contrato fue muy dura y no queremos volver sobre estos temas, ya que provocará mucho desgaste en el equipo comercial y gerencial.

Desde el fondo, tímidamente un miembro del equipo esbozó una inquietante pregunta

—¿No es que este sería un proyecto ágil...?

El gerente, rápidamente y con absoluta convicción, contestó:

—Por supuesto. La agilidad es una directiva estratégica de la empresa. Este proyecto es de alta visibilidad y va a ser modelo de cómo implementamos métodos ágiles en la organización.

El silencio en la sala era ensordecedor...

Los contratos y compromisos de diferente naturaleza son parte de la vida diaria de los proyectos. Muchas veces, quienes escriben, negocian y aprueban contratos no entienden (aunque muchas veces creen que sí) de desarrollo ágil. En este capítulo veremos algunas de las formas de contratos y acuerdos típicos, y su relación con los métodos ágiles.

En la vida profesional es habitual intervenir de alguna forma en la creación, control y cumplimiento de contratos y acuerdos. Algunos de nuestros lectores quizás hayan pasado al *lado oscuro* (como gerencias o ventas), y trabajen diariamente con estos temas. A ellos los queremos alertar sobre los desafíos de generar acuerdos compatibles con los valores de la agilidad expuestos en este libro.

Tipos de contratos

En el desarrollo de software es común encontrarse con diversas formas de formalizar los compromisos que se establecen con los clientes tanto externos como internos. Ello ocurre más aún cuando trabajamos en una organización que brinda servicios de desarrollo.

Hay dos situaciones que podríamos considerar casos extremos, habiendo muchos mixtos. En un extremo podemos considerar a los contratos de tiempo y materiales y, en el otro, a los de precio y alcance fijos.

En principio, los contratos de tiempo y materiales son los más compatibles con el desarrollo ágil, ya que cuentan con flexibilidad total. Sin embargo, tienen escollos que merecen ser analizados.

Los contratos de precio y alcance fijos, en cambio, no se llevan bien con la agilidad¹⁵¹, ya que tratan los cambios como una excepción, definiendo y comprometiendo el alcance completo (y el esfuerzo que se pretende para llevarlo adelante) antes de comenzar el trabajo.

Existen, además, situaciones intermedias, pero analizando estos casos y algunas variantes comunes, creemos que podremos hacernos una buena idea sobre la relación entre los contratos y el agilismo.

Contratos de tiempo y materiales¹⁵²

En estos contratos usualmente se acuerda un equipo que se dedicará al proyecto, tanto en cantidad de gente como en los perfiles y responsabilidades. Algunas veces hay objetivos de alto nivel para el producto a desarrollar, pero sin ser vinculantes.

Se considera que son contratos muy compatibles con un proyecto ágil¹⁵³. Por un lado, permiten que el cliente vaya llevando el proyecto hacia donde mayor valor aporte para sus necesidades. Por el otro, el equipo puede estimar y planificar su trabajo en forma transparente, acordando los alcances y los plazos de entrega junto con el cliente de manera de llegar a acuerdos productivos.

En proyectos bajo el marco de estos contratos, la adaptación al cambio se da naturalmente, ya que el cliente puede solicitar un cambio (y hasta cambiar el rumbo del proyecto) en cualquier iteración, sin que esto accione alarmas en los responsables de gestión de proyecto, que usualmente son quienes deben tener el ojo sobre el alcance y el presupuesto.

Todo muy lindo, pero...

En teoría, este tipo de proyectos evitan discusiones contractuales. La realidad es que, a pesar de contratar un equipo por horas, el cliente siempre tiene una expectativa de entrega, tanto en alcance, como en plazos. Y los presupuestos nunca son infinitos...

Un problema muy común en proyectos bajo el marco de este tipo de contratos es que se comienza de forma idílica, donde el cliente va haciendo brainstorming con el equipo, probando ideas y se siente libre de ir modificando el alcance casi como un artista. Esto sucede especialmente cuando el cliente conoce del negocio, pero no tiene disciplina de proyecto (informáticos o de cualquier otro tipo).

Esto está muy bien, pero a veces llega un momento que el cliente mismo quiere ver su producto completo, a veces por su propia ansiedad, otras porque se percata de una fecha que se aproxima, o sus propios superiores (o sus clientes) le piden ver el producto. En ese momento es cuando suele cambiar la relación. El cliente comienza a dar por supuesto que debería haber algo –con mayor avance–, más –completo–, o hasta puede cuestionar la manera en que se han usado el tiempo y los recursos del equipo.

En los casos en que el cliente ingresa en esta fase de quejas, el equipo se suele sorprender, al punto de no entender de qué les están hablando. Al fin y al cabo, se hizo todo lo que pidió. Incluso está todo documentado: las reuniones de planificación actualizaron el backlog luego de cada iteración y allí se cambiaron las prioridades cada vez que fue necesario, en todos los casos con su anuencia. Siempre hubo absoluta visibilidad de todo el trabajo que se estaba haciendo. El cliente no solo estaba enterado y daba el visto bueno: ¡todo lo que se hizo fue a su pedido!

Si vamos a lo que está escrito, el equipo indudablemente tiene razón: se ha cumplido al pie de la letra... Aún así, también es una realidad que el cliente no obtuvo lo que esperaba, en el tiempo que esperaba, y con los recursos que esperaba. Y eso encamina los proyectos al fracaso, sin importar quién tenga razón.

Estos contratos de tiempo y materiales, tan comunes en contextos ágiles, no son siempre bien vistos desde otras perspectivas, o por

gente que no los ha usado.

Por ejemplo, muchos enfoques basados en el PMBOK [PMBOK4] sostienen que los contratos de tiempo y materiales solo son apropiados para proyectos pequeños y que requieren un gran trabajo de control sobre el contratado.

El primero es un prejuicio liso y llano, cuya invalidez se ha demostrado a lo largo de cientos de proyectos ágiles, pequeños, medianos y grandes.

El segundo no tiene en cuenta que, en el marco del desarrollo ágil, hay un vínculo de confianza entre el cliente y el equipo de desarrollo que se asume como punto de partida. Además, el mayor involucramiento del cliente en comparación con metodologías tradicionales, también hace que ese trabajo de control sea en realidad parte de un contexto colaborativo esencial al desarrollo ágil.

Asimismo, ocurre a menudo que el equipo es parte de una organización externa al cliente, con la forma de una empresa de servicios.

En estos casos, muchos gerentes de las empresas de servicios plantean que los contratos de tiempo y materiales les dejan poco margen para lograr beneficios económicos derivados de una mayor productividad.

Si bien puede ser cierta esa objeción, nuestra experiencia indica que no es una buena idea buscar el resultado económico inmediato.

En algunos proyectos en los que hemos trabajado, el valor de venta de la hora de desarrollo se ha podido aumentar en proyectos posteriores, gracias al vínculo de confianza obtenido y a la calidad lograda trabajando de esta manera.

¿Entonces...?

Lo que se puede hacer para mitigar este riesgo es realizar un trabajo importante de gestión del proyecto, consistente en mantener al cliente enfocado en sus propios objetivos y reforzar la visibilidad. De esta manera, el escenario descripto arriba se evita, sin salirnos de los fundamentos de la agilidad.

Una virtud importante del desarrollo ágil es que es predecible. Luego de cada reunión de planificación, es importante hablar con el cliente y mostrarle la distancia a la cual están los hitos que son importantes para él. Estos pueden ser versiones para mostrar (cuando aún tiene que vender la idea), versiones beta, la versión entregable. En el caso de productos ya desarrollados y en mantenimiento evolutivo, podrían ser las fechas en que se prevé que estarán disponibles funcionalidades importantes o determinados arreglos de defectos.

Enfatizando esto con el cliente (entendido no solo quien cumpla el rol de responsable de producto en el proyecto, sino incluyendo a los interesados) es posible evitar esta clásica trampa de los proyectos con contrato tiempo y materiales.

Contratos de precio fijo

Suelen denominarse así aquellos contratos en los que el entregable es fijo (en tiempo y en precio) y queda librado al criterio del contratista los recursos que se utilizarán para cumplir con el contrato.

En su forma típica, las premisas básicas de estos contratos no se llevan bien con los principios del desarrollo ágil.

Ello ocurre porque se pretende especificar todo el proyecto de antemano, o al menos los parámetros más críticos: tiempo total, alcance total, recursos globales totales (en general, expresadas en un valor monetario).

Además, los cambios son vistos como un problema, ya que potencialmente provocan modificaciones en los tres parámetros principales del proyecto, y requerirían una renegociación de los términos. Es usual que se estipulen mecanismos burocráticos para control de cambios (como si los pudiéramos controlar).

Aquí tenemos, a primera vista, dos caminos posibles: renunciar al desarrollo ágil o renunciar al proyecto. La segunda no es siempre deseable o posible. Por suerte existen algunas formas de mitigar los riesgos propios de estos proyectos, tanto durante la generación de la propuesta (que se transformará en el contrato) como la ejecución del mismo.

Estimación

La estimación inicial en un proyecto de precio fijo es toda una disyuntiva. Si la estimación es demasiado conservadora (alta), es muy probable que perdamos el proyecto; si es demasiado audaz (baja), perdemos dinero. En general, en esta etapa es común que intervenga el equipo comercial, pidiendo una estimación más baja, para poder ganar el negocio. Esto es un error: las estimaciones, por su propia definición, no son negociables. Ceder ante esta presión no solo hará que se pierda dinero en el proyecto sino que hará que el trabajo sea terriblemente arduo, con grandes probabilidades de transformarlo en un esfuerzo insostenible y sin sentido, fomentando la deserción de los miembros. Una verdadera situación de perder o perder...

Una forma de mitigar esto es mantener la estimación siempre realista, es decir, realizarla de acuerdo a las prácticas propuestas en este libro, donde el equipo toma responsabilidad en el tema. En este contexto, se puede separar la estimación real del precio objetivo del área comercial. Si para ganar el negocio se va a un valor muy bajo respecto a los parámetros normales de ganancias en la organización, esto debe quedar totalmente en claro y transparente.

En las etapas de preventa, las estimaciones deben tomarse como datos, no como variables a ajustar. En todo caso, puede ajustarse el beneficio esperado (aumentando los recursos asignados), pero en ningún caso deben reducirse los tiempos o recursos para un determinado alcance, con objetivos comerciales.

Partir en porciones más pequeñas

Ya hemos visto en el capítulo “Iterativo por naturaleza” cómo partir un proyecto en partes más pequeñas reduce la incertidumbre y facilita el desarrollo. Siempre que se pueda, es positivo subdividir un proyecto de precio fijo en varios más pequeños, y mejor aún si el contrato de cada uno se negocia al finalizar el anterior.

Aislar al equipo de las tensiones contractuales

El equipo de desarrollo ya tiene bastante de qué ocuparse. Es mejor que no se los responsabilice por cosas que no pueden controlar ni sobre las cuales no tienen responsabilidad. Si bien es importante que todos tengan una visión general del proyecto, no por eso se debe permitir que las presiones de los diferentes actores dentro de la organización le lleguen en forma directa.

Foco en la planificación

En proyectos de precio fijo hay un trabajo muy intenso sobre el alcance y la planificación. Debería estimarse mucho trabajo en este sentido, y los responsables de llevarlo a cabo deberán estar atentos a cualquier desvío, alertando lo más tempranamente posible sobre los problemas que pueden (y que seguramente van a) surgir. Estos son los momentos en los cuales, desde las gerencias superiores se suele recurrir a ponerse al hombro el proyecto. Esto debe resistirse y una de las maneras es con una alerta lo más temprana posible

sobre los desvíos.

En fin...

Los contratos de precio fijo no son aptos para desarrollar en forma ágil pura. La realidad es que vamos a encontrarnos con proyectos de este tipo pues no van a desaparecer. Es tarea de los gerentes de proyecto promover los principios del agilismo mientras se cumple con las obligaciones organizacionales. Hasta que punto es posible esto, y cuando hay que renunciar a llevar adelante el proyecto en forma ágil, es imposible decir, cada caso es diferente... y decidir que camino tomar es parte de nuestro arte.

Algunos grises

Contratos de tiempo y materiales con tope [154](#)

En algunos casos, en vista de los problemas que se describen cuando hablamos de proyectos de tiempo y materiales, se introduce un tope en la cantidad de recursos a utilizar, en general para ceder ante ansiedades (lógicas) del cliente que siente que está firmando un cheque en blanco.

No es raro que este tope venga junto con un objetivo a cumplir en cuanto a alcance. Esta es una variante (no tan rara) que puede clasificarse como la peor opción posible para quien desarrolla el servicio. El alcance está (mayormente) fijo, se cobran las horas de las personas asignadas, con visibilidad para el cliente. Pero si se acaban los recursos hay que entregar el producto igual.

Esta variante suena descabellada y es algo a evitar a toda costa, pero las áreas comerciales son muy creativas y contratos como estos aparecen con frecuencia.

Licitaciones

Si bien son otro tipo de contratos, tienen particularidades que merecen una mención especial.

Por definición, son una competencia entre varios proveedores, haciendo la presión de ir a un bajo precio, aún mayor de lo habitual. En los casos de precio fijo, esto lleva a mayores presiones para bajar o afinar la estimación. ¡A no ceder ante estas presiones! Las estimaciones deben estar ajustadas al análisis técnico. Bajar el precio es una decisión comercial que puede ser perfectamente válida, pero para esto no debe modificarse la estimación, ya que de esa forma se perdería contacto con la realidad.

En general, responder a licitaciones no es una tarea grata, pero es parte del mundo de los negocios de desarrollo de software.

Un contrato ágil

Hemos visto algunas características de contratos típicos que suelen manifestarse en la industria. Ahora bien, si tenemos capacidad de decisión sobre la forma que tomará un contrato (tanto siendo la parte contratante como la contratista), y queremos un proyecto ágil, ¿qué modelo deberíamos elegir?

No existe un modelo de contrato ágil, pero podemos enumerar características que hacen que un contrato sea compatible con este desarrollo.

Alcance

Es usual que un cliente quiera comprometer un alcance en el contrato. Aquí conviene establecer un objetivo en cuanto al producto a desarrollar, planteándolo en forma compartida entre contratista y contratante. Traer al cliente dentro del equipo es fundamental para lograr una forma de trabajo colaborativa y evitar que se genere un antagonismo cliente-proveedor.

Compromisos

Es conveniente establecer el compromiso en términos de las características del equipo (composición, dedicación, capacidades) y la forma de trabajo, incluyendo los compromisos del cliente en idéntico sentido. Además de servir para ser exigido durante el proyecto, y como protección ante reclamos de retrasos que son imputables al cliente, afianza la noción del equipo colaborativo, aclarando por escrito que todos los involucrados tienen compromisos firmes a cumplir.

En el desarrollo de un proyecto, un cliente que incumple compromisos contractuales y es alertado sobre ello (no como reclamo, sino en forma objetiva dentro del control del proyecto), luego es mucho más propenso a aceptar un retraso del equipo o algún problema en el desarrollo.

Estimaciones

Es muy probable que en los acuerdos haya secciones que contengan alcances y tiempos en base a estimaciones. En estos casos, es ideal que estas sean tomadas como lineamientos generales, como un marco de trabajo y no como un compromiso.

Como se ve en el capítulo “Estimar no es predecir”, es difícil que las estimaciones (en el sentido clásico de la ingeniería de software) hechas al inicio de un proyecto expresen la realidad con la que nos encontraremos después. En parte porque hay elementos del proyecto que no son conocidos en el momento de realizar la estimación, y también por los cambios que se dan en los proyectos entre el momento de estimar y el de realizar el trabajo, como vimos en el capítulo “La fantasía de evitar los cambios”.

El equipo de ventas

La tendencia natural del equipo de ventas es cerrar el acuerdo cuanto antes. Idealmente, el equipo de ventas debería estar convencido de las bondades que representa para el cliente el desarrollo ágil y ayudar a persuadirlos en este sentido.

Es difícil hacer desarrollo ágil si el equipo de ventas no está convencido de las ventajas.

Esto no siempre es así, y en muchas ocasiones los vendedores ceden a las presiones del cliente. Esto lleva a que se coloquen cláusulas que se contraponen con el espíritu ágil que se desea llevar al proyecto. En estos casos es necesario convencer al equipo de ventas. El éxito de esto depende en parte de nuestro esfuerzo, pero también mucho de la cultura de la organización en la cual estamos trabajando.[155](#).

En resumen

Muchas formas habituales de contratación no se llevan del todo bien con el desarrollo ágil. Por eso, al comenzar un proyecto, es importante tener en cuenta las características del contrato (explícito e implícito) y su relación con las prácticas ágiles que se implementarán.

En los casos en los que tenemos injerencia en la formulación de los términos contractuales, es fundamental orientarlos para que promuevan las prácticas que implementaremos en el proyecto.

Cuando se habla de contratos y desarrollo ágil, es común la recomendación de educar al cliente para que los términos vayan de la

mano con la agilidad. Incluso hay quienes recomiendan rechazar proyectos que imponen condiciones que se contraponen a esta forma de desarrollo.

Pero esto último no siempre es posible. A veces, nuestros clientes no son permeables a que los eduquemos; en otras ocasiones, nuestros interlocutores pueden tener poco o nada de injerencia en las formas de contratación. También, la capacidad de rechazar un proyecto no está siempre en nuestras manos.

En muchas ocasiones el desarrollo del proyecto simplemente no podrá ser ágil. Será cuestión de trabajar con algún método iterativo que sea compatible con las organizaciones que intervienen el proyecto. Dicho esto, no debemos dejar que el espíritu del contrato se traslade al funcionamiento del equipo. El hecho de que un proyecto tenga penalidades por incumplimiento, no significa que debamos trasladar a penalidades hacia el equipo. Asimismo, cuando hay problemas para llegar con el desarrollo en las fechas comprometidas, no debemos caer en la tentación de dejar de lado el time-boxing, imponiendo al equipo un cumplimiento de plazos que le es totalmente ajeno.

Los contratos son una parte integral de los proyectos de desarrollo de software. No hay una fórmula mágica con la cual podremos dejar feliz a cualquier cliente, que sea factible de desarrollarse en forma ágil y que tanto los compromisos y las variaciones que puedan surgir durante su ejecución estén claros para todos. Es importante conocer la relación entre los contratos y la modalidad de desarrollo. De esta forma cada uno, desde su rol, comprende la problemática en juego y puede alertar sobre los posibles conflictos con la antelación necesaria para evitarlos.

[151](#) Como expresa Martin Fowler en [Fowler 2003b].

[152](#) La expresión se generó en la industria de la construcción, de allí la referencia a “materiales”.

[153](#) Véase, por ejemplo, [Arbogast Larman Vodde 2012].

[154](#) En inglés se los suele llamar “capped time and materials”.

[155](#) Muchas veces es valioso, aunque difícil, hacer un esfuerzo adicional en pos de cambiar la cultura organizacional.

Construyendo con calidad día tras día

—y como siempre en nuestros proyectos, no vamos a entregar nada que no cumpla con los más altos estándares de calidad —afirmó el director en la reunión de inicio de proyecto, y agregó— Como además tenemos un cronograma de proyecto extremadamente ajustado, nos estará asistiendo el equipo de control de calidad de la organización.

En otro momento de la presentación, el director se dirigió al gerente de proyecto:

—Poncio, tu tarea será crítica, fuiste elegido para este proyecto por tu capacidad de entender las directivas de la gerencia. Las fechas del proyecto y sus alcances son inamovibles, debemos entregar las funcionalidades a tiempo a toda costa. Luego tendremos tiempo, junto con el equipo de control de calidad, de emprolijar las cosas.

Una parte del equipo intentó disimular sus caras largas. Los más experimentados ni se molestaron, nadie podía pretender que estén de acuerdo con la forma de trabajo que se les planteaba. Todos vislumbraron un proyecto tedioso y del cual les costaría sentirse orgullosos.

A ningún equipo le gusta hacer las cosas mal. En ocasiones hay que tomar algún atajo, que generará una deuda técnica¹⁵⁶, pero en general todos esperan que en sus proyectos esto sea una excepción, y no una directiva de trabajo.

Pretender lograr un producto de calidad utilizando como herramienta principal una etapa de control de calidad al final es imposible, además de tedioso y desmoralizante. En este capítulo veremos cómo se enfoca la calidad en los métodos ágiles, donde no se controla en una etapa separada del desarrollo, sino que se busca que las diferentes prácticas contribuyan a construir software de calidad, evitando defectos y facilitando encontrarlos lo más cercanamente posible al momento en el que se introducen.

Qué entendemos por calidad

Algunas definiciones

Según la Real Academia Española, la calidad es “Propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor”.

Según Pressman [Pressman 1998], la calidad de software es la concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados, y con las características implícitas que se espera de todo software desarrollado de manera profesional.

La definición oficial del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE, Std. 610-1990) es “La calidad del software es el grado con el que un sistema, componente o proceso cumple los requerimientos especificados y las necesidades o expectativas del cliente o usuario”.

Calidad intrínseca y extrínseca

En este capítulo vamos a diferenciar dos aspectos en los cuales podemos dividir las características de calidad del software. Un aspecto es el extrínseco, referido a las características también llamadas externas, que se ven desde afuera del sistema. Según Sommerville [Sommerville 2005], estas incluyen:

- Funcionalidad.
- Seguridad.
- Portabilidad.
- Usabilidad.
- Fiabilidad.
- Eficiencia.

Los aspectos intrínsecos, también llamados internos, son los inherentes al diseño del sistema que no pueden ser percibidos en forma directa desde afuera, o sea sin analizar técnicamente al sistema. Estos incluyen, también según Sommerville [Sommerville 2005]:

- Comprensión.
- Verificabilidad¹⁵⁷.
- Adaptabilidad.
- Reutilización.
- Modularidad.
- Robustez.
- Complejidad.

Calidad en un proyecto ágil

En el desarrollo ágil no hay una práctica en particular que se ocupe de la calidad, sino que sus diferentes aspectos se trabajan a través del conjunto de las prácticas esenciales.

A continuación mostraremos las diferentes prácticas que se ocupan de la calidad del producto de software que se produce en un entorno ágil.

Cliente presente

Como vimos en el capítulo “¿Quién manda a quién?”, al tener el cliente presente en un proyecto, el equipo puede tener respuestas oportunas para tomar decisiones. Permite una validación constante de que el proyecto, más allá de estar cumpliendo con las especificaciones, cumple con sus necesidades.

Para esto es necesario que el cliente vele por las necesidades reales y no por el cumplimiento de los compromisos que pudo haber asumido el equipo en determinado momento. Este cliente debe ser decidido a la hora de explicitar cualquier diferencia entre lo contratado y las necesidades que deberá cumplir el software terminado, para que las consecuencias de estos cambios se puedan atender lo más temprano posible. Es su responsabilidad todo lo referente a la priorización de requerimientos y asegurar que lo que se construye aporte el mayor valor posible.

Desarrollo guiado por pruebas

Si bien esta práctica no es exclusiva del desarrollo ágil, es una parte fundamental de este y ha recibido un gran empuje por los

métodos ágiles (sobre todo XP). Como vimos en el capítulo “Probar, probar, probar” esta práctica permite crear software enfocado en cumplir las necesidades que se plantean, y que el diseño se pueda mejorar a través de refactorizaciones a medida que avanza el proyecto.

El desarrollo guiado por pruebas de aceptación extiende este concepto y enfoca el desarrollo en el cumplimiento de las necesidades del proyecto. De esta manera, se evitan desarrollar características no solicitadas, se mejoran las especificaciones y se logran acuerdos sobre la base de un entendimiento compartido.

Integración continua

Esta práctica, que a veces se utiliza también en el marco de procesos tradicionales al igual que el desarrollo guiado por pruebas, ha sido impulsada en forma importante por el agilismo como parte integral de su filosofía, es una parte fundamental de un proyecto ágil. La integración continua, además de garantizar que el producto construido por el equipo funciona y pasa las pruebas, permite a cada programador estar constantemente sincronizado con el desarrollo del resto del equipo.

Esta práctica mejora la calidad del producto en desarrollo, permite al equipo enfocarse en otras tareas y reduce el tiempo de detección de errores y por lo tanto del costo de su corrección. En efecto, no solo automatiza el proceso de integración sino que le brinda un marco a la integración y el despliegue del producto que se desarrolla, ayudando a cada miembro a integrar sus desarrollos en forma ordenada y coherente con las del resto.

Revisión de la iteración

Al final de cada iteración se hace una presentación de lo realizado hasta el momento. Esta práctica provee un hito claro en el cual validar el avance con el cliente. El hecho de realizarse en forma frecuente evita que el trabajo se desvíe de los verdaderos objetivos del proyecto.

En conjunto con el cliente presente en el equipo, la exposición del proyecto en forma frecuente mejora la calidad del producto, al incorporar una instancia de validación fundamental.

Programación de a pares

De alguna manera, la programación de a pares es parecida a una revisión de código constante. Esta práctica ayuda a generar código de calidad y, sobre todo, altamente legible y mantenible. Al fin y al cabo, cada línea de código es vista por al menos dos personas desde su misma creación.

Este es un claro ejemplo de cómo en un proyecto ágil no es necesaria una etapa dedicada a controlar la calidad. En el caso de la calidad del código, es revisada desde el momento en que es escrito, y cada vez que es modificado o revisado, en forma continua y como parte natural del desarrollo.

Barry Boehm, en su paper sobre reducción de defectos de software [Boehm 2001], menciona que un 60% de los defectos del código se pueden eliminar durante las revisiones por pares. La programación de a pares es una forma de hacer revisión continua del código.

Retrospectivas

En las retrospectivas [158](#) se tratan todos los conflictos, problemas y oportunidades de mejora detectados en la iteración. A través de ellas se puede mejorar las prácticas que llevamos adelante y ajustarlas para lograr mejores resultados. Esto permite actuar sobre los distintos elementos que afectan la calidad del producto: la interna se ve fomentada por la discusión entre los perfiles más técnicos, mientras que la externa resulta favorecida por la presencia del cliente.

Al realizarlas luego de cada iteración, tenemos la oportunidad de implementar mejoras y corregir problemas antes de que se arraiguen en el equipo y sea muy difícil o muy tarde cambiarlo. Una retrospectiva de final de proyecto, si bien es útil para la organización, no sirve de nada a la mejora del proyecto en sí.

Control de calidad tradicional en un proyecto ágil

En un proyecto de desarrollo ágil no tenemos una instancia de control de calidad separada. Más bien, durante el desarrollo, el trabajo sobre una user story [159](#) pasa por diferentes estadios durante una iteración hasta que es terminado. Uno de estos es la prueba de aceptación.

Idealmente, el encargado de realizar estas pruebas es el cliente o el responsable del producto. Como, en general, el tiempo del cliente es escaso, en muchos proyectos se recurre a la ayuda de testers. Ahora bien, hemos visto que el trabajo con un equipo de control de calidad externo al equipo no es, en líneas generales, compatible con el desarrollo ágil, al menos en el ciclo normal del proyecto.

En cambio, en nuestra experiencia, en un equipo mediano o grande, tener un miembro con la tarea exclusiva de probar lo desarrollado es una práctica positiva. En este caso, suele haber una persona, a veces denominado *tester embedido en el equipo*, que no solo encuentra defectos, sino que se sienta con los desarrolladores a reproducir y entender el problema. De esta manera, aunque puede ser un tester de oficio, forma parte del equipo como cualquier otro, y no es percibido como alguien externo. Cuando es posible, el tester trabaja junto con el equipo técnico para automatizar las pruebas y hacer la tarea de testing lo más eficiente posible.

Esta práctica dista enormemente del control de calidad habitual en el desarrollo tradicional, en el cual se envía una versión de software a un equipo de testers para que realicen una gran prueba de regresión, y que luego haya una batalla para determinar qué cosa es un defecto y qué no. Habiendo otras herramientas para atrapar errores conocidos, el tester puede dedicar tiempo a tareas como pruebas exploratorias.

El foco de la calidad en el desarrollo ágil

En este libro mostramos que no hay una actividad o práctica ágil referida específicamente a controlar la calidad. En el desarrollo ágil, la calidad es más bien una cualidad que se logra en forma indirecta e implícita. Las diferentes prácticas del desarrollo ágil se enfocan en la calidad desde la creación del software, permitiendo la revisión del producto desde sus diferentes perspectivas y su mejora constante.

De esta forma, el desarrollo ágil ve la calidad como algo que atraviesa todas las actividades y productos que desarrolla. El mantener al mínimo la documentación y elementos que no son el producto que se deberá entregar, centrándose en las tareas que agregan valor al producto, hacen imprescindible tomar un enfoque de calidad en todo lo que se hace, ya que todo tendrá impacto directo sobre lo que entregamos al cliente.

En el clásico contemporáneo *Zen y el arte de mantenimiento de la motocicleta* [Pirsig 1974], Robert Pirsig se expresa sobre la calidad

como la unión entre los mundos objetivo y subjetivo. Propone que la calidad en sí misma no puede definirse, pero para entenderla la divide en estática (lo que puede ser definido) y dinámica (lo que solo puede percibirse), no como una dualidad, sino como diferentes aspectos de algo único.

En el desarrollo de software hay elementos que podemos clasificar como aspectos de calidad estática (calidad del código, cobertura de las pruebas, ajuste a los requerimientos) y aspectos de calidad dinámica (experiencia del usuario, usabilidad, claridad del código). Los primeros son claros para alguien de formación técnica, los segundos son más complicados de definir y suelen percibirse aun antes de medirse.

El enfoque ágil, como hemos visto, se ocupa de calidad en todos sus aspectos, y como parte de cada una de sus prácticas. Trata al software como uno solo, sin diferenciar entre los elementos internos y externos de la calidad, entendiendo que ambos confluyen en un único producto del desarrollo.

En resumen

Es habitual en los proyectos oír que no se tomarán atajos en cuanto a la calidad. También es habitual que eso sea solo una expresión de deseo. En el desarrollo ágil, al estar implícita la calidad en sus diferentes prácticas, resulta más natural desarrollar productos de calidad.

La naturaleza iterativa y tener algo completo¹⁶⁰ al final de cada iteración, facilitan hacer un trabajo de calidad paso a paso en el proyecto.

El cliente presente que toma las decisiones y aprueba los alcances en cada iteración, permite asegurar la calidad extrínseca, permitiendo detectar desvíos respecto a las expectativas en forma temprana para hacer los ajustes necesarios.

Las diferentes prácticas de desarrollo (desarrollo guiado por pruebas, programación de a pares, integración continua) permiten trabajar sobre la calidad intrínseca en forma constante. Y cuando se decide tomar atajos, nos brindan herramientas para poder pagar la deuda técnica acumulada.

De esta manera, si conseguimos que nuestro equipo adopte a conciencia las prácticas mencionadas, la construcción con calidad pasará a formar parte de la rutina de nuestro trabajo.

Un día de desarrollo ágil

A lo largo de los capítulos anteriores hemos presentado la esencia de los métodos ágiles y diversas técnicas y herramientas concretas. En este capítulo uniremos todo lo leído en un relato de lo que sería el desarrollo de un proyecto. Si bien es ficticio, está basado en experiencias vividas por los autores, con lo cual bien podría ser un caso real.

Como hemos mencionado, no hay limitaciones respecto del tamaño de los proyectos que se pueden llevar a cabo utilizando métodos ágiles, pero más allá de eso, hemos decidido utilizar el caso de un proyecto chico pues cuanto más grande el proyecto más complejo sería explicarlo. Además, creemos que el tamaño del proyecto elegido permite explicar de forma clara y suficiente las técnicas aquí presentadas.

Organización del proyecto

Definición del proyecto

Martín Ventor es un emprendedor que ya tiene en su haber varios casos de éxito. Hace un tiempo se le ocurrió un nuevo producto de software y luego de analizar varios proveedores, se inclinó por la recomendación de un amigo y decidió contactar a la empresa Nuevo Software para construir su producto. El primer contacto de Martín fue Pablo, una persona del área comercial, pero con sólida formación técnica. En la primera reunión, Martín no tenía muy en claro el alcance del producto que quería construir, pero sí la fecha en que lo necesitaba. Pablo escuchó atentamente la idea de Martín asegurándose de entender la visión de negocio y el rol que el software a construir jugaba en dicho contexto.

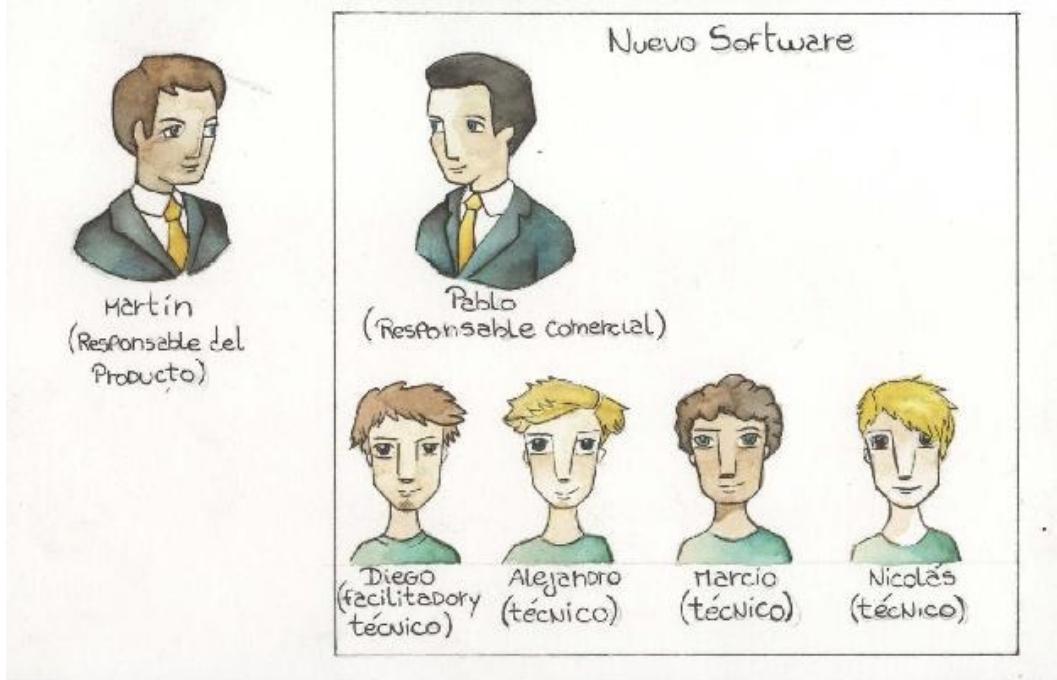
A la semana siguiente, Martín volvió a las oficinas de Nuevo Software para otra reunión, pero en esta ocasión también estaba el equipo de técnicos que en caso de concretarse el proyecto, serían los encargados de la construcción. Dicho equipo estaba formado por Alejandro, Diego, Marcio y Nicolás. La reunión duró unas dos horas y estuvo enfocada en analizar el software a construir. Los elementos de trabajo: notas autoadhesivas de distintos colores, cinta de papel, fichas bibliográficas, una pizarra, varios marcadores y un mazo de cartas de planning poker.

Trabajaron utilizando Visual Story Mapping (VSM), una técnica de análisis totalmente nueva para Martín, pero con la cual todo el equipo estaba familiarizado. Una vez el VSM estuvo completo, se fueron repasando una a una todas las stories identificadas, escribiendo cada una en una ficha bibliográfica, ordenándolas sobre la mesa en base a la importancia establecida por Martín. Mientras se hacía esto, se hablaba brevemente sobre el alcance de cada story y en algunos casos uno de los técnicos agregaba anotaciones al dorso de la story card. Como resultado de la reunión obtuvieron dos artefactos: un Visual Story Map y Product backlog. En este punto se dio por concluida la reunión, Martín y Pablo se retiraron, pero los técnicos se quedaron en la misma sala para realizar una estimación de orden de magnitud. Una vez finalizada, invitaron otra vez a Pablo para comunicarle el resultado de la estimación y para armar una lista de riesgos. Con estos dos artefactos, Pablo armaría la propuesta de proyecto para Martín. El alcance inicial del proyecto estaba dado por 32 user stories, con una complejidad equivalente a 130 story points. El equipo que realizó la estimación sería quien llevaría a cabo el proyecto. Dado que los miembros del equipo ya habían trabajado en conjunto en proyectos de características similares¹⁶¹, acordaron tomar como velocidad inicial: 17 story points / iteración semanal.

Luego de un par de idas y vueltas entre Martín y Pablo, se estableció la fecha de inicio del proyecto para el lunes siguiente. La modalidad de contratación se estableció como Tiempo y Materiales debido a que se trataba de un proyecto innovador con un gran nivel de incertidumbre.

Se consideró una estimación inicial de duración entre diez y doce semanas. El equipo estaría formado por los cuatro técnicos que realizaron la estimación y Diego en el rol de facilitador.

Figura 19.1. Involucrados en el proyecto

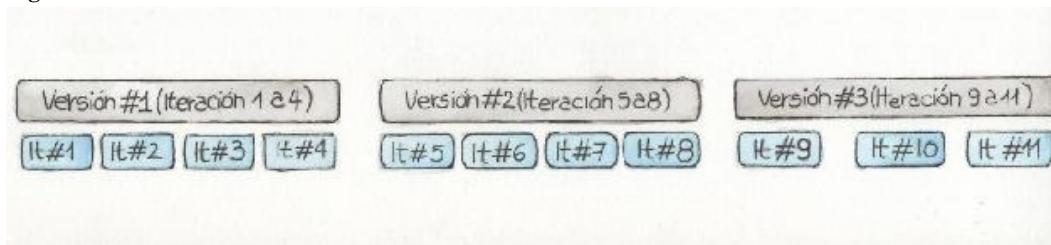


Planificación de alto nivel

Previo al inicio de proyecto, se pactó una reunión de planificación de versiones. En dicha reunión se trabajó sobre el backlog de producto, ya priorizado y estimado, para armar el plan inicial de versiones. Al mismo tiempo, el equipo explicó la dinámica de trabajo e interacción que utilizarían durante el proyecto. Si bien Pablo ya había explicado algo de esto a Martín, Diego se encargó de formalizar ciertas cuestiones respecto de las expectativas de interacción que el equipo necesitaba para poder construir un producto de valor. El equipo trabajaría con iteraciones semanales, se fijó un horario para las reuniones de planificación a realizarse al comienzo de cada iteración (todos los lunes por la mañana) y lo mismo se hizo para las reuniones de revisión, a realizarse al final de cada iteración (todos los viernes por la tarde). Se acordó que las reuniones de planificación durasen a lo sumo de una hora, procurando que fuesen presenciales. La duración de las reuniones de revisión también sería una hora, pudiendo ser en forma remota. Al mismo tiempo, se esperaba que Martín estuviera disponible para consultas en forma frecuente más allá de las reuniones de planificación y revisión. También el equipo acordó hacer una reunión de retrospectiva los viernes por la tarde cada dos semanas.

El plan de versiones quedó conformado con tres versiones. La primera sería al final de la cuarta iteración.

Figura 19.2. Plan de versiones



Ejecución y control

Planificación de iteración

Luego de un mes desde el primer contacto con Martín se comenzó con la construcción del producto. El lunes a las 9.30 de la mañana, todos los miembros del equipo y Martín se encuentran reunidos en una sala de reunión de Nuevo Software. En una pared está pegado el VSM que hicieron tiempo atrás. En otra pared hay una pizarra. Sobre la mesa, un mazo de cartas de planning poker, varios mazos de notas autoadhesivas de diversos colores y marcadores.

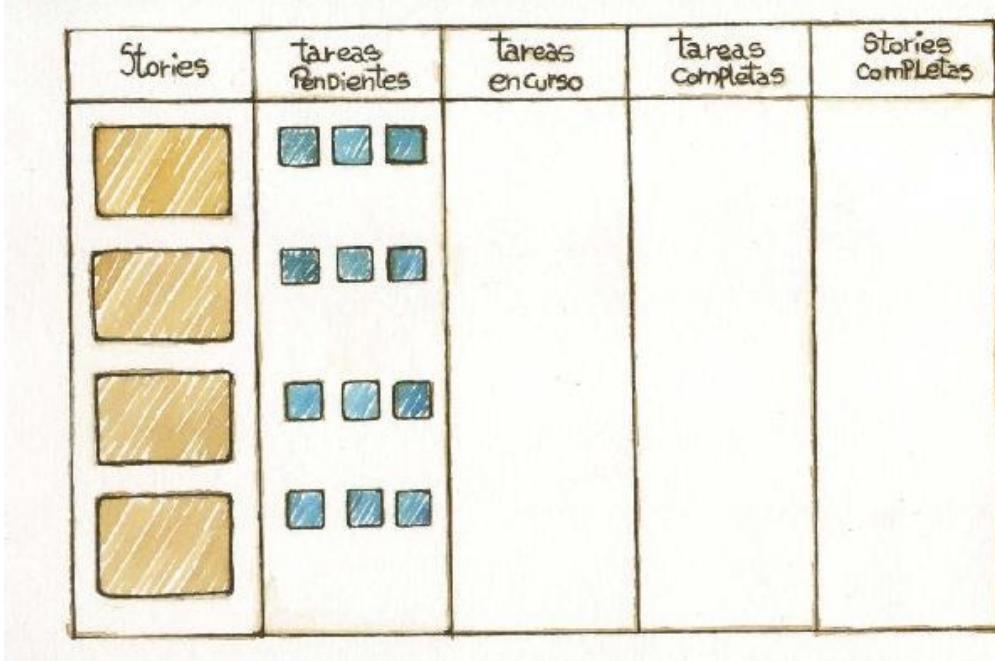
El objetivo de la reunión es planificar el trabajo a realizar durante la iteración. Martín ha seleccionado 5 user stories de las correspondientes a la primera versión, las cuales suman un total de 18 story points. Se van leyendo cada una de las user stories y especificando sus condiciones de aceptación al dorso de cada story card. Al mismo tiempo el equipo va identificando las tareas de ingeniería asociadas a cada user story. Durante este proceso el equipo descubre que una de las user stories es, en realidad, una épica y con el acuerdo de Martín deciden dividirla en 3 user stories. Con esto, las 5 user stories elegidas inicialmente por Martín se convierten en 7, con un peso total de 21 puntos. Dado que el equipo ha asumido una velocidad de 17, Martín debe volver a priorizar las 7 stories pues el equipo no tiene posibilidad de trabajar en todas. Finalmente Martín selecciona 4 user stories por un valor total de 16 puntos. El punto sobrante se utilizaría para algunas tareas de ingeniería de inicio del proyecto (creación del repositorio de código, configuración del servidor de integración continua, etc.). Antes de finalizar la reunión, se actualiza el backlog y el story map agregando las nuevas stories detectadas. El equipo arma el tablero de la iteración, pegando las stories cards que conforman el compromiso de la iteración. También agregan asociado a cada user story una nota autoadhesiva para cada una de las tareas de ingeniería identificadas. Una vez completo el tablero de la iteración, el equipo comenzó a trabajar en forma conjunta en algunas tareas de ingeniería generales del proyecto. Esta vez, a los ya mencionados elementos de trabajo (notas autoadhesivas, pizarra, etc.), se sumó una computadora portátil conectada a un proyector.

Iteración 0

Algunos equipos suelen denominar la primera iteración como iteración 0 y enfocarla en cuestiones de organización/configuración del proyecto como ser: definición de la arquitectura base, configuración de ambientes, creación del repositorio de código y demás herramientas de soporte, etc.

Al cabo de dos horas más de trabajo, el equipo ya había definido algunas cuestiones de diseño generales, había generado la estructura base de la solución de código, creado el repositorio de código y configurado el servidor de integración continua. El siguiente paso fue actualizar el tablero para reflejar el estado de las tareas recién completadas y repartir las que cada miembro continuaría trabajando.

Figura 19.3. Tablero de iteración



El reparto es rápido: no más de 10 minutos y cada miembro vuelve a su puesto de trabajo para dedicarse a la tarea elegida. Así se fue el primer día de la primera iteración del proyecto.

Un día cualquiera

Como de costumbre, a las 9:30 todo el equipo se reúne de pie frente al tablero para comenzar con la reunión diaria. Cada miembro cuenta lo que hizo el día anterior, lo que hará hoy y si tuvo algún impedimento. A continuación, un extracto de la conversación de dicha reunión:

Marcio: ayer completé la story de registro de usuario y hoy pensaba trabajar en la story modificación de perfil. Me gustaría programar en pareja con Ale o Diego, pues vi que requiere unas validaciones en JavaScript y no me doy mucha maña con ello.

Alejandro: no hay problema, podemos trabajar juntos.

Marcio: perfecto, muchas gracias. Alejandro: de nada.

Marcio: bueno, eso es todo, no tengo impedimentos. Diego: ayer trabajé en la story de resumen de transacciones, terminé la lógica de consulta y hoy espero completar la vista. Como impedimento, aún no logré que Martín me diera feedback sobre la story de anulación de transacción.

Nicolás: Martín va a venir esta tarde pues quiere hablar con Pablo sobre otro proyecto, podemos pedirle a Pablo que al terminar la reunión lo traiga para que veamos ese tema. De todas formas pongamos una nota en el tablero para tener presente esta cuestión.

Diego: está bien, al terminar la reunión pongo la nota y hablo con Pablo. No tengo más nada, tu turno, Ale.

Alejandro: bien, ayer trabajé en la story de búsqueda de producto, el caso base está completo; el trabajo de hoy es agregar soporte para algunas de las variaciones y también voy a estar programando en pareja con Marcio para hacer la vista de la story de modificación de perfil.

A medida que cada miembro va hablando, actualiza, de ser necesario, el tablero, moviendo las tarjetas de una columna a otra.

Una vez finalizada la reunión, Alejandro y Marcio se sientan juntos en la máquina de Marcio para programar en pareja. Diego va a hablar con Pablo para pedir que luego de la reunión con Martín le pida que pase a ver al equipo. Por su parte, Nicolás va a su puesto de trabajo para comenzar a trabajar en una nueva story. Comienza por escribir una especificación ejecutable en lenguaje Gherkin. Mientras hace esto, detecta que uno de los criterios de aceptación de la user story es inconsistente. Entonces, se para y agrega una nota en el tablero en el apartado de impedimentos para recordar hablar sobre este tema con Martín esa misma tarde. El resto de los criterios parecen ser razonables, así que completa la especificación Gherkin y a continuación comienza con la implementación de la user story utilizando TDD.

Dos horas después Nicolás tiene diez pruebas corriendo y con ello ha completado la primera tarea de la story que empezó a trabajar; entonces decide que es un buen momento para subir su código al repositorio. Ejecuta el script que corre todas las pruebas y que verifica que su código cumpla con los estándares de codificación. El script falla casi al instante: hay una clase que no cumple con los estándares de codificación. Entonces identifica la clase, arregla las violaciones y vuelve a correr el script. Esta vez la ejecución es exitosa. Nicolás sube sus cambios al repositorio y se dirige a la sala de estar para tomar un café, pero antes de hacerlo decide esperar a que el servidor de integración continua integre sus cambios para poder ir en paz sabiendo que no ha roto nada. Verde, los cambios han sido integrados

correctamente, Nicolás actualiza el tablero y va por su café. Invita a sus compañeros y marchan juntos por un café.

Cierre de iteración

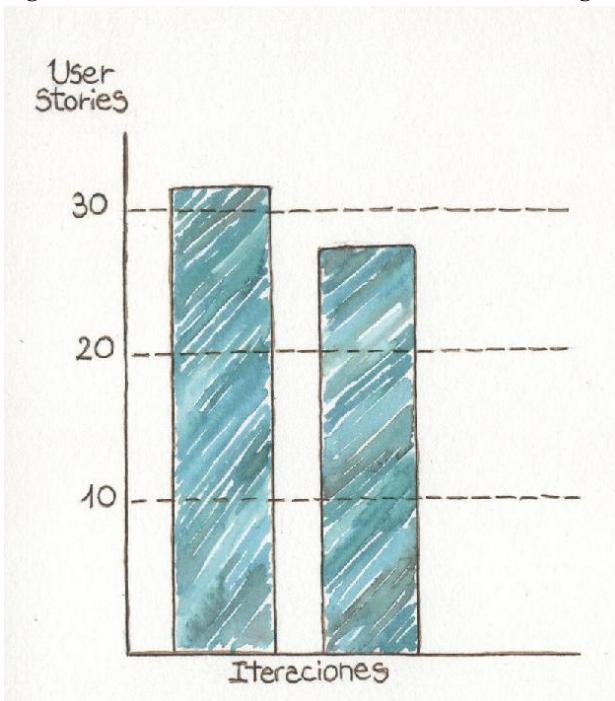
Viernes, día de demo. Como todos los días, a las 9.30, el equipo hace la reunión diaria, en la que cada uno contesta las tres clásicas preguntas. Por la mañana, cada miembro completará las tareas pendientes. Antes del intervalo de almuerzo, se hará el despliegue al ambiente de prueba para la demostración a realizar en la reunión de revisión.

Terminado el almuerzo, a las 14.30, el equipo va directo a la sala de reunión. Diego se encarga de trasladar el tablero de iteración. Como es habitual, allí hay una pizarra, un pack de notas autoadhesivas, un proyector y una computadora. Aún faltan unos 15 minutos para la hora de la reunión, pero el equipo ya está reunido y todo está listo para comenzar en cuanto lleguen el resto de los invitados. Marcio ya repasó la demo y todo ha funcionado correctamente.

Minutos antes de la hora acordada, ya estaban presentes todos los invitados: Martín, el cliente, Germán, su socio, Pablo, el responsable comercial de Nuevo Software que vendió el proyecto y, obviamente, todo el equipo de desarrollo. En esta ocasión, fue Marcio el encargado de facilitar la reunión. En primer lugar, se repasó el compromiso de la iteración, contando el estado actual, y a continuación se hizo una demostración de las funcionalidades desarrolladas. Todo funcionó acorde a lo esperado. El equipo había sido capaz de completar todas las stories comprometidas. Martín hizo algunos comentarios durante la demostración pero todas acotaciones menores principalmente relacionadas a cuestiones estéticas. De todas formas, Diego se encargó de tomar nota de ellas. Si bien la reunión estaba agendada de 30 minutos, la misma se prolongó unos 15 minutos más debido a los comentarios de Martín, que motivaron diversas conversaciones. La reunión de planificación estaba agendada para el lunes siguiente, pero dado que tanto Martín como el equipo tenían tiempo disponible, decidieron hacerla a continuación.

Finalizada la reunión de demostración, se hizo un intervalo para aprovisionarse de café. A continuación Martín y el equipo comenzaron con la planificación de la siguiente iteración. Esta vez, las 5 stories seleccionadas por Martín sumaban 18 puntos y si bien la velocidad en la iteración anterior había sido de 17, el equipo, luego de analizar las stories, se sentía capaz de completarlas, así que asumió compromiso por los 18 puntos. La reunión de planificación terminó alrededor de las 4.30 de la tarde. Martín se retiró y el equipo se quedó actualizando el tablero de iteración, el burndown chart y marcando en el story map aquellas stories que habían sido completadas durante la iteración recién terminada.

Figura 19.4. Burndown chart de stories al cabo de la segunda iteración.



Otro día cualquiera

El equipo estaba terminando la reunión diaria cuando arribaron Pablo y Martín. El motivo de la visita tenía que ver con un pedido de cambio. Resulta que había surgido una oportunidad de hacer una demostración del producto a un potencial inversor. Martín consideraba que para ese inversor particular había una funcionalidad que podría ser muy atractiva. La situación era que no formaba parte del backlog de iteración. Marcio le pidió a Martín que identificara la user story asociada a la funcionalidad requerida en el Story Map. Esa user story estaba estimada en 3 puntos. El equipo se encontraba a mitad de iteración, con 1 story completa, 2 en curso y 1 no iniciada. Justamente la story no iniciada estaba estimada en 4 puntos, con lo cual, en principio parecía que se podía hacer un cambio de stories sin mayor impacto. Pero a pesar de eso, Nicolás planteó una objeción. La "nueva" story había sido estimada solo en la etapa de preventa, cuando el equipo tenía muy poco conocimiento del negocio y por ello Nicolás dudaba que los 4 puntos que tenía asignados fueran algo representativos de su complejidad. "Hagamos un planning poker" propuso Marcio mientras tomaba el mazo de cartas ubicado junto a la pizarra. Todos estuvieron de acuerdo. Tal cual ocurría en las reuniones de planificación, Martín tomó la story card y comenzó a explicar los detalles contestando las preguntas del equipo y anotando al dorso las condiciones de aceptación de la story. Una vez terminada la explicación e identificadas las tareas asociadas a la story, el equipo estimó. Parecía ser que la story era un poco más compleja de lo que había sido estimada inicialmente. Para poder incluirla en el backlog no bastaba con descartar la story de 4 puntos que aún no había sido comenzada. A pesar de esto, Martín consideraba que la "nueva" story era estratégica para contar con el apoyo del potencial inversor. El dilema era interesante: el cliente tenía una necesidad, pero el equipo no podía comprometerse a satisfacerla en la situación actual. Si no se implementaba la "nueva" story, el negocio perdía; si el equipo asumía un compromiso que no era capaz de completar, el equipo perdía. Y dado que todos (el equipo y el negocio) eran parte del mismo barco, ambas situaciones significaban

perder-perder. La solución pasaba por generar un escenario ganar-ganar. Entonces Pablo tomó la palabra:

Pablo: a ver... Martín ¿vos necesitas esa funcionalidad tal cual la describiste para mostrársela al potencial inversor? o ¿será posible mostrar una variante simplificada de dicha funcionalidad?

Martín: buen punto. Definitivamente el inversor no necesita todo lo que describí.

Pablo: bien, entonces dividamos la user story en dos: una que contemple aquellas cuestiones que son imprescindibles para este potencial inversor y otra que contemple el resto de las cuestiones que completan la idea original de la story.

Martín: para mí suena razonable.

Y así se hizo. A partir de la nueva story se generaron 2 stories: una de alta prioridad en la que el equipo trabajaría inmediatamente y otra para ser trabajada a posteriori. Luego el equipo estimó la story de alta prioridad. Esta vez la estimación arrojó un resultado menor, y el equipo pudo asumir el compromiso de completarla en la iteración actual, siempre y cuando dejaran de lado la story de 4 puntos que habían planificado pero que aún no habían iniciado. Martín estuvo de acuerdo con esto. Alejandro entonces actualizó el story map y el tablero de iteración a partir de los cambios acordados.

Cierre de otra iteración

Una iteración más llegó a su fin y con ello una nueva reunión de demostración. El encargado de facilitar la reunión esta vez fue Nicolás. Esta vez las cosas no fueron tan bien, ya que el equipo no logró completar todas las stories comprometidas. A pesar de esto, la reunión no fue tensa, pues Martín ya estaba al tanto de la situación. Siguiendo el viejo dicho de que “las malas noticias es mejor comunicarlas rápido”, el equipo informó a Martín que no llegaría a cumplir el compromiso apenas detectó la situación.

Nicolás decidió comenzar la reunión haciendo la demostración de las funcionalidades completadas, pues prefiere pasar la parte más estresante de la reunión y luego sí, hablar sobre las funcionalidades que quedaron pendientes. La demostración salió tal cual lo esperado, lo cual relajó a todos. A continuación Nicolás explicó a Martín el contratiempo que impidió completar el compromiso. Dado que el equipo siempre trabajó primero las stories de mayor prioridad de negocio, la story que no logró completarse fue la de menor prioridad. Esto se debió a que una de las tareas requeridas para completar la story en cuestión resultó ser mucho más compleja de lo esperado e insumió varias horas de resolución de problemas. El incidente fue resuelto, pero insumió más tiempo de lo esperado. En consecuencia, el equipo simplemente no tuvo tiempo suficiente de completar todas las tareas.

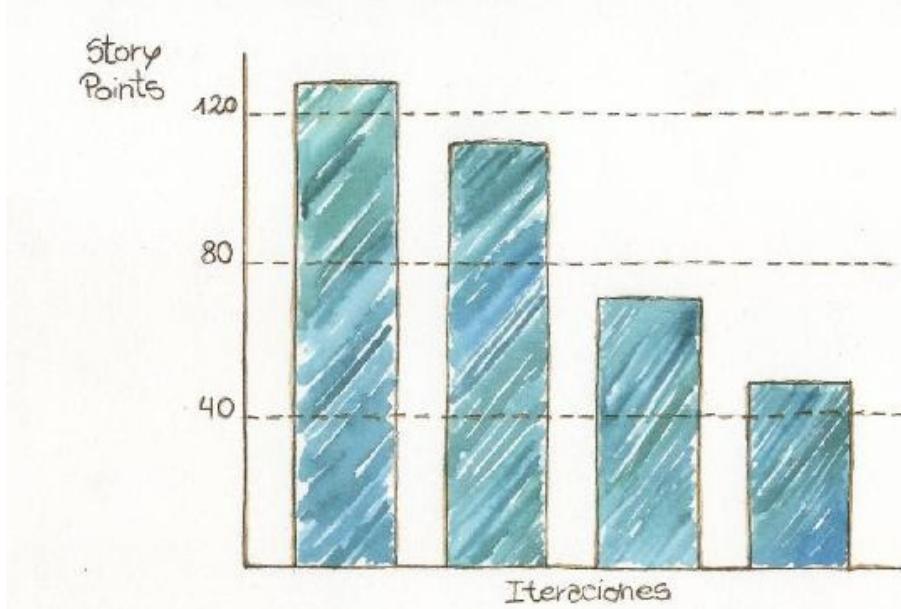
Puesta en marcha

La cuarta iteración tuvo el condimento adicional de la puesta en marcha. La primera versión del producto sería puesta a producción al finalizar la semana. Por eso, el backlog de iteración incluyó algunas tareas relacionadas a la salida a producción.

Si bien para algunos equipos la puesta en producción podría resultar un evento traumático, no era el caso para este equipo, pues:

- El equipo sabe que ha construido el producto que el cliente deseaba, ya que lo ha ido validando de manera periódica con el cliente.
- El producto cuenta con un conjunto completo de pruebas automatizadas.
- El procedimiento de despliegue a producción es análogo al del entorno de pruebas y está automatizado en el servidor de integración continua.

Figura 19.5. Burndown chart de stories al final de la primera versión



En resumen

En la presentación de este caso buscamos evidenciar algunas cuestiones como:

- La estimación inicial del proyecto fue realizada por el mismo equipo de técnicos que luego realizó el proyecto.
- La totalidad del equipo trabajó en este único proyecto a tiempo completo.
- Los miembros del equipo no variaron a lo largo del proyecto.
- No hubo una persona que asignara tareas a otras, sino que cada miembro del equipo elegía en las que quería trabajar.
- En ocasiones hubo dos programadores trabajando en una misma máquina.

- Las pruebas se escribieron antes que el código productivo.
- Hubo un alto grado de automatización de diversas tareas. Somos conscientes de que algunas de estas cuestiones pueden resultar poco comunes o incluso impensables en algunas organizaciones, pero acostumbrarse a estas cuestiones es parte del cambio hacia la adopción de los métodos ágiles.

[156](#) Véase el capítulo “Arquitectura y diseño en emergencia”.

[157](#) En inglés, testability.

[158](#) Véase en detalle en el capítulo “En retrospectiva”.

[159](#) La unidad mínima de requerimiento funcional en un proyecto ágil. Véase el capítulo “Delineando el alcance”.

[160](#) Véase definición de “completo” en el capítulo “Esto está listo”.

[161](#) Por características similares entedemos, proyectos de complejidad similar, con misma tecnología, equipo de mismo tamaño y trabajando con las misma técnicas.

Todo muy lindo pero...

Todas las personas que intentan hacer algo tienen ante sí a tres grupos de individuos: los que intentan hacer lo mismo; los que intentan hacer justo lo contrario; pero, sobre todo, y mucho más numerosos, los que hablan y hablan y no hacen absolutamente nada.

(Visto en internet)

Nos pasa a veces que, cuando terminamos de leer un libro sobre algún nuevo método o manera de hacer las cosas, muchas veces pensamos que son buenas ideas, pero que difícilmente apliquen a nuestra realidad. También es bastante frecuente escuchar a líderes, jefes, gerentes y directores que ante la iniciativa de algún integrante o subordinado de intentar un enfoque ágil, pretenden justificar su negativa explicando que es algo inaplicable con expresiones tales como “esta empresa es muy particular”, “aquí esto no funcionaría” o aclaraciones por el estilo.

En este capítulo intentaremos derribar el mito de la inaplicabilidad dando ejemplos de diferentes situaciones propuestas en los que parece que un enfoque ágil es imposible.

Para empezar dejemos en claro que la cuestión no pasa por ser o no ágil como si fuera algo que debemos lograr a toda costa. Tampoco por cumplir con todas las prácticas mencionadas para tener un sello de certificación de agilidad. Lo importante es ir incorporando las prácticas que generen valor de tal manera que *nos ayude a solucionar los problemas que vayamos teniendo*. Si no podemos llegar a ser completamente ágiles, nadie nos dirá nada, solo que seguramente no aprovecharemos los beneficios de serlo.

Ripasemos entonces algunas situaciones donde podríamos pensar que existe alguna restricción para iniciar nuestro camino en el mundo ágil.

Las excusas

Mi cliente no puede estar todo el tiempo con el equipo

Aunque los autores de los libros donde se explican muchas metodologías ágiles [Beck 1999] insistan en que lo mejor es que el cliente esté en el día a día conviviendo con el equipo, la realidad es que muchas veces esto no es posible, ya sea porque tiene otras tareas o porque se encuentra físicamente alejado. A pesar de esta restricción, muchas empresas han aplicado métodos ágiles de manera exitosa con clientes fuera del lugar del trabajo o incluso con el equipo en un continente y el cliente en el otro. La pregunta que debemos hacernos es cuánto se pierde por la distancia en la relación cliente-equipo y cómo podemos mitigar esas desventajas. Lo que perdemos es mucho porque cuanto más indirecto sea el camino para validar las necesidades y verificar lo construido, más problemas aparecerán durante su recorrido. Si no podemos tener al cliente con nosotros, al menos debemos intentar achicar esas distancias y remover intermediarios innecesarios.

Como primer punto, es fundamental establecer un canal de comunicación directo entre el equipo y el cliente. Es importante que desde ambas partes exista disponibilidad para consultar y responder preguntas en el menor tiempo posible. Los medios orales o audiovisuales son mucho más eficientes que los escritos gracias a la comunicación no verbal. También es bueno que luego de intercambiar ideas por el primer canal, las pasemos en limpio por el segundo. Si tenemos a nuestra contraparte a pocos metros, lo mejor es que las preguntas las hagamos cara a cara. Si nos separan unos cientos de metros o kilómetros, la videoconferencia o la conferencia telefónica es un método efectivo. Los canales escritos son útiles para enviar fórmulas o temas difíciles de expresar con palabras. Si el equipo tiene una duda, debería tener la posibilidad de llamar y aclararla en ese mismo momento. Si hay problemas de disponibilidad, una buena idea es plantear un lapso de tiempo diario para charlar estos temas (más allá de las reuniones diarias donde el cliente a veces participa).

Si no logramos un tiempo mínimo frecuente de atención por parte de nuestro cliente, entonces estamos ante un gran problema. Es posible que el proyecto no tenga demasiada importancia en la organización o que quizás no esté agregando tanto valor al negocio. Habrá que revisar la causa de la falta de tiempo para responder, validar o verificar y reencaminar el proyecto en base a estas respuestas.

A veces, algunos clientes se sienten agobiados (como todos nos sentimos alguna vez por las tareas que tenemos) y puede ser que esté recibiendo tantos llamados y correos que le resulte imposible contestar todo en tiempo y forma, además de cumplir con su trabajo de rutina. En estos casos, una buena idea es crear un rol de intermediario entre el cliente y el equipo para que filtre, unifique y, en algunos casos, hasta responda las preguntas y temas pendientes. Es decir, nuestro cliente sigue siendo completamente responsable del producto que se está construyendo, pero tendrá un ayudante interno que le quitará algo de trabajo. Este representante, que canaliza toda la comunicación es un rol bastante sensible y algo discutido¹⁶². Por eso es necesario utilizarlo solo si aporta a la dinámica de trabajo porque de lo contrario estaremos agregando un eslabón prescindible en la cadena de comunicación. Deberá existir buena química entre este nuevo rol y el cliente para que este sienta que lo que sale de su boca es lo que llega al equipo y viceversa.

Lo fundamental es que nuestro cliente sienta que forma parte del equipo aunque no pueda estar físicamente presente ni tenga disponibilidad completa.

Mi equipo trabaja en diferentes horarios y lugares

Un tema cada vez más frecuente es el de trabajo en equipos distribuidos, lo que conlleva a trabajar en husos horarios muy diferentes. Una opción en estos casos es que se defina, al menos, un rango de horas solapadas al día que permita la interacción de todos los integrantes. Las reuniones diarias mejoran el sentimiento de equipo y la tecnología actual permite una buena simulación de reunión presencial utilizando videoconferencia o teleconferencia. Cuando hay demasiadas diferencias para encontrar un horario común, algunos equipos utilizan una alternativa para las reuniones diarias que denominan maily meeting (derivado de la daily meeting) donde cada integrante escribe a una lista de correo un mail indicando los tres puntos, “que hice ayer”, “que haré hoy” e “impedimentos encontrados”. Este reemplazo de la reunión diaria no tiene la misma dinámica ni asegura que todos hayan leído lo que se envió pero a algunos equipos lo consideran un sustituto suficientemente útil para utilizarlo.

También habría que tratar de encontrar un horario común para las reuniones que involucren al cliente. En estos casos, si hay muchas dificultades para encontrarlo, se puede designar un intermediario que comparta horas tanto con el equipo como con el cliente.

Tengo que presentar un plan de proyecto al comienzo

En muchos proyectos, ya sea porque se encuentra escrito en el contrato, porque lo exige la empresa contratante o porque se lo piden a nuestro cliente, tenemos que generar un plan del proyecto al comienzo de nuestro trabajo. Lo primero que debemos hacer es validar qué es lo que nos están pidiendo ya que muchas veces se pide un plan pero lo que se espera en realidad es un diagrama de Gantt o un

cronograma de entregas.

Si solo tenemos que presentar un cronograma de entregas, es bastante fácil de hacer y podemos tomar el listado de funcionalidades priorizadas y estimadas en el backlog como fuente de entrada y construir los hitos de entregas agrupando las funcionalidades acordando más o menos que se espera en cada una. Organizándolas cada dos o cuatro semanas se puede satisfacer el pedido sin mayor complejidad.

Si lo que nos piden, en cambio, es presentar un plan que incluya temas como la asignación de recursos, la explosión de entregables en actividades, los hitos, la secuencia de actividades indicando precedencias, camino crítico, el plan de costos y tiempos, etc., entonces es algo más complicado y quizás tendremos que crear un plan completo a pesar de estar convencidos de que no es lo mejor.

Una alternativa a la asignación de recursos para todo el proyecto es la de asignar roles en vez de personas concretas (Desarrollador 1, Desarrollador 2, en vez de Nicolás y Diego). Esto nos dará cierta flexibilidad para la autogestión del equipo cuando cada integrante quiera ir tomando las tareas disponibles.

Debemos eliminar dependencias innecesarias entre las tareas. No es necesario terminar el relevamiento para comenzar con los casos de prueba o iniciar el desarrollo, tampoco terminar el desarrollo para comenzar con las pruebas.

Además, en los hitos de entrega, necesitaremos establecer revisiones del proyecto para definir si es necesario ir modificando el plan.

Lo más importante es crear un plan realista y no menos importante es asumir que va a cambiar. A partir de esa premisa podremos generar facilidades y que esos cambios no sean un dolor de cabeza para quien tenga que realizar esas modificaciones.

La experiencia dice que los planes se siguen al pie de la letra cuando hay problemas en el proyecto. Cuando todo va bien, el control externo suele relajarse. Si nos aseguramos un plan inicial que priorice la entrega de valor al nuestro cliente y cumplimos con las primeras entregas (lo que no debería ser un problema, ya que sería la planificación de uno o dos hitos de pocas semanas), aumentará la confianza en el equipo de trabajo y se abrirán puertas para realizar mejoras en el modo de trabajar. Si, por el contrario, se comienza con problemas y retrasos, difícilmente se pueda recuperar ese tiempo perdido y, lo que es peor, cada vez quedará menos margen para intentar promover la agilidad.

Mi contrato es llave en mano [163](#)

A veces no se ve la utilidad de adoptar métodos ágiles cuando se trabaja con contratos llave en mano. ¿Para qué necesito una metodología adaptativa si al final hay que hacer todo lo que dice el contrato? Los contratos llave en mano son muy comunes cuando se compite contra otras empresas en una licitación o cuando el cliente quiere asegurarse no pagar de más en un proyecto. A veces se trabaja con esta modalidad por necesidad y otras veces por costumbre. Si estamos en el segundo de los casos, intentar migrar es más que nada una cuestión cultural y hay que trabajarla como se trabaja un cambio cultural. Si estamos en presencia del primer caso pero tenemos intención de cambiar, entonces tenemos que intentar encontrar las partes “libres” de la metodología de trabajo establecida.

Cuando nos contratan por un trabajo llave en mano, en realidad, nos están dejando trabajar tranquilos a cambio de la creación de un entregable. Si este está bien, se aprueba, se paga y se termina el contrato. Si está mal, no se paga. Si solo fueran estos términos, estaríamos en una posición bastante favorable para aplicar métodos ágiles, puesto que podemos elegir nuestras prácticas y métodos, siempre y cuando cumplamos con lo que se pide. A veces pueden existir hitos de control y facturación intermedios y también exigencias sobre la documentación requerida para controlar el proyecto: en esos casos puede ser que tengamos que hacer algo de trabajo adicional.

El primer consejo para agilizar el trabajo es utilizar una herramienta para definir el alcance donde quede claro qué tamaño inicial tiene cada requerimiento y cuál es el tamaño total del sistema inicial a construir. Tiene que ser fácil de visualizar y de entender por nuestro cliente por qué esta herramienta será clave para poder “jugar” con los cambios cuando lleguen los nuevos requerimientos.

Luego habría que definir un plan de entregas que permita iteraciones cortas. Por ejemplo, agregar hitos de control cada dos semanas

El tercer paso es establecer un mecanismo o proceso de control de cambios muy liviano que permita al cliente ir cambiando los requerimientos que no fueron desarrollados y que, salvo que se quiera modificar el alcance total del proyecto, no requiera un flujo de autorización muy burocrático. En el caso de que ingrese un nuevo requerimiento, el cliente puede incorporarlo siempre que quite un conjunto de requerimientos que sean equivalentes en tamaño. Si necesita sacar uno más grande que el que quiere ingresar, quedará con saldo a favor para el próximo cambio. Solo debería existir una validación por parte del equipo para revisar si el cambio no tiene alguna dependencia fundamental que no pueda ser quitada o para estimar los nuevos requerimientos.

De esta manera, estamos camuflando una planificación adaptativa en un contrato llave en mano. Al principio se hace una estimación macro de toda la funcionalidad y se define un alcance inicial. Con los hitos cortos nos aseguramos trabajar con bastante precisión y foco en la próxima entrega. Con el control de cambios liviano, permitimos hacer y deshacer el producto todavía no construido sin mucha complejidad, ayudando a nuestro cliente a generar valor.

De más está decir que el resto de las prácticas como el uso de pruebas de aceptación, la automatización, la comunicación, integración continua, entre otras cosas, se pueden aplicar independientemente de que el contrato sea llave en mano o no.

El proyecto es muy grande

Si el proyecto es muy grande, más allá de la metodología elegida, lo sugerido es crear subproyectos que pueden o no ejecutarse en paralelo. Todo dependerá de las condiciones del proyecto y de la capacidad del equipo que debe trabajar en él.

Si se divide en subproyectos consecutivos, la solución pasa por determinar el mínimo conjunto de funcionalidades necesarias para construir algo que aporte valor cada vez. Luego, las siguientes versiones incrementarán dicho valor con nuevos conjuntos de funcionalidad agrupados por prioridades.

Si se desea trabajar en paralelo, cada equipo se puede gestionar de manera ágil, aunque es necesaria la creación de algún mecanismo de gestión inter-equipo. Este trabajo también puede ser tratado como un proyecto donde existe un responsable de cada subproyecto y aplicar todas las prácticas ágiles como si fuera un proyecto simple (un responsable del producto, reuniones diarias, entregas cortas, autogestión, integración continua, etc.). Algunas metodologías proponen extensiones para el caso en que haya que trabajar con varios equipos simultáneos. Para el caso concreto de Scrum, existen varios enfoques para escalar el tamaño del proyecto aunque no se detallarán en este libro [164](#).

Mi empresa es muy burocrática / Mi jefe no cree en las metodologías ágiles

Los cambios culturales no siempre empiezan desde la cúpula. Muchas veces, en las grandes organizaciones, las nuevas ideas

proviene de bien abajo. Las estructuras son reacias a los cambios si no están convencidas de que permiten generar mejores resultados. Recién cuando un área comienza a mostrar que alguna nueva técnica incrementa los resultados de algún indicador, es cuando llama la atención y puede tener chances de implementarse de manera corporativa.

Por eso, si trabajamos en una estructura de ese tamaño, o si tenemos un jefe que no cree en las nuevas ideas, lo mejor es intentar con la técnica de guerrilla. Es decir, ir incorporando en diferentes proyectos alguna práctica que al principio no moleste y pase lo suficientemente desapercibida como para evitar el rechazo pero que el resultado de su uso tenga impacto positivo e inmediato.

Por ejemplo, podríamos empezar pidiendo una pizarra para darle visibilidad al proceso. Luego, podríamos generar el hábito de parar a todo el equipo en frente ella para hacer la reunión diaria. Este cambio de comportamiento es muy visible desde afuera. Que los integrantes de otros proyectos y el propio jefe vean que todos los días, el equipo se reúne frente a la pizarra, actualiza el estado de las tareas y algún indicador sencillo, da muestras de un grupo que trabaja junto y que está bien organizado.

Como el ser humano suele comparar su trabajo con el de los demás de manera bastante pesimista (el pasto del vecino siempre se ve más verde), es muy probable que haya integrantes de otros equipos que entiendan que trabajar en esta nueva modalidad es mejor. Es muy probable también que le pregunten a su líder por qué el otro equipo trabaja de esa manera y ellos no, o que directamente los líderes le pregunten al jefe por qué su equipo no tiene una pizarra. Así, “atacando” desde el llano con prácticas de alta exposición visual, se instala un virus de cambio que va dispersándose primero por todo el sector y luego va tomando el resto de la empresa a medida que los resultados acompañan. Una vez iniciada la tendencia, por dinámica de grupo, el individuo que se encuentra fuera de esta querrá formar parte y sumarse a la nueva manera hacer las cosas.

No confío en mi equipo, tengo que controlarlos todo el tiempo

No creer en el equipo parte de una posición bastante complicada para los métodos ágiles, ya que tienen un soporte muy fuerte en la confianza. Estos temas son más complejos que los de seleccionar una práctica o elegir una forma de trabajar. Están más relacionados con entender el origen de esta falta de confianza, si estas causas son reales o aparentes y que se puede hacer para revertir la situación.

Lo que podemos pensar es que si no existe la confianza de un líder hacia el equipo, es muy probable que la situación sea igual a la inversa. Es más, seguramente el término líder no sea correcto, ya que uno verdadero debería ser capaz de influir en el equipo para que trabaje motivado hasta cumplir con el objetivo. Si no existe la confianza en el líder, lo demás desaparece: el equipo oculta información, no es sincero, tiene comportamiento reactivo, etc. Así comienzan problemas con entregas que parecían completas pero no lo estaban, estimaciones sobreestimadas (o generadas sin compromiso), o errores que aparecen recién cuando el cliente los reporta.

Trabajar sin confianza es un dolor de cabeza para todos y revertir esta situación es responsabilidad de quien está a cargo del grupo.

El jefe o *pseudolíder* debe intentar identificar las causas por las cuales desconfía del equipo o de alguno de sus integrantes¹⁶⁵. Luego será necesario intentar descifrar si estos motivos ocurren realmente o son producto del estereotipo que creamos de esta persona. Para lograr esto, es necesario ratificar nuestras creencias con hechos. Si nos parece que llega tarde, tomar la asistencia durante un par de semanas; si creemos que entrega código de baja calidad, llevar un registro de los errores; y así con cada problema.

El siguiente paso es reunirse con quien sea dueño del problema y hablar sobre la situación. Conviene exponer los hechos sin animosidad e intentar generar un ambiente donde la persona pueda entender el porqué del problema de confianza, pero también pueda defenderse adecuadamente. Una vez aclarados los puntos en discordia, expresar con claridad las expectativas sobre el trabajo a realizar.

Muchas de estas reuniones terminan con sorpresas, porque para cada hecho puede haber una causa no analizada.

Por ejemplo, alguien puede estar llegando tarde porque tiene al padre enfermo y no se animó a contarlo. Otro puede estar entregando con errores porque la capacitación prometida por la que se postuló al proyecto finalmente nunca se efectivizó. También suelen darse casos donde los problemas de motivación aparecen porque siempre se le asignan las tareas más monótonas y la persona esperaba algo con más responsabilidad para sentir que crece profesionalmente en su trabajo.

Surgen, entonces, círculos viciosos donde la persona se desmotiva, baja la productividad, el jefe le asigna tareas más fáciles por la baja confianza y todo vuelve a comenzar.

El resultado de estas charlas es muy fructífero y genera nuevos lazos de confianza que permiten que la información vuelva a fluir y que el equipo se sienta motivado.

Siempre existen excepciones y empleados que conviene evitar por su dificultad para asumir responsabilidades, así como hay jefes que ocultan la verdad o tienen intereses mezquinos, pero la mayoría de la gente puede trabajar muy bien con la motivación adecuada.

Para fomentar la confianza del equipo, podemos comenzar dejando que cada integrante se asigne las tareas delante del resto del equipo. Es el primer paso hacia la autogestión y es bastante sencillo de iniciar sin consecuencias graves. En cada reunión diaria, cada integrante que termina con una tarea continuará seleccionando el próximo trabajo del listado que se encuentra en la pizarra. Si no sabe, otro integrante puede explicar alguna de las tareas disponibles para ayudarlo a tomar una decisión. Elegir el trabajo crea la sensación de ser dueño y de trabajar hasta dejarlo lo mejor posible.

La otra actividad que hay que fomentar es la comunicación en el equipo. Debemos evitar la comunicación direccional integrante-líder y fomentar el canal desde cada integrante hacia el resto del equipo. Cada novedad, cambio en un requerimiento, problema, solución, impedimento, retraso, llegada tarde, ausencia, modificación de la fecha de entrega, salida a producción, cambio de ambiente, etc., debería ser comunicado a todo el equipo por cualquiera de los integrantes que cuente con esa información. Conocer los pormenores del proyecto, entender la situación de algún compañero, estar al tanto de cambios, tener responsabilidad en la elección de las tareas y en las estimaciones, trabajar con un líder o jefe sincero, aumentan el sentimiento de equipo, motivan a los integrantes y mejoran la productividad de su trabajo. La confianza en quienes trabajan con nosotros genera un sentimiento recíproco de quienes reciben ese beneficio.

Hay partes del software / negocio que solo conoce

una persona

Por cuestiones de eficiencia o solo porque el tiempo fue determinando que haya gente experta en una parte de nuestros sistemas, podemos llegar a tener situaciones en donde existen ciertas actividades que solo pueden ser cubiertas por una persona identificada con nombre y apellido.

Como todos sabemos, esto tiene varios problemas. Por ejemplo:

- Gran dependencia de la finalización de la tarea a la disponibilidad o humor de esta persona.

- Imposibilidad de asignarle tareas diferentes porque siempre tiene que estar disponible para cuestiones relacionadas con su experiencia.
- Imposibilidad de asignar tareas específicas a otros integrantes porque desconocen como está hecho el trabajo.
- Riesgo de alto impacto si la persona abandona la empresa o el proyecto.

Aunque las metodologías ágiles ponderan a las personas por sobre los procesos, tener este tipo de asignaciones genera bastantes problemas para la organización del trabajo del grupo. Para fomentar la autogestión y la responsabilidad y conocimiento compartido, debemos intentar evitar la existencia de *héroes*.

Algunas prácticas que mejoran notablemente este problema son las demostraciones de los productos al final de cada iteración donde se muestra todo lo construido. De esta manera, todos los integrantes y el cliente estarán al tanto del trabajo realizado. La otra práctica que puede ser útil es la de las inspecciones de código, donde el responsable de una sección de la aplicación explica, a través del código, cómo fue implementada la solución desarrollada. La práctica de programación de a pares de XP también ayuda a la transferencia de conocimiento.

Otro consejo sobre este punto es trabajar sobre la elección de objetivos claros y concisos en las iteraciones, de forma tal que todo el equipo esté trabajando hacia un objetivo común y que tenga más sentido la comunicación sobre el trabajo entre los integrantes.

En resumen

Este capítulo es un pequeño listado de problemas que podemos encontrar quienes intentamos generar cambios y proponer mejoras desde un enfoque ágil. Las implementaciones ideales no existen y cada empresa, organización o equipo adapta las prácticas de manera que le resulten útiles para trabajar mejor. Siempre se pueden crear pretextos para evitar comenzar a experimentar. Las oportunidades para intentar un cambio aparecen todo el tiempo. Queda en cada uno decidir si la situación amerita probar alguna práctica nueva o si aún no es momento de hacerlo. Los inconvenientes y posibles desafíos son muchos y muy diversos, pero cada vez que encontramos algo que nos impide avanzar tenemos que preguntarnos si lo que hay enfrente es una restricción o un obstáculo a ser removido.

[162](#) Véase [Schwaber 2011].

[163](#) Ya hablamos algo de este tema en el capítulo “Formalizando compromisos” en la sección Contratos de precio fijo.

[164](#) El interesado puede consultar, *Scaling Scrum* o la sección correspondiente de *Scrum and XP from de Trenches*. *Scaling Scrum* está disponible en [Bird 2007]. Para Scrum and XP from the trenches ver [Kniberg 2007].

[165](#) Ejemplo: no parece comprometido, siempre entrega con errores, llega y se va a la hora que quiere, falta en los momentos cruciales, echa culpas a sus compañeros, no termina a tiempo.

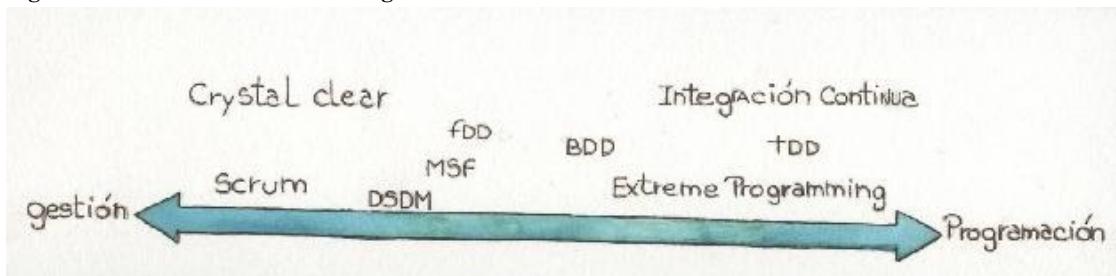
La riqueza de la diversidad

A lo largo de todo el libro hemos hablado de las generalidades de los métodos ágiles. En este apéndice daremos una mirada rápida a algunos métodos ágiles en concreto.

El espectro de métodos ágiles

Dentro de lo que habitualmente se denominan métodos ágiles, se encuentra un variado espectro de técnicas, prácticas y metodologías. En forma genérica nos referimos a ellos como métodos. En un extremo de este espectro de métodos se encuentran los relacionados a cuestiones de gestión mientras que en el otro se encuentran aquellos más relacionados con cuestiones cercanas a la programación y el código fuente. Ocurre al mismo tiempo que gran parte de los denominados métodos de gestión son aplicables en diversos entornos mucho más allá del software, cosa que no ocurre con los del otro extremo.

Figura A1.1. Universo de métodos ágiles.



Los métodos y el manifiesto

Dependiendo de la forma en que uno se acerque a los métodos ágiles, puede que comience por algún método “completo” o bien por alguna práctica en particular, pero generalmente en algún momento, más tarde o más temprano uno llega a preguntarse: ¿qué define un método como ágil? Uno quisiera tener un algoritmo para que, dado un método de desarrollo, poder obtener una respuesta del tipo “Sí, es ágil” o “No es ágil”. Desconocemos si dicho algoritmo existe, pero sin duda debería, en última instancia, validar el método de desarrollo en cuestión, a la luz del manifiesto ágil, pues es la única fuente formal universalmente aceptada sobre el agilismo. Entonces, podemos decir que un método es ágil si está alineado con el manifiesto. Si queremos bajar esto a algo más concreto podríamos decir que todo método ágil en primer lugar no debería contradecir al manifiesto y luego debería estar alineado con los siguientes puntos:

- Incorporar feedback del cliente en forma frecuente.
- Tomar el valor de negocio como principal guía del proyecto.
- No olvidar que estamos tratando con personas.
- Ser capaz de adaptarse fácilmente a los cambios.
- Buscar la mejora continua.

Agilidad y disciplina

Si uno se pusiera a contar la cantidad de “reglas” definidas por los distintos métodos ágiles, notaría que en comparación con otros, son muchas menos. Esto podría llevar a pensar que son “más fáciles de aplicar” pero es erróneo. Veamos por qué. Si uno utiliza un método A que define cien reglas, pero por alguna razón, se ignoran cinco, se puede decir que se cumple con el 95% del método.

En otro escenario, supongamos que un método B define diez reglas, si uno “ignora” cinco, entonces está cumpliendo con el método solo en un 50%, y decrecen considerablemente las probabilidades de éxito de lo que se está haciendo, ya que las reglas pertenecientes a un sistema guardan cierta relación entre sí.

La combinación más común: Scrum + XP

Como ya mencionamos, el espectro de métodos ágiles es muy amplio, pero todos comparten la misma base: el manifiesto. Es por ello que resulta común trabajar con más de uno a la vez, combinando generalmente uno de gestión con uno o más técnicos. En este sentido, una de las fórmulas más populares es usar Scrum como método de gestión combinado con las prácticas de XP. Veamos entonces una pequeña reseña.

Scrum

Si bien sus orígenes se remontan a la década de 1980, su aplicación en el desarrollo de software fue formalizada en 1995 cuando Jeff Sutherland y Ken Schwaber publicaron un trabajo titulado “*Scrum methodology*” en la conferencia OOPSLA [Sutherland 95].

Siendo estrictos debemos decir que Scrum es un marco de trabajo que propone un proceso, un conjunto de prácticas y roles, dejando a criterio del lector si es suficiente para denominarlo metodología.

Una particularidad de Scrum es que es un método de gestión de aplicación general, lo cual permite que sea utilizado más allá del desarrollo de software.

El flujo de trabajo

Scrum propone un esquema de trabajo iterativo incremental dado por iteraciones denominadas Sprints. Un Sprint es una iteración de tiempo fijo (time-boxed) de máximo un mes de duración, durante la que el equipo genera un incremento de producto en condiciones de ser liberado. Al mismo tiempo, Scrum propone un control de proceso empírico que se basa en la idea de que el conocimiento procede de la experiencia y que es en base a ella que deben tomarse las decisiones. Llevar esto a la práctica implica inspeccionar y adaptar en forma continua a lo largo de todo el proyecto, para lo cual Scrum define los siguientes cuatro eventos dentro de cada Sprint:

- *Reunión de planificación*: se realiza al comienzo de cada Sprint. En ella colaboran el product owner y los miembros del equipo planificando el Sprint y armando el Sprint Backlog.
- *Reunión diaria*: se realiza todos los días y su duración está restringida a 15 minutos. En ella cada miembro del equipo comenta su avance desde la anterior reunión, que planea hacer hasta la siguiente y si tuvo algún impedimento.
- *Reunión de revisión*: se realiza al final de cada Sprint con el objetivo de que los miembros del equipo presenten el incremento del

producto al product owner para su aprobación formal. También pueden participar otros interesados del proyecto y trabajando todos colaborativamente se adapta el product backlog a partir de lo detectado en el producto actual.

• *Reunión de retrospectiva*: se realiza a continuación de la reunión de revisión, pero solo participan los miembros del equipo. Es un espacio para que el equipo identifique oportunidades de mejora, inspeccionando las relaciones interpersonales, las herramientas y el proceso.

Roles

Scrum define tres roles: Miembro de Equipo, Product Owner y Scrum Master.

Los *miembros de equipo* son los técnicos a cargo de la construcción del producto. Aquí debemos destacar que Scrum no hace ninguna mención explícita de roles dentro del equipo, sino que solo habla de miembros de equipo a secas. Se espera que sea autoorganizado y multidisciplinario. Dado que es el equipo el que construye el producto, son sus miembros quienes lo deben estimar. El equipo trabajará implementando el producto en base a las definiciones y prioridades provistas por el Product Owner.

El *Product Owner* es literalmente el dueño del producto y como tal, responsable para definir sus funcionalidades y priorizarlas. Durante el desarrollo de una funcionalidad debe colaborar con el equipo para despejar dudas y brindar feedback. Además, es el responsable de la aceptación de las funcionalidades una vez terminadas.

Por último, tenemos al *Scrum Master* que es el responsable de que el equipo aplique Scrum. Muchas veces se lo suele confundir con una especie de jefe o con el clásico líder de proyecto, lo cual es incorrecto. Es un facilitador, una especie de guía espiritual que vela porque el equipo aplique las prácticas de Scrum.

Puede que en un determinado contexto la definición de las características del producto dependan de varias personas, en ese caso, quien ocupe el rol de Product Owner deberá lidiar con todos ellos con el fin de que el equipo pueda confiar plenamente en la palabra del Product Owner.

Los artefactos

El artefacto más importante de Scrum es el denominado *Product Backlog*, en ocasiones traducido como pila de producto. Es básicamente una lista de funcionalidades a construir para completar el producto. Cada uno de los ítems de este backlog es justamente eso, un ítem de backlog: Scrum no entra en mayor detalle, no dice como se especifican esos ítems.

Cuando se utiliza Scrum en conjunto con XP, cada Product Backlog ítem es una User Story.

Además del Product Backlog existe un *Spring Backlog*, que contiene el subconjunto de Product Backlog ítems seleccionado para ser trabajado durante el Sprint actual junto con las tareas que el equipo ha identificado que deben llevar a cabo para completar los ítems del backlog.

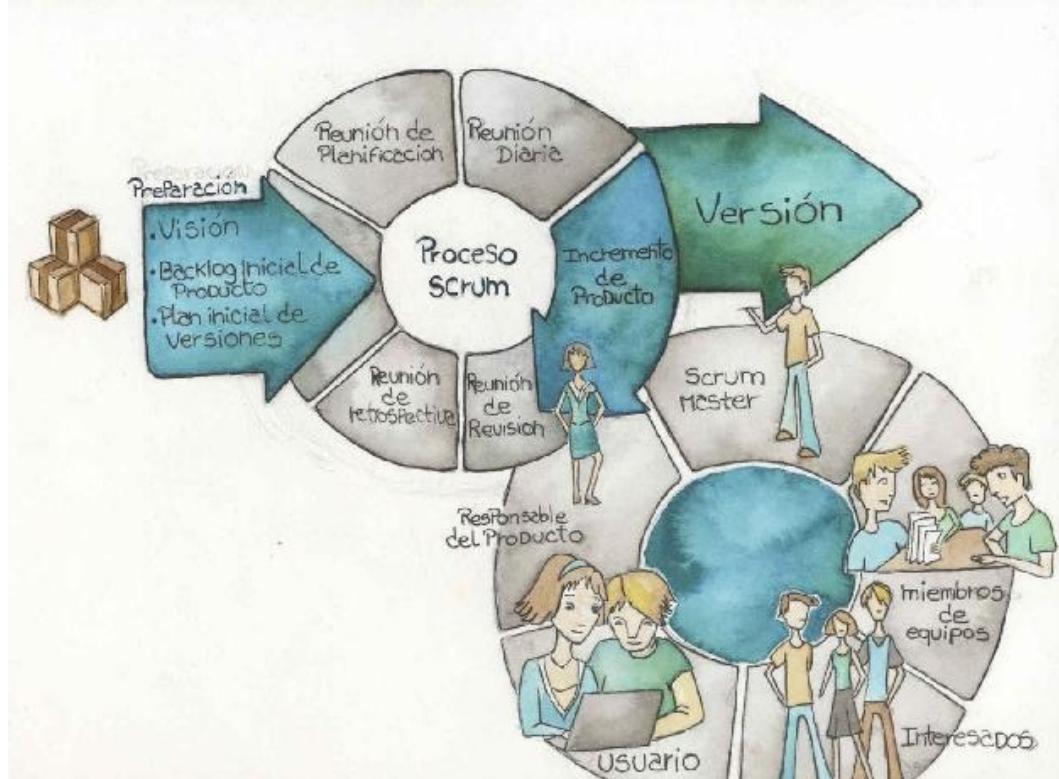
Por último, al finalizar cada Sprint, se obtiene un *incremento de producto* dado por el conjunto de funcionalidades completadas durante el Sprint en cuestión. Las funcionalidades incluidas en el incremento de producto deben estar en condiciones de ser liberadas para ser utilizadas por los usuarios.

Scrum en una imagen

La siguiente figura muestra los eventos, los roles y los artefactos de Scrum.

Para aquellos interesados en profundizar en Scrum la primera recomendación es el libro escrito por Ken Schwaber y Mike Beedle, *Agile software Development with Scrum* [Schwaber 2001].

Figura A1.2. Scrum en una imagen.



Extreme Programming¹⁶⁶ (XP)

Es una metodología trabajo para la construcción de software. Sus primeras aplicaciones datan de mediados de la década de 1990 pero

el primer libro que la formalizó fue el de Kent Beck en 1999 [Beck 1999].

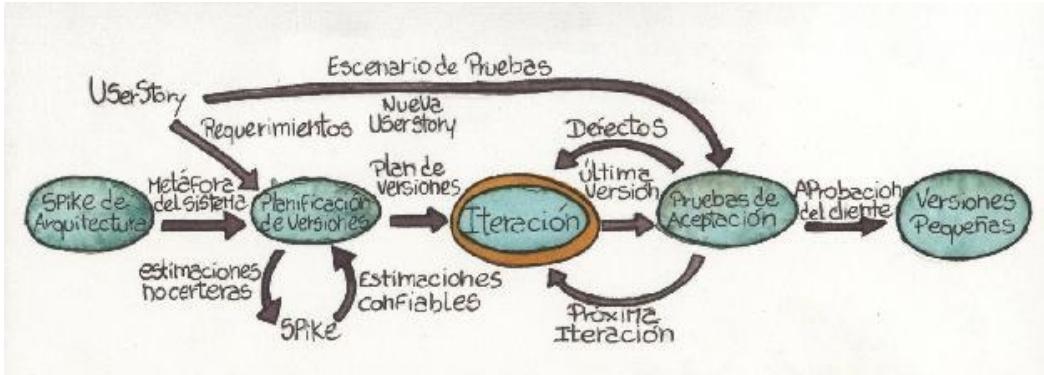
Tal vez algo curioso de esta metodología es que como parte de su definición hay un hincapié muy importante en cuatro valores:

- Simplicidad
 - Comunicación
 - Respeto
 - Coraje

El flujo de trabajo

La siguiente figura muestra el flujo de trabajo propuesto por XP junto con algunas de las prácticas más comunes.

Figura A1.3. Flujo de trabajo de XP.



Como puede observarse el flujo de XP es compatible con el propuesto por Scrum, ya que el flujo propuesto por este es lo suficientemente genérico para permitir la inclusión de las prácticas propuestas por XP. Es así que mientras que Scrum habla de Sprints, XP se refiere a iteraciones. Mientras que Scrum habla en modo genérico de Backlog ítems, XP habla de user stories.

Roles

Existen cuatro roles principales. El primero es el *Customer*, el cliente, básicamente una persona con conocimiento del negocio que sabe que es lo que el sistema en desarrollo debe hacer. Se espera que la persona que ocupe este rol deje de hacer su trabajo cotidiano para ser parte del equipo de desarrollo. Tendrá su espacio de trabajo junto al equipo y cada día se dedicará a hacer pruebas de aceptación (customer tests). Es por esto que a este rol se lo suele denominar on-site customer.

El siguiente rol es el *Programador*, que es quien construye el producto, lo cual implica bastante más que programar. En la concepción de XP, el programador hace todas las tareas necesarias para programar y lograr la aprobación de la funcionalidad construida. Junto con el customer son los dos roles indispensables en todo proyecto.

El *Tester* es un rol también presente en otras metodologías, pero en este caso tiene una concepción distinta. Se espera que trabaje en conjunto con el *Customer* creando pruebas funcionales, implementando el código necesario para automatizar dichas pruebas y ejecutándolas una vez listas. Es importante destacar que en XP hay un equipo, que incluye tanto a programadores como testers y que cuando un ítem se considera completo es porque ha pasado por el tester y la aceptación del customer (esto está en línea también con el enfoque de Serum, aunque es un poco más explícito).

El *Coach* ayuda al equipo para hacer XP, guía con el ejemplo y es su mentor. Podría decirse que su presencia es obligatoria para todo equipo que comience a trabajar con XP, pero al mismo tiempo un buen coach es el que logra que el equipo adopte exitosamente XP haciendo que la presencia del coach ya no resulte necesaria. Es por esto que, en alguna bibliografía, el coach es considerado un rol complementario.

Adicionalmente, existen algunos otros roles complementarios que pueden o no estar presentes en un equipo XP: el *tracker* y el *gran jefe* (*the big boss*). El tracker se encarga de llevar el calendario del proyecto y ciertas métricas como la velocidad del equipo. El gran jefe es un rol gerencial que no forma parte del equipo en el día a día. Es quien debe encargarse de que el equipo cuente con los recursos necesarios y quien se encarga de las cuestiones contractuales/legales.

Las distintas publicaciones sobre XP a lo largo del tiempo, comenzando por el primer libro publicado por Beck, han ido variando la definición de los roles complementarios, incluso haciendo mención en algunos casos a otros roles no mencionados aquí como XPManger y Consultor.

Las prácticas

En el corazón de XP hay doce prácticas de desarrollo, casi todas ellas de índole técnica. Algunas de ellas han tenido repercusión mucho más allá de XP y por eso han sido tratadas en capítulos anteriores.

1. *El juego de planificación* (the planning game), los expertos del negocio y los técnicos del equipo de desarrollo planifican el trabajo. La gente de negocio presenta las necesidades, los técnicos preguntan para poder estimar. Los expertos de negocio priorizan las necesidades y finalmente asumen un compromiso de entrega para determinada fecha.

2. Versiones pequeñas, al realizar la planificación, trabajar siempre en pequeñas entregas de manera de poder entregar valor en forma frecuente, típicamente cada dos semanas.

³ Metáfora del sistema, establecer una visión de alto nivel del sistema para guiar la construcción y facilitar su diseño.

4. **Diseño simple**, elegir siempre la opción más simple que pueda funcionar. Como es inevitable que los requerimientos cambien, no tiene sentido invertir en un diseño complejo que luego deberá ser cambiado.

5. **Pru^aba continua** (continuous testing), el sistema se prueba de forma constante a medida que se va desarrollando, generando pruebas automatizadas. Los programadores son responsables de la **prueba unitaria**, la cual suele escribirse incluso antes de escribir el código debido al uso de la técnica de diseño basado en pruebas (TDD). También se automatizan las **pruebas de aceptación**, las cuales

son escritas por el customer con apoyo del tester. Cuando las pruebas de aceptación pasan, se considera que la correspondiente user story ha sido completada.

6. *Refactoring*, se utiliza esta técnica para permitir la continua evolución del código para facilitar su posterior mantenimiento.

7. *Programación de a pares*, el código es escrito por dos programadores trabajando en una misma máquina, pues dos cabezas piensan mejor que una, aumentando la calidad del código y disminuyendo el porcentaje de errores en la aplicación [167](#).

8. *Propiedad colectiva del código* (collective code ownership), todo el código es de todos los miembros del equipo y por eso todos tienen el poder de modificarlo. No hay nadie que sea “propietario” de alguna porción específica.

9. *Integración continua*, todo el código del sistema es integrado, al menos una vez al día, procurando estar listos para liberarlo en cualquier momento.

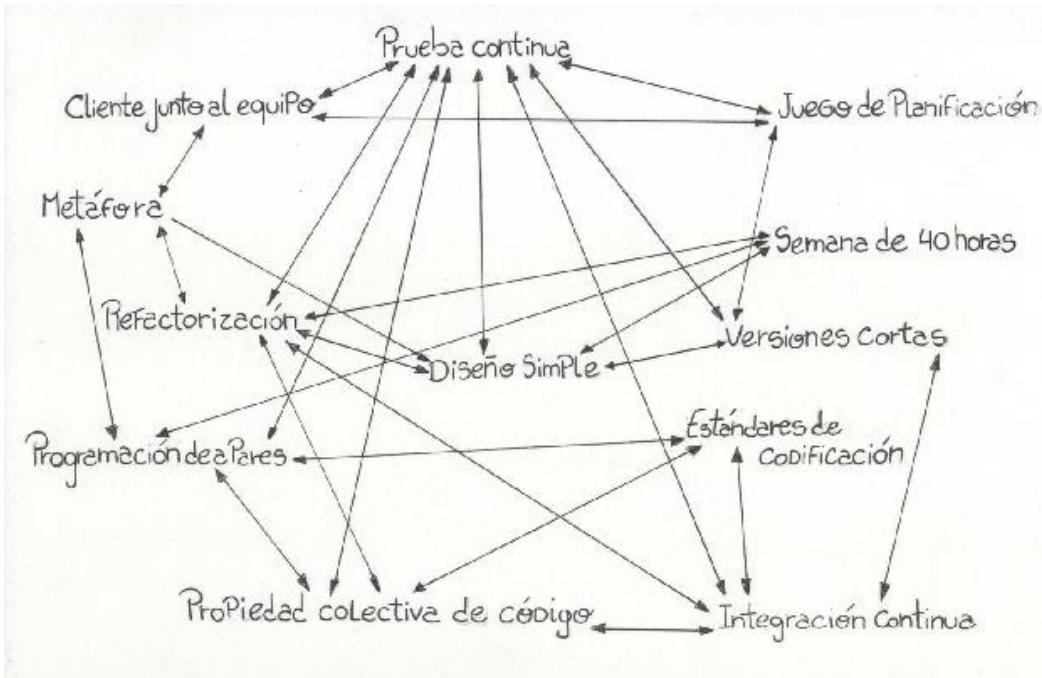
10. *Semana laboral de 40 horas*, el equipo debe trabajar de forma tal de poder asegurar un ritmo sostenible, ya que no hay forma que un equipo trabaje en forma continua 60 horas semanales. Puede que un equipo trabaje, 35, 40, 45 horas semanales, pero generalmente se habla de 40 horas en forma genérica haciendo referencia a un ritmo sostenible razonable.

11. *On-site customer*, el customer tiene que estar todo el tiempo disponible para evacuar las dudas de los técnicos, por ello deja su puesto de trabajo habitual para mudarse con el equipo de técnicos.

12. *Estándares de codificación*, todo el código es escrito siguiendo ciertas normas definidas por el propio equipo, lo cual facilita la comunicación y permite que todos se sientan cómodos modificando cualquier parte del sistema.

Estas prácticas, si bien son relativamente pocas, están íntimamente relacionadas y es imprescindible que se apliquen todas en forma conjunta para así aumentar la probabilidad de éxito utilizando este método. La siguiente figura muestra como cada práctica influyen en otras.

Figura A1.4. Relación entre las prácticas de XP.



Otros métodos no tan difundidos

Como ya mencionamos, más allá de Scrum y XP existen otros métodos, tal vez menos populares, pero no por ello menos ágiles. A continuación enumeraremos algunos alternativos mencionando algunas de sus características distintivas.

DSDM

El Dynamic System Development Model es un método desarrollado en el Reino Unido a mediados de la década de 1990. En sus comienzos, reunió varias de las prácticas propuestas por el enfoque RAD. A largo de los años ha evolucionado incorporando varias lecciones aprendidas y generando nuevas versiones. La versión actual es conocida como Atern.

Una particularidad es que su foco es el desarrollo de sistemas de información con restricciones de calendario y presupuesto, permitiendo que el alcance sea variable.

Otra particularidad interesante es que define explícitamente ciertos requisitos para su uso. Entre ellos se destaca aceptar que el principal factor de fracaso en los proyectos es de índole humana.

Al mismo tiempo toma como fundamentales los siguientes principios:

- Foco en las necesidades de negocio.
- Entrega a tiempo.
- Colaboración.
- Nunca comprometer la calidad.
- Construir sobre bases firmes.
- Desarrollo iterativo.
- Comunicar continua y claramente.
- Demostrar control.

DSDM está estructurado en siete fases que cubren desde la concepción de proyecto (pre proyecto) hasta su finalización (post proyecto) y donde distintos esquemas iterativos pueden plantearse combinando dichas fases.

En cuanto a los roles, los mismos se presentan en dos grupos: Roles de Proyecto y Roles de desarrollo.

La fuente oficial de información de DSDM es el sitio del Consorcio DSDM: <http://www.dsdm.org/>

Figura A1.5. Ejemplo de posibles estructuras iterativas de fases.

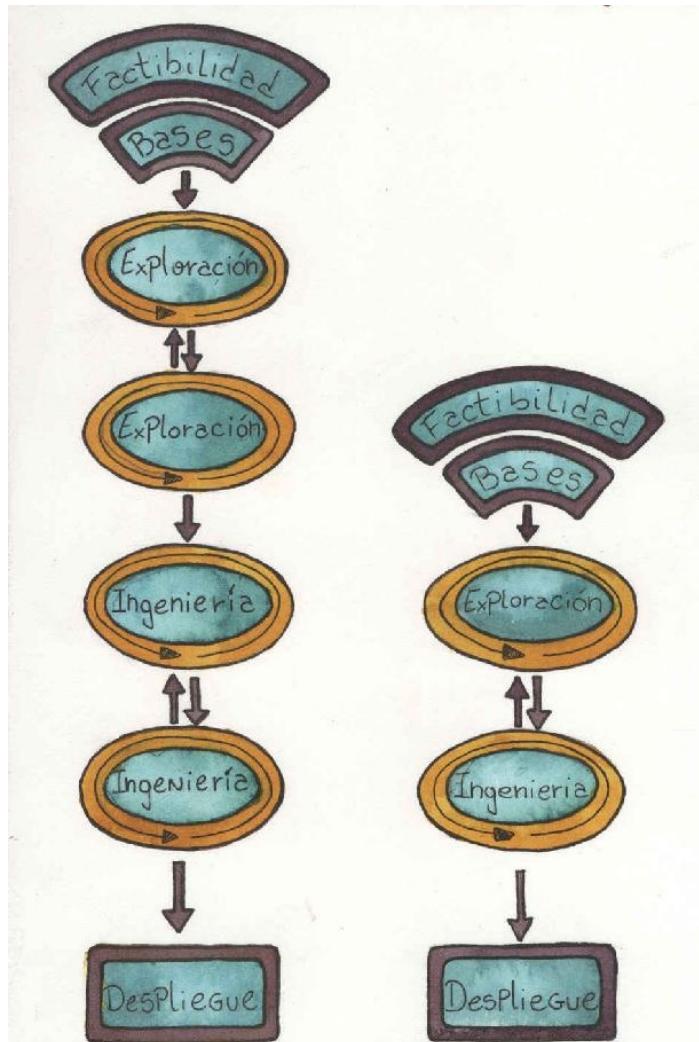
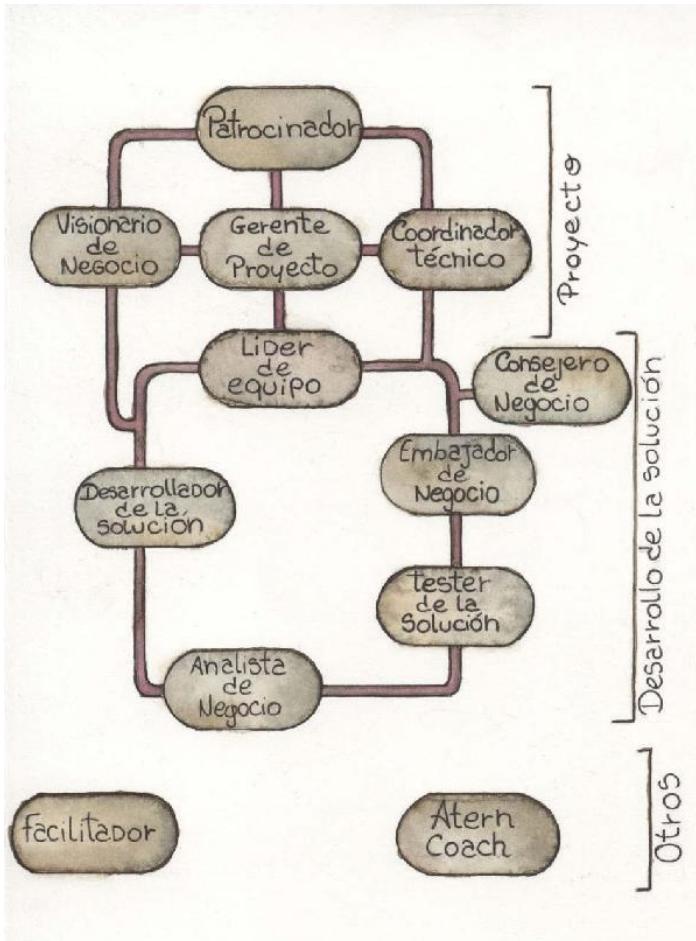


Figura A1.6. Roles de DSDM/ Atern.



Feature Driven Development

Este método fue propuesto por Jeff De Luca en 1997 para lidar con un proyecto de cincuenta personas que debía ser completado en quince meses. Este punto inicial de por si solo ya marca una diferencia con varios de los otros métodos ágiles que generalmente proponen equipos más pequeños. El hecho de trabajar con equipos grandes tiene varias implicancias que llevan a diferencias radicales en la forma en que se enfocan ciertas cuestiones. Para aquellos interesados en ahondar en este método, el punto de entrada es el sitio es su website www.featuredrivendevolution.com.

Crystal Clear

Crystal Clear un método ágil desarrollado por Alistar Cockburn y pertenece a la familia de métodos Crystal. Está enfocado en las personas y en prácticas de gestión. Se lo puede ubicar más cerca de Scrum que de XP.

El punto de partida para este método es el libro escrito por el mismo Alistar Cockburn [Cockburn 2004].

Microsoft Solution Framework

Durante la década de 1990 Microsoft formalizó su metodología de desarrollo de software con el nombre de Microsoft Solution Framework (MSF). Desde sus comienzos propuso un modelo de proceso iterativo e incremental, incluyendo varias prácticas más del espectro de los métodos ágiles. Hacia 2005 Microsoft liberó la cuarta versión de MSF la cual introdujo dos variantes de implementación, una para integración con el modelo propuesto por CMMI y otra para desarrollo ágil.

El núcleo de MSF es el mismo para ambas variantes, la diferencia está a nivel de implementación, ya que para poder integrarse con el modelo CMMI es necesario generar algunos artefactos adicionales. En el núcleo de MSF se define un modelo de equipo, un modelo de gobierno¹⁶⁸, tres disciplinas (gestión de proyecto, gestión de riesgos y gestión de la preparación¹⁶⁹) y un conjunto de principios y modos de pensar¹⁷⁰. Es una metodología completa en el sentido que define un proceso, un conjunto de roles con sus responsabilidades e incluso define en forma explícita las tareas y artefactos sobre los que cada rol debe trabajar.

Resulta especialmente interesante para aquellos que trabajen en el desarrollo de un producto¹⁷¹.

Figura A1.8. Modelo de gobierno de MSF.

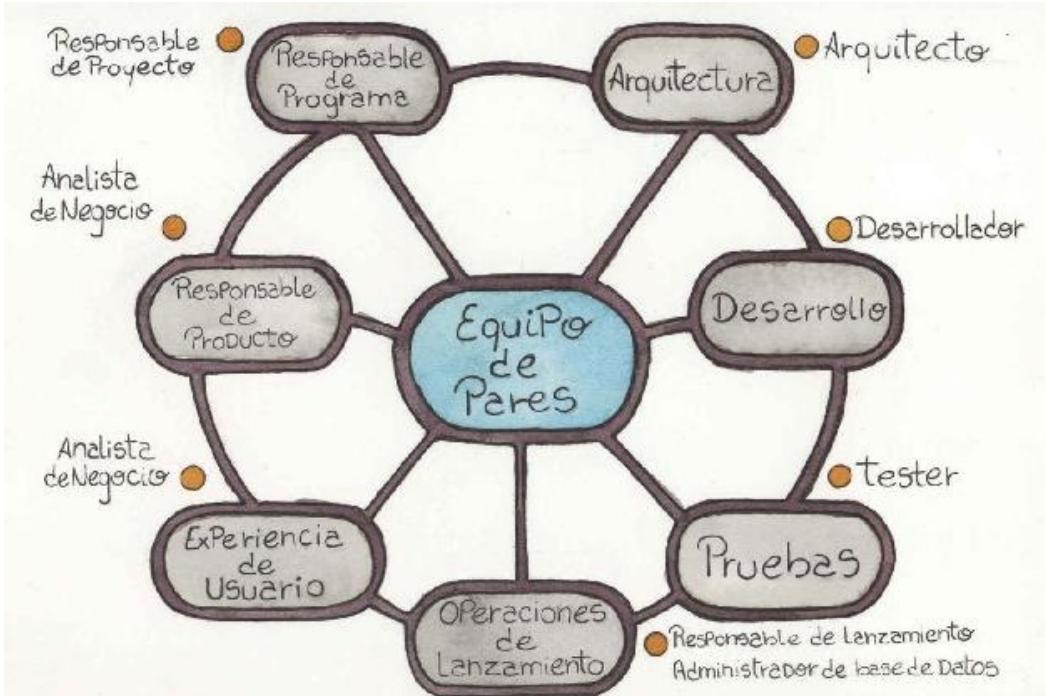
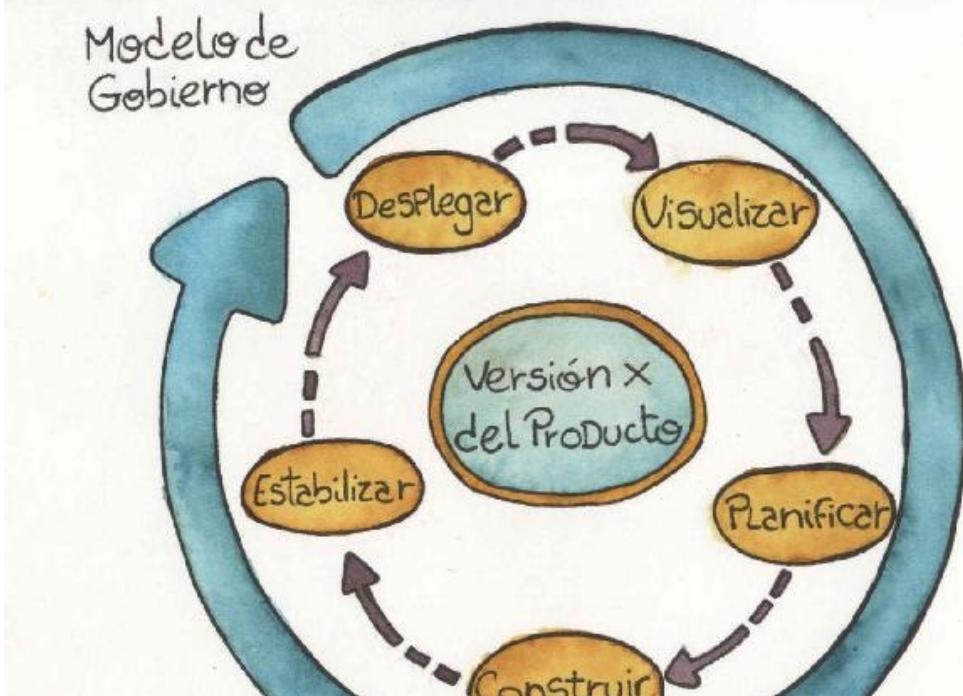


Figura A1.7. Modelo de equipo de MSF.



Lean y Kanban

Lean es una forma de trabajo surgida del estilo de manufactura utilizado por Toyota y caracterizada por siete principios:

1. Eliminar los desperdicios.
2. Ampliar el aprendizaje.
3. Decidir lo más tarde posible.
4. Entregar lo más rápido posible.
5. Potenciar al equipo.
6. Construir integridad.
7. Ver todo el conjunto.

En 2003 Mary y Tom Poppendieck publicaron su libro *Lean Software Development: An Agile Toolkit* [Poppendieck 2003] en el que proponen un conjunto de herramientas para transformar cada principio Lean en prácticas ágiles. Algunas de las herramientas son: identificar desperdicios, utilizar iteraciones y sistemas pull.

Años más tarde, David Anderson utilizó las ideas de Lean en lo que denominó el método Kanban. Algunas características son:

- Se enfoca en el flujo de trabajo para lo cual propone limitar el trabajo en progreso. Este límite aumenta el flujo y la continuidad del trabajo a través de todas las etapas del proceso de desarrollo. En caso de surgir impedimentos, se debe trabajar en ellos para solucionarlos a la brevedad en lugar de agregar nuevas tareas que aumentarían el trabajo en progreso.

- Utiliza prácticas de gestión visual para explicitar y gestionar el flujo de trabajo tal como se explicó en el capítulo “Irradiando Información”. El uso de estas técnicas visuales permite identificar el trabajo en progreso y detectar cuellos de botella con facilidad.
- No utiliza iteraciones. Quizás este punto es el aspecto más polémico de Kanban y por eso dedicamos algunos párrafos más al respecto.

Como ya mencionamos, los sistemas Kanban parten de la base que al limitar el trabajo se permite detectar a la brevedad los cuellos de botella y los problemas, logrando un flujo continuo de tareas. De esa forma, no sería necesario tomarse el tiempo para estimar ni para hacer una revisión completa con el cliente de toda la iteración, como propone Scrum, ya que se van liberando las funcionalidades en la medida en que el trabajo fluye. Por otro lado, si bien las estimaciones le sirven al cliente para poder priorizar funcionalidades según el costo, también se pueden utilizar medidas en base a una simple estimación de orden de magnitud como para poder decidir, por ejemplo, “bajo, medio, alto, muy alto”, y se eliminarían las reuniones de planificación y estimación. También el cliente podría cambiar prioridades en cualquier momento sin generar impactos con el compromiso del equipo, ya que no se trabaja sobre un grupo de tareas de iteración sino sobre lo que más valor aporte en el momento. Así se pueden hacer despliegues a producción a medida que las funcionalidades se terminan y no hay que esperar a que se concluya con todo el trabajo de la iteración.

Ahora bien, los autores consideramos que este esquema de trabajo podría aplicarse en contextos donde los requerimientos son volátiles y poco claros, con prioridades poco definidas y muy cambiantes. Por ejemplo, en servicios de mantenimiento con nuevas tareas que surgen y cambian día a día, y no sobre proyectos con un alcance más concreto y objetivos claros. El foco principal estaría puesto en asegurar el flujo de tareas y analizar constantemente la posibilidad de aumentarlo y eliminar los desperdicios (como los cuellos de botella e impedimentos).

Resulta interesante la propuesta de James Shore [Shore 2008] que tiene el siguiente criterio para decidir cuando utilizar iteraciones o un sistema Kanban:

- Si el equipo es nuevo en la planificación ágil, utilizar iteraciones. Ser disciplinado y cumplir con el criterio de “hecho” en cada iteración, usar slacks, conseguir una velocidad estable que permita comprometerse y cumplir con los compromisos.
- Si se está desarrollando un nuevo producto o una nueva versión para uno existente, usar iteraciones. Crear una visión, hacer un plan de entregas, conseguir un cliente realmente comprometido. El objetivo es crear un producto con un valor significativo y se beneficiará de la previsibilidad que brindan las iteraciones.
- Si está en un ambiente donde las tareas cambian constantemente o son difíciles de predecir, probar con Kanban, que convertirá el caos en un flujo de tareas constante y suave. Por ejemplo, entornos empresariales o mantenimiento de sistemas maduros.

[166](#) En algunos países se lo llama Programación Extrema, pero en este contexto utilizaremos XP que es la abreviatura en inglés de Extreme Programming.

[167](#) En algunos contextos está práctica hace innecesario el uso de revisiones por pares.

[168](#) El término original es governance.

[169](#) El término original es readiness.

[170](#) El término original es mindset.

[171](#) Toda la documentación está disponible online en msdn.microsoft.com/enus/library/jj161047.aspx.

Bibliografía y referencias

- [Aitken 1960] Aitken, Hugh G.J., *Scientific Management in Action: Taylorism at Watertown Arsenal*, Harvard University Press, 1960.
- [Anderson 2007] Anderson, David, Garber, Rick, *A Kanban System for Sustaining Engineering on Software Systems*, disponible en línea: <http://www.agilemanagement.net/AMPDFArchive/KanbanAtLeanNPD.pdf>, 2007, como estaba en febrero de 2013.
- [Anderson 2010] Anderson, David, *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010.
- [Adkins 2010] Adkins, Lyssa, *Coaching Agile Teams, A companion for Scrum Masters, Agile Coaches, and Project Managers in Transition*, Addison-Wesley Professional, 2010.
- [Adzic 2009] Adzic, Gojko, *Bridging the Communication Gap. Specification by example and agile acceptance testing*, Neuri, 2009.
- [Adzic 2012] Adzic, Gojko, *Impact Mapping: Making a big impact with software products and projects*, Provoking Thoughts, 2012.
- [Al-Rawas 1996] Al-Rawas, A., Easterbrook, S. M. "A Field Study into the Communications Problems in Requirements Engineering", Proceedings of the Conference on Professional Awareness in Software Engineering (PACE-96), London, febrero 1996.
- [Arbogast, Larman, Vodde 2012] Tom Arbogast, Craig Larman, and Bas Vodde, *Agile Contracts Primer*, disponible en línea: <http://www.agilecontracts.org/>, como estaba en julio 2013.
- [Austin 2003] Austin, Robert D., Devin, Lee, *Artul Making. What managers need to know about how artists work*, Prentice Hall, 2003.
- [Bass 1997] Bass, Len, Clements, Paul, Kazman, Rick, *Software Architecture in Practice*, SEI Series in Software Engineering, AddisonWesley Professional, 1997, primera edición.
- [Bass 2003] Bass, Len, Clements, Paul, Kazman, Rick, *Software Architecture in Practice*, SEI Series in Software Engineering, AddisonWesley Professional, 2003, segunda edición.
- [Beck 1999] Beck, Kent, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional, 1999 [trad. cast.: *Una explicación de la Programación extrema*, Pearson Addison-Wesley, 2002].
- [Beck 2001] Beck, Kent, et al, *Agile Manifesto*, disponible en línea: www.agilemanifesto.org, 2001, como estaba en octubre de 2012.
- [Beck 2002] Beck, Kent, *Test Driven Development: By Example*, Addison-Wesley Professional, 2002.
- [Bird 2007] Bird Colin, Davies Rachel, *Scaling Scrum*, Conchango, disponible en línea: <http://www.scrumalliance.org/resources/287>, como estaba en 2007.
- [Bizinella 2012] Bizinella Nardon, Fabiane, *Zero Downtime Continuous Deployment of Java Web Applications*, video de la charla del evento JFokus2012, disponible en línea: <http://parleys.com/play/5148922a0364bc17fc56c603>, como estaba en Febrero de 2012
- [Boehm 1981] Boehm, Barry, *Software Engineering Economics*, Prentice Hall, 1981.
- [Boehm 2001] Boehm, Barry, Basili, Victor, *Software Defect Reduction Top 10 List*, IEEE Computery, January 2001.
- [Brooks 1975] Brooks, Frederick P. Jr., "No Silver Bullet", capítulo del libro *The Mythical Man-Month, Essays on Software Engineering*, Addison-Wesley, 1975.
- [Brooks 2010] Brooks, Frederick P. Jr., *The Design of Design, Essays from a computer scientist*, Addison-Wesley Professional, 2010.
- [Brown 2000] Brown, John Seely, Duguid, Paul, *The Social Life of Information*, Harvard Business School Press, Boston, 2000.
- [Burella 2010] Burella, Juan, Rossi, Gustavo, Robles Luna, Esteban, Grigera, Julián, "Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering Approach", Proceedings of the The 11th International Conference on Agile Software Development, SpringerVerlag, LNCS, 2010.
- [Chaplin 2001] Chaplin, Dave, *Test First Programming*, Tech Zone, 2001. [Cockburn 2001] Cockburn, Alistair, *Agile Software Development*, Addison-Wesley Professional, 2001.
- [Cockburn 2004] Cockburn, Alistair, *Crystal Clear, A Human-Powered Methodology for Small Teams*, Addison-Wesley Professional, 2004
- [Cockburn 2008a] Cockburn, Alistair, *Information radiator*, disponible en línea: <http://alistair.cockburn.us/Information+radiator>, 2008, como estaba en febrero de 2013.
- [Cockburn 2008b] Cockburn, Alistair, *Walking Skeleton*, disponible en línea: <http://alistair.cockburn.us/walking+skeleton>, 2008, como estaba en febrero de 2013.
- [Cohn 2003] Cohn, Mike, *Scrum Task Board Training*, disponible en línea: <http://www.mountaingoatsoftware.com/scrum/task-boards>, como estaba en febrero de 2013.
- [Cohn 2004] Cohn, Mike, *User Stories Applied: For Agile Software Development*, Addison-Wesley Professional, 2004
- [Cohn 2005] Cohn, Mike, *Agile Estimating and Planning*, Prentice-Hall, 2005.
- [Coplien 2004] Coplien, James, Harrison, Neil B., *Organizational Patterns of Agile Software Development*, Prentice Hall, 2004.
- [Cunningham 1992] Cunningham, Ward, "The WyCash portfolio management system", OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum), pp. 29-30, ACM, 1992, disponible en línea: <http://c2.com/doc/oopsla92.html>, como estaba en junio de 2013.
- [Cyment 2012] Cyment, Alan, *El espíritu de Scrum, El arte de amar los lunes*, trabajo en progreso, 2012, <http://www.scribd.com/doc/84799747/El-espíritu-de-Scrum>, como estaba en febrero de 2013.
- [Davies 2009] Davies, Rachel, Sedley, Liz, *Agile Coaching*, The Pragmatic Programmers, 2009.
- [De Marco 1987] De Marco, Tom, Lister, Timothy, *Peopleware, Productive Projects and Teams*, Dorset House, 1987
- [De Marco 2002] De Marco, Tom, *Slack, Getting past Burnout, Busywork and the Myth of Total Efficiency*, Broadway, 2002.
- [Derby, Larsen 2006] Derby, Esther, Larsen, Diana, *Agile Retrospectives: Making Good Team Greats*, Pragmatic Programmers, 2006.
- [Devin 2012] Devin, Lee, Austin, Robert D., *The Soul of Design, Harnessing the Power of Plot to Create Extraordinary Products*, Stanford University Press, 2012.
- [Evans 2003] Evans, Eric, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2003.

- [Freeman 2010] Freeman, Steve, Pryce, Nat, *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley Professional, 2010.
- [Fontdevila 2010] Fontdevila, Diego, Salías, Martín, "Software Architecture in the Agile Lifecycle", Microsoft Architecture Journal, Issue 23, March 2010, disponible en línea: <http://msdn.microsoft.com/enus/architecture/f476940>, como estaba en febrero de 2013.
- [Fowler 1999] Fowler, Martin, *Refactoring, Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.
- [Fowler 2003] Fowler, Martin, "Who Needs an Architect", IEEE Software, 3, 2003.
- [Fowler 2003b] Fowler, Martin, *FixedPrice*, disponible en línea: <http://martinfowler.com/bliki/FixedPrice.html>, como estaba en julio 2013.
- [Fowler 2006] Fowler, Martin, *Continuous Integration*, disponible en línea: <http://martinfowler.com/articles/continuousIntegration.html>, como estaba en mayo de 2006.
- [Garlan 1995] Garlan, David, Allen, Robert, Ockerbloom, John, "Architectural Mismatch, or Why it's hard to build systems out of existing parts", Proceedings of the 17th International Conference on Software Engineering (ICSE-17), April 1995. Una versión revisada y extendida de este trabajo apareció en (EEE Software, Volume 12, Issue 6, Nov. 1995 (pp. 17-26).
- [Glover 2007] Glover, Andrew, *Is BDD TDD Done Right?*, disponible en línea: <http://hendryluk.wordpress.com/2009/07/17/bddtdd-done-right/>, como estaba en octubre de 2012.
- [Goleman 2000] Goleman, Daniel, "Leadership that gets results", Harvard Business Review, 2000.
- [Humble, Farley 2010] Humble, Jez, Farley, David, *Continuous Delivery*, Addison-Wesley, 2010.
- [Humble 2012] Humble, Jez, *Q & A with Jez Humble: "Continuous Delivery"*, GOTO Conference 2012, disponible en línea: <http://gotocon.com/aarhus-2012/qandawithjezhumble>, como estaba en octubre de 2012.
- [Humble, Fowler 2012] Humble, Jez, Fowler, Martin, Hanselman Scott, *Continuous Delivery with Jez Humble and Martin Fowler*, disponible en línea <http://www.hanselminutes.com/339/continuous-deliverywith-jez-humble-and-martin-fowler>, como estaba en octubre de 2012.
- [Humphrey 2001] Humphrey, Watts S., *Winning with Software: An Executive Strategy*, Addison-Wesley Professional, 2001.
- [Humphrey 2005] Humphrey, Watts S., *TSP, Leading a Development Team*, SEI Series in Software Engineering, Addison-Wesley Professional, 2005.
- [Humphrey 2005b] Humphrey, Watts S., *PSP, A Self-Improvement Process for Software Engineers*, SEI Series in Software Engineering, Addison-Wesley Professional, 2005.
- [Jacobson 1992] Jacobson, Ivar, Christerson, Magnus, Jonsson, Patrik, Overgaard, Gunnar, *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Professional, 1992.
- [Kerth 2001a] Kerth, Norman, *Project Retrospectives: A Handbook for Team Reviews*, Dorset House Publishing, 2001.
- [Kerth 2001b] Kerth, Norman, *The Retrospective Prime Directive*, disponible en línea: <http://www.retrospectives.com/pages/retroPrimeDirective.html>.
- [Kniberg 2007] Henrik Kniberg, *Scrum and XP from the Trenches*, disponible en línea: <http://www.infoq.com/minibooks/scrum-xp-from-the-trenches>, como estaba en junio de 2007.
- [Koskela 2007] Koskela, Lasse, *Test Driven: TDD and Acceptance TDD for Java Developers*, Manning Publications, 2007.
- [Lacey 2008] Lacey, Mitch, *How Do We Know When We Are Done*, disponible en línea: <http://www.scrumalliance.org/community/articles/2008/september/how-do-we-know-when-we-are-done>, como estaba en septiembre de 2008.
- [Larsen 2009] Larsen, Diana, Charla titulada "Trust: The Key to Agile Team Collaboration", Agiles 2009, 2da Conferencia Latinoamericana de Metodologías Ágiles, Florianópolis, Brasil, octubre de 2009. disponible en línea: <http://agiles2009.agiles.org/es/session.php?id=88>, como estaba en febrero de 2013.
- [Marick 2002] Marick, Brian, *Boundary Objects*, disponible en línea <http://www.exampler.com/testing-com/writings/marick-boundary.pdf>, como estaba en octubre de 2012.
- [Marick 2003] Marick, Brian, *Exploration Through Example*, <http://www.exampler.com/old-blog/2003/08/21/>, como estaba en octubre de 2012.
- [Mayer 2009] Mayer, Tobias, Simple Scrum, Agile Anarchy Blog, disponible en línea: <http://agileanarchy.wordpress.com/2009/09/20/simplescrum/>, 20 de septiembre, 2009, como estaba en febrero de 2013.
- [Mayer 2013] Mayer, Tobias, *The People's Scrum, Agile Ideas for Revolutionary Transformation*, Dymaxicon, 2013.
- [McConnell 2006] McConnell, Steve, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996.
- [McConnell 2006] McConnell, Steve, *Software Estimation: Demystifying the Black Art*, Microsoft Press 2006.
- [Meszaros 2003] Meszaros, Gerard, Bohnet, Ralph, Andrea, Jennitta, "Agile Regression Testing Using Record & Playback", presentado en OOPSLA 2003.
- [Meszaros 2007] Meszaros, Gerard, *xUnit Test Patterns*, Addison-Wesley Professional, 2007.
- [Mugridge 2005] Mugridge, Rick, Cunningham, Ward, *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005.
- [Mugridge 2008] Mugridge, Rick, "Managing Agile Project Requirements with Storytest-Driven Development", IEEE software 25, pp. 68-75, 2008.
- [North 2006] North, Dan, *Introducing BDD*, Better Software, 2006. [Patton 2005] Patton, Jeff, *It's all in how you slice*, Better Software, 2005, p.295.
- [Pawlak 2009] Pawlik, Greg, *The Toyota System, One Piece Flow*, disponible en línea: http://www.thetoyotasystem.com/lean_concepts/one_piece_flow.php, como estaba en enero de 2014.
- [Pink 2011] Pink, Daniel H., *Drive, The Surprising Truth about what Motivates Us*, Riverhead Books, 2011.
- [Pirsig 1974] Pirsig, Robert, *Zen and the Art of Motorcycle Maintenance*, Riverhead Books, 1974.

[Poppendieck 2001] Poppendieck, Mary, *Lean Programming*, disponible en línea: <http://www.drdobbs.com/lean-programming/184414734>, como estaba en febrero de 2013.

[Poppendieck 2003] Poppendieck, Mary, *Lean Software Development: an agile toolkit*, Addison Wesley, 2003.

[Poppendieck 2004] Poppendieck, Mary, “*Team Compensation*”, Better Software, July / August 2004, disponible en línea: www.poppendieck.com/pdfs/Compensation.pdf, como estaba en febrero de 2013.

[PMBOK4] PMI, *A Guide to the Project Management Body of Knowledge: PMBOK Guide*, 4th Edition, 2008.

[Pressman 1998] Roger S. Pressman, *Ingeniería de Software, Un enfoque práctico*, Mc Graw Hill, 1998.

[Pichler r2013] Pichler, Roman, *The Product Canvas*, disponible en línea: <http://www.romanpichler.com/blog/agile-product-innovation/the-product-canvas/>, como estaba en mayo de 2013.

[Quesada 2009] Xavier Quesada Allue, *Elements of taskboard design*, Visual Management Blog, disponible en línea: <http://www.xqa.com.ar/visualmanagement/elements-of-taskboard-design/>, como estaba en febrero de 2013.

[SEI 2013] Software Engineering Institute, *Architecture and agile*, SATURN Blog, disponible en línea: <http://saturnnetwork.wordpress.com/category/architecture-and-agile/>, como estaba en junio de 2013.

[Schwaber 2004] Schwaber, Ken, *Agile Project Management with Scrum*, Microsoft Press, 2004.

[Schwaber 2011] Schwaber, Ken, *Product Owners not Proxies*, disponible en línea: <http://kenschwaber.wordpress.com/2011/01/31/product-owners-not-proxies/> como estaba en enero de 2011.

[ScrumOrgDoD] disponible en línea: <http://www.scrum.org/Resources/Scrum-Glossary/Definition-of-Done>.

[Shore 2007] Shore, James, Warden, Shane, *The Art of Agile Development*, O'Reilly, 2007.

[Shore 2008] Shore, James, *Kanban Systems*, disponible en línea: <http://www.jamesshore.com/Blog/Kanban-Systems.html>, como estaba en febrero de 2013.

[Shore 2010] Shore James, Belshee Arlo, *Single Piece Flow in Kanban: A How-To*, disponible en línea: <http://www.infoq.com/presentations/Single-Piece-Flow-Kanban>, como estaba en enero de 2014.

[Sommerville 2005] Sommerville, Ian, *Ingeniería de Software*, séptima edición, Pearson Educación, 2005.

[Stewart 2011] Stewart, Simon, *Page Objects*, disponible en línea: <http://code.google.com/p/selenium/wiki/PageObjects>, como estaba en junio de 2011.

[Sutherland 95] Sutherland, Jeffrey Victor; Schwaber, Ken, “*Business object design and implementation*”, OOPSLA '95 workshop proceedings, The University of Michigan, p. 118.

[Version One 2012] *7th Annual State of Agile Development Survey*, disponible en línea: <http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf>, como estaba en julio de 2013.

[Waters 2007] Waters, Kelly, 2007, *Definition of DONE! 10 Point Definition of DONE! 10 Point Checklist*, disponible en línea: <http://www.allaboutagile.com/definition-of-done-10-point-checklist/>, como se estaba en Abril de 2007.

[Wirfs 2002] Wirfs-Brock, Rebecca; McKean, Alan, *Object Design: Roles, Responsibilities, and Collaborations*, Addison Wesley, 2002.

