UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Análisis y adaptación de BDD en un desarrollo semi-ágil: un caso de estudio.
Autores: Tamara Torres
Director: Roxana Giandini
Codirector: -
Asesor profesional: -
Carrera: Licenciatura en Sistemas

Resumen

Hoy en día, luego de más de una década en la cual la ingeniería de software ha evolucionado, las metodologías Agiles no sólo proporcionan beneficios, sino que también se ha convertido es un 'MUST' para las empresas exitosas, ya que han demostrados ser efectivas.

*Dentro del variado conjunto de metodologías, BDD nació originalmente con la intención de cambiar la palabra **test** por **should**, haciendo foco en lo verdaderamente importante del software: su comportamiento. Para lograrlo es que su principal pilar es la premisa: "Utilizar ejemplos que nos permitan describir comportamiento".*

Teniendo en cuenta la evolución que ha tenido BDD en los últimos 10 años así como también sus orígenes, buscamos en esta tesina mejorar el proceso de desarrollo de software a partir de la implementación de una guía practica basada en BDD para un desarrollo semi- ágil. Para lograrlo realizamos una simulación de modo tal de determinar la utilidad de dicha guía.

Palabras Claves

Metodologias agiles
BDD (Behavior-driven development)
TDD (Test Driven Development)
ATDD (Acceptance Test Driven Development)
DDD(Domain Driven Design)
Desarrollo semi- ágil,
Automatización de pruebas,
Documentación viva.

Conclusiones

Esta tesina presenta una guía de implementación práctica para incorporar la metodología BDD en un desarrollo semi-ágil. El aporte principal reside en que la guía beneficie a quienes incursionan en dicha metodología en el contexto de desarrollo planteado. Los resultados obtenidos de aplicar esta guía en un caso de estudio simulado fueron satisfactorios.

Además se incluye una análisis comparativo entre metodologías: TDD, ATDD, DDD y BDD.

Trabajos Realizados

-Estudio y análisis del estado del arte, realizando comparaciones entre las distintas metodologías.

-Elaboración de una guía práctica.

-Puesta en práctica sobre un caso de estudio simulado.

-Análisis de los resultados.

Trabajos Futuros

- Incorporar un conjunto de mejoras a la guía propuesta, de forma tal de crear una versión más completa que permite obtener resultados mas detallados.

- Realizar una nueva puesta en práctica, esta vez en un caso más complejo, en donde se pueda realizar un seguimiento completo y detallado. Esto permitirá un análisis más profundo de los resultados

Fecha de la presentación: **Octubre 2016**

Análisis y adaptación de BDD en un desarrollo semi-ágil: un caso de estudio.

Autor: Tamara Torres

Directora : Dra. Roxana S. Giandini

Tesina de Licenciatura en Sistemas

Facultad de informática

UNIVERSIDAD NACIONAL DE LA PLATA

2016

Agradecimientos

A toda mi familia, por estar presente en todos estos años durante los cuales comencé este camino facultativo, en especial a mis padres *Mario* y *Laura*, que siempre me dieron su apoyo incondicional sin el cual no hubiese logrado estar hoy aquí.

A mi abuelo *Rafa* que me mira desde el cielo y espero esté orgulloso de mí y a mi abuela *Dora* que es mi fiel confidente y siempre me acompaña.

A mi amor, *Alejandro*, mi compañero de ruta y de vida, con el cual no solo comparto el amor por esta profesión que es la informática, sino que el amor que me da, me dio el valor para desarrollar esta tesina.

A mi directora, *Roxana*, que desde el primer momento que me acerque a ella para pedirle que sea mi directora, no titubeo y supo guiarme en el camino.

Índice General

Capítulo 1. Introducción

En esta sección se presenta la motivación de la presente tesina, así como también el objetivo que se busca alcanzar y los resultados que se esperan obtener. Al finalizar la sección se realiza una breve descripción de la organización del trabajo.

1.1 Motivación

Hoy en día, luego de más de una década en la cual la ingeniería de software ha evolucionado, así como también las metodologías que la acompañan, la metodología Agile (6) no es solo un concepto atrayente para ser usado, sino que también se ha convertido es un *'MUST'* para las empresas exitosas. Sin embargo, aún hallamos difícil la tarea de desarrollar "*software that matters"* (4), es decir software que tiene un valor para el negocio.

Una de las primeras metodologías surgida luego del Manifiesto Ágil (9) fue TDD (Test Driven Development). Básicamente podemos decir que el foco de TDD son los *Tests* y su mayor premisa es: "*Write the test first*" (8). A partir de TDD, se fueron abriendo camino otras metodologías como ATDD (Acceptance Test Driven Development), que el lugar de escribir primero los *test*, se escriben primero las *pruebas/criterios de aceptación* (7). Otra metodología que también se abrió paso a partir de TDD fue DDD (Domain Driven Design) que propone en cambio pensar en términos de especificaciones o comportamiento (10) (14).

En este contexto, BDD (Behavior-driven development), surgió originalmente con la intención de cambiar la palabra *test* (proveniente del uso de TDD) por *should*, queriendo así mejorar la comunicación entre desarrolladores y testers (2), haciendo foco en lo verdaderamente importante del software: su comportamiento. Para lograrlo, BDD se basa en la premisa: "*Utilizar ejemplos que nos permitan describir comportamiento"* (12). Esto último es particularmente interesante, ya que, en lugar de pensar en pruebas o test, (tal como TDD promueve) hablamos de describir comportamiento (como plantea DDD), lo que a su vez significa considerar la especificación de criterios/pruebas de aceptación (tal como plantea ATDD).

Hoy en día BDD, ha evolucionado más allá de lo que su creador Dan North hubiese podido imaginar hace más de 10 años, sin embargo, aún es válida la afirmación de que BDD no hace más que *parafrasear* buenas prácticas existentes, basándose en: *"Getting the words right".* (11)

Teniendo en cuenta todo lo mencionado previamente, la motivación de esta tesina es poder mejorar el proceso de software a partir de una adaptación de BDD en un proyecto de desarrollo semi- ágil.

1.2 Objetivo

El objetivo de esta tesina es analizar y adaptar BDD (Behavior-driven development) a un proceso de desarrollo semi- ágil sobre un caso de estudio simulado.
Como resultado se busca determinar las ventajas y mejoras alcanzadas sobre dicho proceso al aplicar BDD mediante el uso de una guía práctica.

Específicamente para lograr este objetivo general se buscará desarrollar los siguientes subjetivos:

- Estudio y análisis del estado del arte.
- Elaboración de una guía práctica.
- Puesta en práctica sobre un caso de estudio simulado.
- Análisis de los resultados.

1.3 Estructura de la Tesina

La presente tesina se encuentra divida en distintos capítulos, dentro de cada uno de ellos iremos evolucionando en el desarrollo de la misma; partiendo de un marco teórico, pasando por la creación de una guía práctica de implementación, la descripción del caso de estudio, la ejecución de una simulación sobre dicho caso de estudio, y a partir de ella analizaremos los resultados para finalmente mencionar las conclusiones alcanzadas y cuáles son los pasos a futuro.

En otras palabras, en el **Capítulo 2,** definiremos el marco teórico, aquí hablaremos de TDD (Test Driven Development), ATDD (Acceptance Test Driven Development), DDD (Domain Driven Design) y finalmente BDD (Behavior-driven development). Mencionaremos además ciertas comparaciones y diferencias entre todas ellas.

En el **Capítulo 3**, describimos la guía práctica que es el foco central de la tesina, partiendo de un cuestionario inicial y luego describiendo cada una de las fases de implementación.

En el **Capítulo 4**, veremos cuál es nuestro caso de estudio y cuáles son las características particulares, siendo de mayor importancia aquellas que hacen a este caso de estudio, un desarrollo semi- ágil. Además, veremos los resultados de la simulación de la guía práctica aplicada sobre el caso de estudio.

En el **Capítulo 5,** analizaremos los resultados luego de la simulación, mencionaremos las ventajas, desventajas, aportes para cada rol dentro del equipo de desarrollo y además enunciaremos una serie de mejoras a la guía.

Por último, en el **Capítulo 6**, mencionaremos las conclusiones alcanzadas y cuáles son los trabajos propuestos a futuro.

Capítulo 2. Marco Teórico

Definiremos en este capítulo, el marco teórico de este trabajo. Haremos una introducción de las metodologías de desarrollo: TDD (Test Driven Development), ATDD (Acceptance Test Driven Development), DDD (Domain Driven Design) y finalmente BDD (Behavior-driven development). Incluiremos además, comparaciones y diferencias entre ellas.

2.1.TDD (Test Driven Development)

"If it's green, the code is clean" , JUnit Slogan.

2.1.1 Qué es TDD?

Test Driven Development o su traducción en castellano "*Desarrollo guiado por pruebas*" es una *práctica de diseño de software iterativa*, surgida como parte del "*Extreme Programming*" que se basa en dos reglas principales:

- Escribir las pruebas automatizables antes que el código a probar
- Eliminar duplicaciones de código (Esto se lleva a cabo con "*Refactoring*[1]")

Tal como la describe Beck (8), "***TDD es una manera de manejar el miedo durante la programación***".

Podemos decir que, en cierta manera, manejar el miedo al programar, es el fin último de TDD, ya que gracias a su ciclo iterativo (Ver Figura 1) de rojo (el test falla), verde (se codificó y el test pasa), refactoring (ahora que logramos que pase el test, mejoramos el código), brinda al desarrollador una confianza en el código que está siendo programado.

A partir de las reglas en las cuales se basa TDD, podemos desglosar las tareas que debe cumplir el desarrollador:

- Se deben escribir las pruebas antes que el código. => Esto implica un entendimiento del comportamiento (QUÉ) más allá de su implementación (CÓMO)
- Se deben automatizar las pruebas => Esto implica que cada vez que se realice una modificación todo el conjunto de pruebas se ejecuta para corroborar que el/los cambio/s se realizaron con éxito.
- Se deben eliminar duplicados => Esto implica realizar refactoring constante para mantener una buena calidad de código.

[1]Fowler : "*Refactoring es una técnica disciplinada para reestructurar un código existente, alterando su estructura interna pero sin modificar su comportamiento externo*" http://refactoring.com/

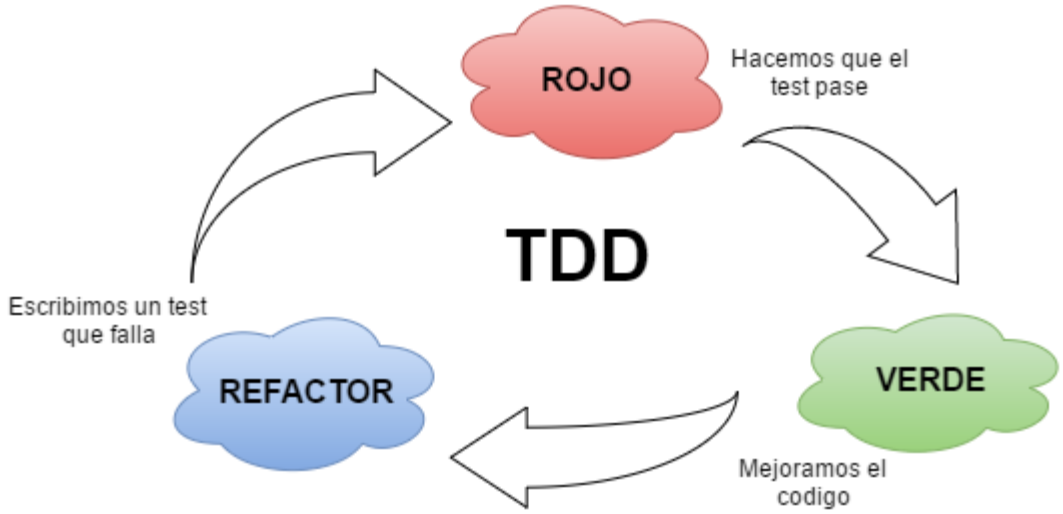


Figura 1. Ciclo Iterativo TDD

2.1.2 Patrones y Anti patrones de TDD

Si bien cuando hablamos de TDD, principalmente nos referimos al ciclo iterativo que mencionamos recientemente (Rojo-Verde-Refactor), existe además una serie de patrones que nos brindan un aporte extra.

Tal como define Beck (8) los patrones de TDD son:

- Prueba ("*Test*"). Probar significa evaluar, ningún desarrollador entrega ni siquiera un cambio mínimo sin antes haber probado que funcione. Es por ello que buscamos crear pruebas automatizadas de manera de acelerar la validación.
- Prueba aislada ("I*solate Test"*). La ejecución de las pruebas no debe afectar a ninguna otra, y cada prueba debe poder ejecutarse de manera independiente, es por ello que hablamos de pruebas aisladas, no deben existir dependencias entre ellas, ya que de esta manera podemos a su vez aislar e identificar los problemas cuando las pruebas fallan.
- Lista de pruebas ("*Test List*"). Es preciso saber qué debemos probar antes de comenzar a preparar nuestras pruebas, es por ello que la mejor opción es realizar un listado de ellas, de esta manera logramos una organización y una mejor focalización en el objetivo que queremos alcanzar.
- Probar primero ("*Test First*"). El mejor momento de escribir las pruebas es ni más ni menos que antes de escribir el código que va a ser evaluado, una de las reglas de oro de TDD.

- Verifica primero ("*Assert First*"). El momento adecuado de escribir las validaciones es en el momento que se escriben las pruebas, ya que dichas validaciones son parte de las pruebas que deseamos escribir antes que el código que implementará la funcionalidad. El momento indicado para crear las verificaciones de nuestro código es sencillamente antes de escribirlo, ya que sabemos una cosa por seguro: el mismo fallará porque aún no hemos escrito nuestro código que implementa aquello que debemos validar.
- Datos de prueba ("*Test Data*"). Los datos en las primeras pruebas deberían ser datos que hagan la prueba fácil de leer y de seguir, una buena alternativa para ello es utilizar datos realísticos o reales, es decir usar datos del mundo real.
- Datos evidentes ("*Evident Data"*). Una de las ideas principales de las primeras pruebas o "Test first" es que las pruebas sean lo suficientemente legibles y entendibles por sí mismas. Con este propósito es que es importante representar la intención de los datos, y para ello incluimos en las pruebas tanto el resultado esperado como el resultado actual, esto nos permite dar un indicio de lo que se busca con la prueba.

Si en cambio hablamos de anti patrones en TDD, tal como describe James Carr (14) podemos mencionar:

- El mentiroso ("*The Liar*"). En este caso todas las verificaciones ("*asserts*") se cumplen, pero el test no está probando la funcionalidad que realmente se estaba buscando probar.
- Configuración Excesiva ("*Excessive setup*"). Hay demasiado código en la configuración, que impide ver la finalidad de lo que está siendo probado.
- El gigante ("*The Giant*"). Sucede cuando el test tiene demasiado código.
- El emulador ("*The Mockery*") Aquellos test en los cuales se abusa del uso de mocks[2].
- El inspector ("*The inspector"*). Pruebas en las cuales la encapsulación es violada.
- Sobras abundantes ("*Generous leftovers*"). Sucede cuando el test viola la propiedad de aislamiento.
- El héroe local ("*The local hero*"). Las pruebas solo funcionan con una configuración o plataforma específica.
- El quisquilloso ("*The nitpicker*"). Se validan demasiados detalles que no son significativos para con la finalidad de la prueba.
- El cazador secreto ("*The secret catcher*"). Existe una ausencia de validaciones en la prueba, pero se incluye la verificación de excepciones.
- El desertor ("*The dodger*"). Se verifican muchos efectos colaterales, pero no aquellos referidos a la finalidad de la prueba.
- El alborotador ("*The loudmouth*"). Sucede cuando se abusa del uso de logging[3] en la consola.
- Cazador glotón ("*The greedy catcher*"). Las excepciones son capturadas, pero se ignoran.
- El secuenciador ("*The sequencer*"). Los datos de las validaciones son fijos y en el mismo orden.

[2] Se llaman Mock a los objetos que imitan el comportamiento de objetos reales de una forma controlada y se usan para probar a otros objetos

[3] Término anglosajón para referirse a la palabra bitácora en español.

- Dependencia oculta ("*Hidden dependency*"). Se presume la existencia de datos que no son explicitados en la prueba.
- El enumerador ("*The enumerator*"). Los nombres de los métodos son enumerativos en lugar de ser descriptivos.
- El extraño ("*The stranger*"). Sucede cuando se llevan a cabo pruebas sobre un objeto que no es el evaluado.
- Evangelista del sistema operativo ("*The operating system evangelist*"). Aquellas pruebas que dependen del sistema operativo donde se ejecutan.
- A pesar de todo funciona ("*Success against all odds*"). Se escribe la prueba luego del código a probar.
- El único ("*The one*"). Se prueba en un mismo método todos los escenario o casos de uso.
- El mirón ("*The peeping tom*"). Se utilizan datos compartidos entre distintas pruebas lo que hace que una prueba falle aun cuando el sistema probado es válido.
- La Tortuga ("*The slow poke*"). Pruebas muy lentas en su ejecución.

2.1.3 Beneficios de TDD

Entre los beneficios que nos brinda TDD, podemos mencionar como principales:

- Mejora la ***calidad del código*** implementado. En este punto debemos destacar que el código obtenido es más flexible, modular, desacoplado, fácil de entender, fácil de mantener. Y por sobre todas las cosas es un código confiable.
- Permite crear una ***suite de regresión*** robusta.
- Ayuda a los programadores a descubrir si el código que están desarrollando cumple con lo esperado, permitiéndoles saber si se encuentran por buen camino.
- Los defectos detectados tempranamente, eso se debe a que cada módulo dentro de una funcionalidad tiene su propia prueba, ya que debería cumplirse la premisa "*No se codifica ni una línea de código sin su prueba*". Esto también produce que los defectos encontrados luego se reduzcan drásticamente.

2.1.4 El lado oscuro de TDD

Si bien TDD es una buena práctica que se podría implementar en casi cualquier tipo de desarrollo en pos de una búsqueda no solo de mejorar la calidad del código, sino también de una mejora en el trabajo diario del desarrollador, una mayor confianza y seguridad en el código obtenido y de yapa pruebas automatizadas que le sirven al QA cómo base de su regresión, dicha afirmación no es tan cierta. Para poder alcanzar todo esto se requiere:

- Disciplina
- Perseverancia
- Práctica, práctica, práctica.

A pesar todo lo que nos brinda TDD, no es posible abusar y buscar utilizar esta práctica en casos en los cuales NO aplica. Un ejemplo es en el mantenimiento de *código Legacy*: originalmente TDD fue concebida para la construcción de aplicaciones desde el comienzo, lo cual hace muy complejo su realización a la inversa, es decir a partir del código ya escrito y funcionando, crear las pruebas, ya que, si bien sería posible, las pruebas dejarían de estar guiando el desarrollo.

Otro ejemplo de caso en el cual no aplicar TDD es cuando lo que buscamos es implementar la capa de interfaz de usuario de una aplicación. Esta capa de abstracción suele no cuadrar demasiado con lo que plantea TDD, haciendo que en cambio sea útil para llevar a cabo la implementación de otras capas, como la de lógica de negocio.
Por último, cabe mencionar que TDD no es la mejor opción para situaciones en donde se requiera integración de Base de datos, esto se debe principalmente a que en dichos casos al modificar la base es necesario volverla al estadio previo para poder continuar, lo cual dificulta la utilización de la técnica.
Todos estos casos son bien descritos en el trabajo de Fontela (7) en donde se realiza un análisis más detallado de esta metodología.

2.1.5 Herramientas

En lo que respecta a las herramientas que implementan esta forma de desarrollo basada en pruebas, podemos mencionar como las más destacadas:

- **JUnit** [4] Es un framework de Java que nos permite escribir pruebas repetibles.
- **XUnit** [5] Es una herramienta de test unitario para .NET, de código abierto y focalizada en comunidad.
- **VBUnit**[6] Es una herramienta de creación de test unitarios para Visual Basic y objetos COM.
- **CUnit** [7] Es un sistema liviano para la escritura, administración y ejecución de pruebas en C.
- **HtmlUnit**[8] Es un framework de pruebas unitarias que permite crear un navegador sin UI[9] para programas JAVA y brinda soporte para JavaScript y AJAX.

Como se puede observar todas forman parte de la familia de herramientas xUnit, con lo cual su propósito principal son las pruebas unitarias; con ellas se especifica, diseña y verifica el código.
Existen otras herramientas que nos permiten una mayor variedad de tipos de pruebas (de integración, de interacción, etc.) pero no las mencionaremos en el presente trabajo.

[4] http://junit.org/junit4/
[5] https://xunit.github.io/
[6] http://www.vbunit.com/
[7] http://cunit.sourceforge.net/
[8] http://htmlunit.sourceforge.net/
[9] UI es un acrónimo que parte del anglicismo User interface y se refiere a la creación de la interfaz, ya sea gráfica o desarrollada con diversas tecnologías web como css, jQuery, jQuery UI, EXT JS, YUI, etc.http://blog.eltallerweb.com/diferencias-entre-ui-y-ux/

2.2 ATDD (Acceptance Test Driven Development)

"Begin with the end in mind.", Stephen R. Covey

2.2.1 Qué es ATDD?

Acceptance Test Driven Development o su traducción al castellano "Desarrollo guiado por pruebas de aceptación" (también conocido como "*Especificación basada en ejemplos*"), es una práctica de diseño de software en la cual el equipo entero junto con los *stakeholders*[10] discute de manera *colaborativa* los criterios de aceptación, siendo ésta la mejor manera de asegurar que todos comparten el mismo entendimiento acerca de lo que se va a construir.

Cuando hablamos de ATDD, decimos que lo que escribiremos primero son los criterios y las pruebas de aceptación, sin lo cual no es posible comenzar a codificar la aplicación/sistema; en ATDD nos focalizamos en el QUE de una aplicación en lugar del COMO.

*ATDD es esencialmente una **actividad** grupal de equipo, es un **proceso** en equipo.*

El ciclo de ATDD consta de los siguientes pasos:

- Seleccionar una historia de usuario. (En este punto asumimos que se realizó una priorización que nos permite saber cuál historia de usuario es la próxima elegible, en general esto se da luego de algún workshop[11] (ej. Story Planning[12]) en el cual los DEV, QA y BA obtuvieron información al respecto)
- Escribir las pruebas de aceptación para dicha historia. (Aquí es donde todos los involucrados deben participar colaborativamente)
- Automatizar las pruebas. (Para este paso existen variadas formas de llevarla a cabo, sin embargo, lo importante es que a partir de las pruebas de aceptación del paso anterior obtenemos pruebas ejecutables, que nos permitirán conocer cuánto de la funcionalidad aún falta)
- Implementar el código. (Al igual que el paso anterior existen diversas formas de hacerlo, una de ellas es haciendo uso de TDD)
- Repetir. (Es la base de esta práctica al igual que TDD, es una práctica iterativa que finaliza cuando los requerimientos del cliente han sido satisfechos)

[10] "***Stakeholder** es una palabra del inglés que, en el ámbito empresarial, significa 'interesado' o 'parte interesada', y que se refiere a todas aquellas personas u organizaciones afectadas por las actividades y las decisiones de una empresa.*" http://www.significados.com/stakeholder/

[11] "*(...) evento en el cual los asistentes pueden formarse sobre un determinado tema de manera intensiva.*" http://definicion.de/workshop/

[12] Lo veremos en el capítulo que viene detalladamente.

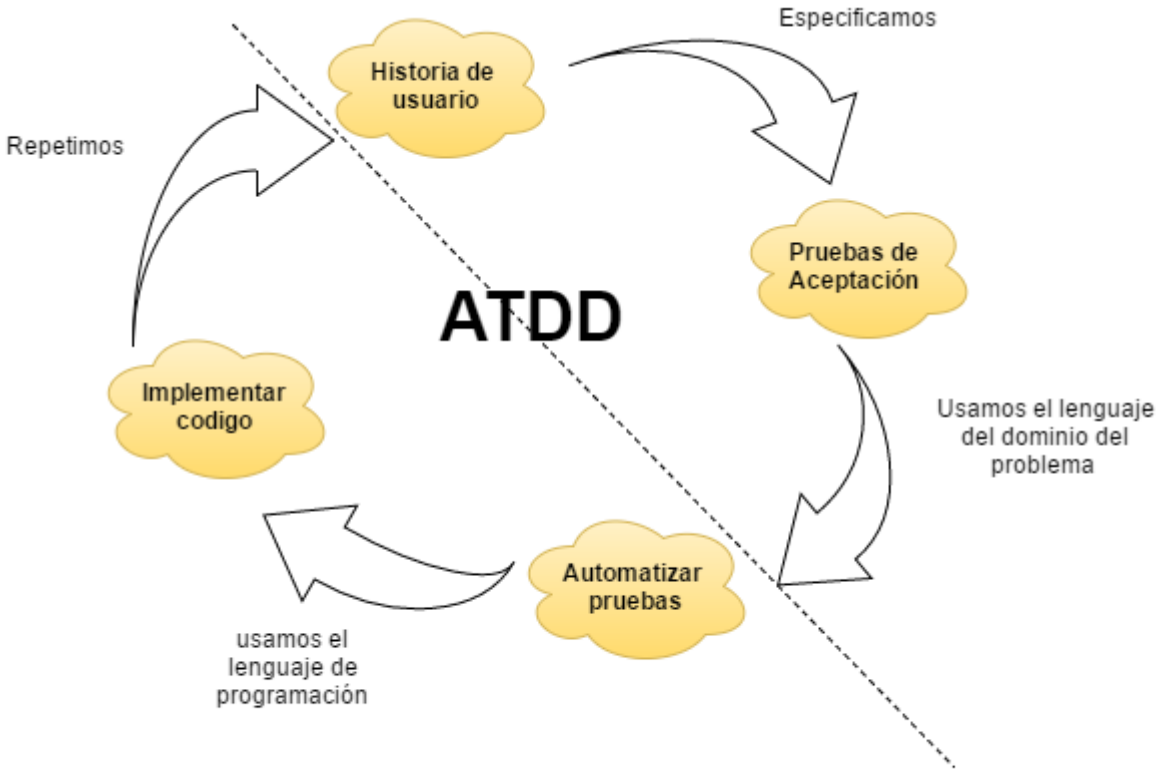


Figura 2. Ciclo Iterativo ATDD

2.2.2 Qué diferencia a ATDD de TDD?

A diferencia de TDD que se centra en la creación de pruebas unitarias primero, ATDD se encarga de la creación de pruebas de aceptación primero; estas nos permiten saber si el producto que estamos creando es el *correcto para el negocio*, en lugar de verificar que el código sea *técnicamente correcto*. Ambas técnicas pueden utilizarse de manera complementaria, ya que de hecho ATDD surgió como una técnica complementaria y más abarcativa de TDD.

Pruebas Unitarias	**Pruebas de aceptación**
Aseguran que el código está correctamente construido	Aseguran que el código es el correcto a construir
Es el código correcto técnicamente?	Es el producto correcto según el negocio?
Es importante que se ejecuten rápidamente y nos permiten obtener un feedback de cómo estamos trabajando.	Es importante que sean fáciles de entender y que nos permitan determinar si la funcionalidad está completa.
Definen QUÉ hace una funcionalidad concreta que debemos construir.	Definen QUÉ hace el sistema/aplicación a construir.

En otras palabras, decimos que ATDD nos ayuda a construir un software de alta calidad que satisface las necesidades del negocio de manera tan fiable, así como TDD nos ayuda a asegurar la calidad técnica del software.

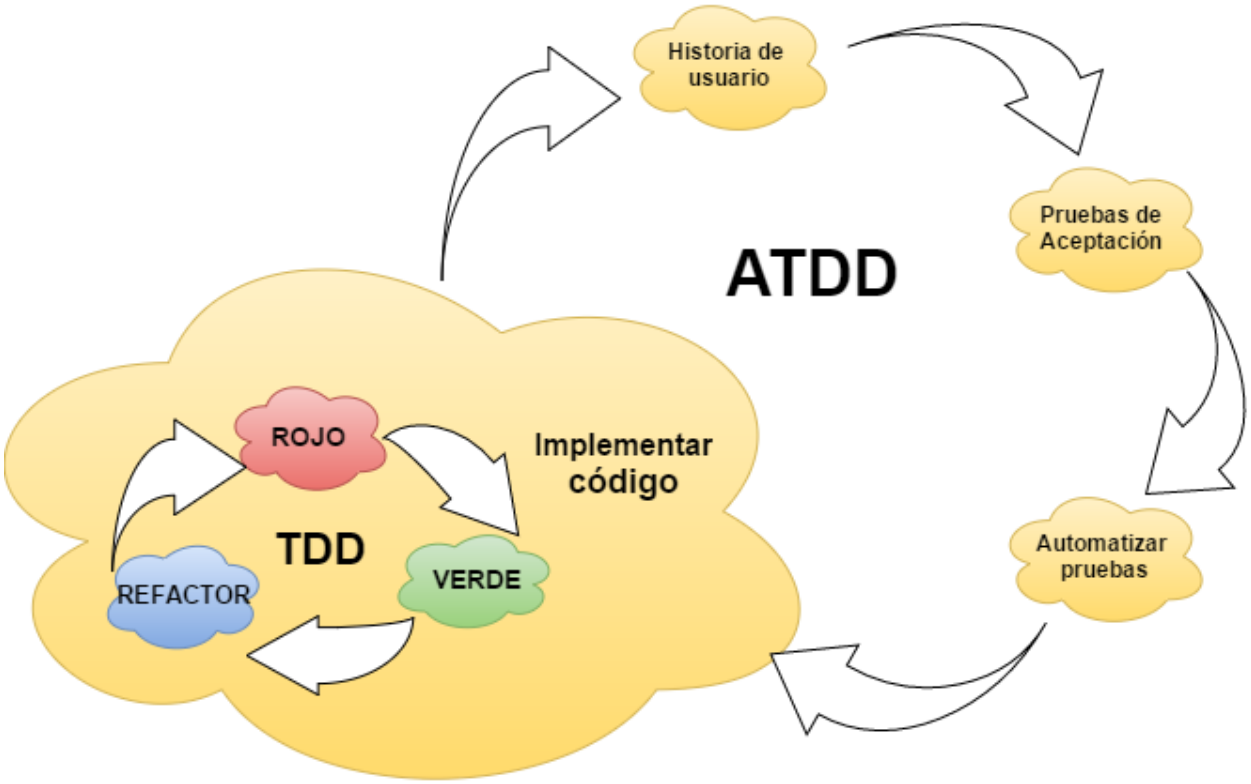


Figura 3. Ciclo Iterativo ATDD + TDD.

2.2.3 Historias de usuario y pruebas de aceptación

Una historia de usuario es una forma extremadamente sencilla de expresar requerimientos. Las historias de usuario representan los requerimientos en sí mismos, actuando como una promesa para futuras conversaciones entre el cliente y el desarrollador. Las historias de usuario ayudan a los equipos a priorizar y definir el alcance de los proyectos, de manera tal que los clientes obtengan un valor real mucho más rápido y que el tiempo y esfuerzo sean consumidos eficientemente.

*Las historias de usuario verbalizan o expresan el **QUÉ** aporta valor al cliente y no **CÓMO** el sistema proveerá de tal valor. (19)*

A partir de las historias de usuario es que obtenemos los criterios de aceptación; los criterios de aceptación son especificaciones del comportamiento deseado o esperado y la funcionalidad del sistema, nos dicen para una historia de usuario, cómo el sistema maneja ciertas condiciones de entrada y cuáles son los tipos de resultados.

Decimos entonces que las historias de usuario son el alcance de un proyecto, nos brindan un panorama general, mientras que las pruebas de aceptación son las especificaciones para dichas historias, ambas se complementan entre sí.

Ahora bien, las pruebas de aceptación tienen una serie de propiedades que mencionaremos a continuación:

- Pertenecen a un cliente => Esto hace que el cliente o experto del negocio participe de manera activa en el proceso.
- Son escritas en conjunto => Esto alienta a la comunicación entre BA, QA, DEV[13] y el cliente.
- Se refieren al QUÉ y no al CÓMO => Recordemos que son una especificación de la historia de usuario, una descripción un poco más detallada, que no debe estar viciada por el cómo llevaremos a cabo la implementación del sistema.
- Escritas en lenguaje del dominio del problema => Esto es para que TODOS puedan entenderlas fácilmente.
- Concisas, precisas y no ambiguas => Esto es para evitar posibles errores en el futuro.

2.2.4 Principios de ATDD

Así como describe Gojko (2) para poder achicar el gap de comunicación, lo que hacemos es juntar a todos los roles (BA, QA, DEV) y a los stakeholders, haciéndolos partícipes e involucrándose desde el comienzo, lo que estamos buscando es que exista una coordinación entre todos ellos, con el mismo fin en común: obtener el producto correcto.

Para ello es que ATDD cuenta con conjunto de principios:

- Utilizar ejemplos. La idea detrás de utilizar ejemplos y en particular ejemplos concretos, es facilitar el común entendimiento y evitar posibles ambigüedades. El ejemplo concreto escrito en un lenguaje coloquial nos permite comunicarnos mejor que si creamos definiciones abstractas.
- Especificar colaborativamente. A partir de los ejemplos y refinándolos es que podemos identificar los criterios de aceptación, para poder hacerlo de manera colaborativa se plantea realizar workshops (“Meet the power of three” / “3 Amigos meeting”[14])
- Automatizar literalmente. Una vez que ya tenemos los criterios de aceptación podemos entonces utilizarlos para crear las pruebas de aceptación automatizadas. La razón de automatizarlas es debido a que requieren un rápido feedback y serán ejecutadas frecuentemente durante el desarrollo. Al realizar la automatización de forma literal nos aseguramos que el entendimiento alcanzado en los workshops no se modifica.
- Probar *hábilmente.* Luego lo que resta es: crear el código que cumpla con las pruebas de aceptación. Para ello es que es de suma importancia: escuchar a las pruebas, ellas nos dicen mucho del código que está siendo generado y refactorizarlas continuamente, para adaptarnos a los cambios que pueden producirse durante el desarrollo.

[13]Se utilizan las abreviaturas a lo largo del presente trabajo. BA= Business Analyst. QA=Quality Assurance. DEV= Developer

[14]También conocida como la reunión de 3 amigos, la describiremos más adelante en el trabajo.

2.2.5 Ventajas de ATDD

Algunas de las ventajas de ATDD son:

- Los ejemplos reales nos ayudan a entender el dominio e identificar correctamente las ***reglas de negocio***.
- Se trabaja de manera más ***colaborativa***, con lo cual las actividades de los distintos roles se encuentran alineadas.
- Existe un mayor ***compromiso*** y una mayor confianza, esto se debe principalmente al hecho de la participación activa de los miembros del equipo desde el comienzo del proyecto.
- Hay un criterio visible para dar por finalizadas las historias de usuario o lo que podemos también llamar: ***criterio de Done.*** Esto nos permite determinar por un lado "*donde estamos parados*" es decir cuánto de la funcionalidad de la historia de usuario cumplimos y además "*cuando parar*" es decir cuándo la historia de usuario la damos por terminada y no necesita nada más.
- Se genera menos ***re-trabajo***. La puesta en práctica de ATDD minimiza los malos entendidos que pudiera haber respecto a lo que se busca del producto de software.
- Brinda una coordinación para proyectos de software que ayuda a realizar entregas al cliente de lo que quiere cuando lo quiere.

2.2.6 Qué puede fallar con ATDD?

Como ya mencionamos antes, ATDD es considerada una mejora de TDD, y como tal asumimos que las ciertas fallas de TDD son evitadas, a pesar de ello cuando implementamos ATDD en un proyecto existen ciertos factores que pueden convertirse en problemas, en lugar de mejorar el proceso. Utilizando como punto de referencia a Ron Quartel (17) se elaboró una agrupación de estos factores en 3 categorías:

- Respecto a las personas.

Las máquinas pueden funcionar, pero el factor humano siempre va a ser el crucial, y aquí sucede lo mismo, la teoría puede ser muy hábil y estratégica pero lo crucial son las personas que la hacen posible. Una de las cosas que tenemos que tener en cuenta es que la personas se involucren en el proceso , y con esto también nos referimos a los Product Owners[15]. Los product owners no son personas que desarrollan código, que entienden de código, sino que son personas de negocio y necesitaremos que ellos tengan un entendimiento fluido con las personas que se encargaran de hacer realidad el producto que buscan. Pero no solo necesitamos que los product owners se involucran, sino que también el BA, DEV, QA deberán ser parte, de otro modo ATDD no resultará efectivo.

[15] "*(..)Es el portador de la voz de los stakeholders (..), quien define que hay que hacer y en qué orden*" http://www.innolution.com/resources/glossary/product-owner

- Respecto de las herramientas.

Muchas veces lo que sucede es que se pierde el foco sobre los criterios y sobre el QUÉ vamos a construir y se pone demasiado énfasis en la herramienta que vamos a utilizar o en cómo se puede integrar con otras. En ese tipo de casos ATDD pasa a ser una herramienta más.

- Respecto de las pruebas.

Las pruebas son el segundo pilar en el cual se sostiene la práctica, sin embargo, hay un par de tips que hay que tener en cuenta para que las mismas no se vuelvan en nuestra contra. Una de las cosas que hay que evitar, es que las pruebas comiencen a ser difíciles de modificar, para ello lo mejor es refactorizar tempranamente y continuamente. Es habitual que a veces ciertas conjunto de pruebas se vuelvan lentas en su ejecución, nuevamente la mejor solución es evitar el problema de antemano y realizar refactoring constante, un refactoring que además tenga en cuenta quitar aquellas pruebas que ya no apliquen o que no sean más necesarias.

2.2.7 Herramientas

Con ATDD hemos descrito que creamos pruebas de aceptación automatizadas y para ello mencionaremos algunas herramientas:

- **Spectacular**[16] .Es una herramienta que permite implementar tanto ATDD como BDD y engloba distintos tipos de pruebas de framework en una sola, introduciendo además el concepto de Caso de uso Ejecutable (Executable Use Cases (EUC))
- **FIT**[17]. Es una herramienta para mejorar la colaboración en el desarrollo de software, permitiéndole a los DEV, QA y BA aprender qué es lo que el software debería hacer y qué es lo que hace.
- **Fitnesse**[18]. Esta herramienta nos permite especificar y verificar los criterios de aceptación de una aplicación, generando un servidor wiki, que sirve a su vez funciona como fuente de documentación.
- **Concordion**[19]. Es una herramienta de código abierto para automatizar especificaciones basada en ejemplos o SPE, que cómo mencionamos antes es otro de los nombres con los cuales se conoce a ATDD.
- **Robot Framework**[20]. Es un framework genérico de pruebas automatizadas y ATDD.

[16] https://code.google.com/archive/p/spectacular/
[17] http://fit.c2.com/
[18] http://www.fitnesse.org/
[19] http://concordion.org/
[20] http://robotframework.org/

2.3 DDD (Domain Driven Design)

"A model is a selectively simplified and consciously structured form of knowledge.", Eric Evans.

2.3.1 Qué es DDD?

Domain Driven Design o su traducción al castellano "*Diseño guiado por el dominio*" es una filosofía de desarrollo que está diseñada para manejar la creación y mantenimiento de software escrito para problemas de dominio complejos. También se define DDD como una colección de patrones, principios y prácticas que pueden aplicarse al diseño de software para manipular la complejidad.

*"DDD es acerca del descubrimiento de lo **QUÉ** es necesario escribir, **PORQUÉ** es necesario escribirlo y **CUÁNTO** esfuerzo hay que utilizar para ello".*(18)

Es importante destacar y diferenciar qué NO es DDD:

- NO es un lenguaje de patrones.
- NO es un framework.
- NO es una "*silver bullet*"[21]
- NO es una filosofía basada en el código.

Los valores en lo que DDD se basa son:

- El dominio central.
- Colaboración y exploración con los expertos del dominio ("*Domain Experts*"[22])
- Experimentación para producir un modelo útil.
- Entendimiento de los contextos que intervienen en el problema complejo del dominio.

[21]En la jerga informática se utiliza este término para hacer referencia a una solución mágica, una solución que por sí misma resuelve todos los problemas.

[22]Son aquellas personas que tienen un conocimiento profundo del negocio del dominio tanto en sus políticas como en sus flujos de trabajo. (18)

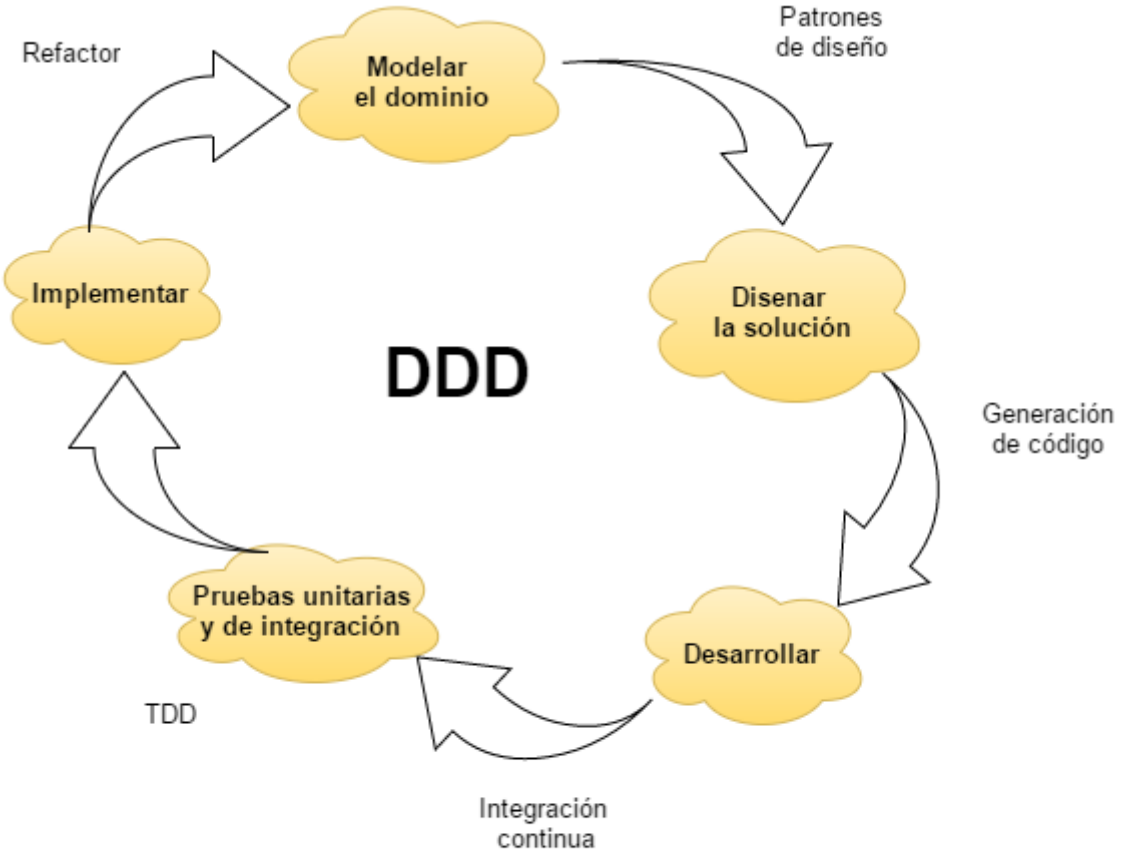


Figura 4. Ciclo Iterativo DDD.

2.3.2 Principios de DDD

Como bien enumera Koskela (17) los principios en los cuales se basa DDD son:

- Foco en el dominio central => Esta es la razón por la cual el producto está siendo construido, es la razón de ser del proyecto que lo desarrollará.
- Aprendizaje a través de la colaboración =>DDD es menos acerca de patrones de diseño y más acerca de resolver problemas a través de la colaboración.
- Creación de modelos a través de la exploración y experimentación. =>La creación de un modelo efectivo es fundamental para DDD, ya que es el artefacto, la pieza principal para resolver el problema complejo del dominio.
- La comunicación => Es la faceta más importante de DDD: la creación de un lenguaje ubicuo ("*Ubiquitous Language*"[23])
- Entendimiento de la aplicabilidad del modelo.
- Evolución constante del modelo.

[23]Es un lenguaje en común entre los expertos del negocio/stakeholders/product owners y el resto de los miembros del equipo (BA, QA, DEV) que provee claridad y consistencia. (18)

2.3.3 Patrones de DDD

DDD parte de la idea de dos retos actuales en los problemas de dominio complejo: por un lado, entender el problema del dominio y por otro lado crear una solución que sea mantenible.

Para ello es que propone los ***patrones estratégicos***:

- Destilar el problema de dominio para revelar qué es lo importante => DDD hace mucho hincapié en la necesidad de focalizar el esfuerzo en el dominio central que es el área que contienen más valor para el éxito de la aplicación.
- Crear un modelo para resolver el problema del dominio.
- Utilizar un lenguaje compartido que permita un modelamiento colaborativo =>Aquí es donde DDD habla de la necesidad de un lenguaje ubicuo de comunicación, este lenguaje tiene como función principal unir el modelo de software con el modelo de análisis. (Ver Figura 5)

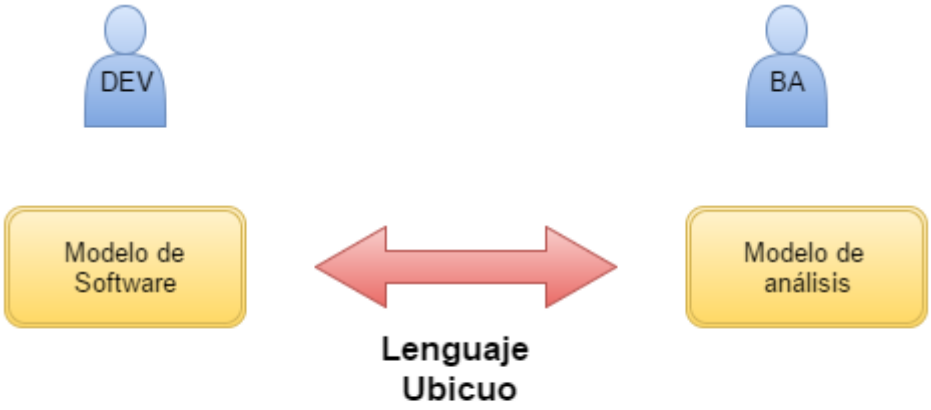


Figura 5. Modelos unidos por el lenguaje ubicuo.

Para poder desarrollar el lenguaje ubicuo que plantea DDD, son necesarias las reuniones conocidas como "*Knowledge crunching*[24]", en ellas se obtienen los casos de uso que serán necesarios para la creación del modelo.
Para que dichas sesiones sean efectivas, existen un conjunto de patrones:

- Poner foco en las conversaciones más interesantes, ya que de ellas es de dónde obtendremos los datos para relevantes para los casos de uso.
- Comenzar con un caso de uso.
- Hacer preguntas poderosas o inteligentes.
- Hacer bocetos en papel.
- Utilizar tarjetas de responsabilidad de clases, es decir que una tarjeta describe las responsabilidades de una clase en particular.

[24]**Knowledge crunching,** es un proceso de modelado efectivo. Se denomina *Knowledge crunching* porque al igual que el procesamiento de números *(Number crunching)* se parte de un conjunto de información no relacionada, a partir de la cual se intenta sintetizar algo. Con el procesamiento de conocimiento queremos hacer lo mismo a partir de un conjunto de información dada, queremos obtener el "valor real" del problema del dominio. Es una tarea que debemos completar para poder entender verdaderamente el problema del dominio.

- Postergar el nombramiento de conceptos para el momento en que se cree el modelo.
- Utilizar BDD.
- Realizar prototipos "*on the fly*".
- Revisar los sistemas en papel, esto es más que nada para aquellos casos en los que no existe un sistema informatizado previo.

2.3.4 Qué ganamos al usar DDD?

A diferencia de aquellas cosas que pueden certificarse de manera fehaciente, no existe una forma de certificar que un equipo aplique DDD, cuándo se aplican sus principios y patrones. Sin embargo, algunos de los resultados que se pueden obtener de su aplicación son:

- Un producto que es fácil de ***entender.***
- Un producto que es fácil de ***mantener***.
- Un producto que ***satisface*** las expectativas de los stakeholders.
- Equipos que se saben ***focalizar*** en el problema del dominio.
- Miembros de equipos que **entienden** mejor el negocio.

2.3.5 Problemas comunes al usar DDD

Debemos recordar que DDD, es una filosofía que parte o surge de la idea de re-alinear el foco de los equipos cuando se trata de resolver problemas para dominios complejos. Al momento de llevar a cabo la implementación de DDD en un proyecto real, es que muchos equipos se enfrentan a los mismos problemas, mencionaremos algunos de ellos a continuación:

- Poner demasiado énfasis en la importancia de los *patrones técnicos*[25]

Esto suele suceder por varios motivos, entre ellos, porque se pone demasiado foco en el código en lugar de poner ese foco en los principios de DDD.

- Olvidar cual es el verdadero valor de DDD.

En DDD contamos con 3 C: Colaboración, Comunicación, Contexto que son las bases de la filosofía. Alrededor de dichas bases pueden generarse inconvenientes: con respecto al contexto produciendo BBofM[26] ,con respecto a la comunicación causando ambigüedad y malas interpretaciones porque se falla en la creación de un lenguaje ubicuo y finalmente con respecto a la colaboración cuando hay diseño mal logrado técnicamente.

- Disponer demasiado tiempo en lo que NO es importante.

Muchos equipos en lugar de focalizar su esfuerzo en entender el verdadero valor del software que se desarrolla, pierden su tiempo en buscar una solución que se integre

[25]Son un conjunto de patrones que DDD provee para la creación del modelo de dominio, en el presente trabajo no haremos mención de ellos.
[26]En ingles *Big Ball of Mud*, acrónimo de *Spaghetti code*, se refiere a un sistema estructurado de manera desorganizada y caprichosa.

correctamente, al perder en énfasis puesto en el para QUÉ, el éxito del proyecto puede irse a pique.

- Hacer los problemas simples complejos

Muy a menudo se cree que DDD puede ser una solución mágica para cualquier tipo de problema, pero recordemos que DDD en su misma definición se dedica al tratamiento de problemas de dominio complejos, es entonces que cuando se intenta aplicar DDD a problemas de dominio triviales se está complejizando un problema que no requiere de DDD en lo absoluto.

- Subestimar el costo de implementar DDD.

Esto puede darse cuando se intenta forzar la aplicación de DDD, por ejemplo, en equipos desmotivados o en proyectos donde el experto del dominio no se involucra, o en proyectos donde no se utiliza una metodología de desarrollo iterativa, siendo todas ellas precondiciones clave para una implementación exitosa de DDD en un proyecto.

2.3.6 Herramientas

Algunas de las herramientas disponibles para DDD son:

- **Actifsource**[27]. Es un plugin para Eclipse que permite combinar el desarrollo del software y DDD; brinda soporte para la creación de múltiples modelos de dominio y también da soporte para la generación de código a partir de templates de código genérico definidos por el usuario.
- **ECO** (Domain Driven Design)[28]. Es un framework para DDD diseñado para incrementar la productividad utilizando las facilidades que brinda el mapeo objeto relacional para persistir objetos de dominio, modelado UML para las clases y máquinas de estado ejecutables.
- **OpenMDX**[29]. Es un framework open source MDA(Mechanics-Dynamics-Aesthetics), que se diferencia de otros frameworks ya que utiliza modelos para directamente manipular en tiempo de ejecución el comportamiento operacional de los sistemas.
- **OpenXava**[30]. Es un marco de trabajo AJAX para desarrollo rápido de aplicaciones web empresariales. Solo hay que escribir las clases del dominio con java para obtener una aplicación.
- **CubicWeb** [31]. Es un framework semántico web con licencia LGPL que ayuda a los desarrolladores a construir aplicaciones a través de la reutilización de componentes, siguiendo los principios del diseño orientado a objetos. Solo definir el modelo de datos es suficiente para obtener una aplicación web funcional.
- **ENode**[32]. Es un framework de C# que brinda soporte para el desarrollo de aplicaciones con DDD, CQRS, EDA y event sourcing.

[27] http://www.actifsource.com/
[28] http://www.new.capableobjects.com/
[29] http://www.openmdx.org/
[30] http://openxava.org/
[31] https://www.cubicweb.org/
[32] https://github.com/tangxuehua/enode

2.4.BDD (Behaviour Driven Development)

“Using examples in conversation to illustrate behaviour”, Liz Keogh.

2.4.1 Qué es BDD?

Existen variadas, complejas, y diferentes definiciones acerca de qué es BDD, en este trabajo usaremos la versión actualizada (ya que ha ido variando a lo largo de los años) de su creador Dan North (19):

> *“BDD es una metodología ágil de **segunda generación**, basada en la extracción, con múltiples interesados, múltiples escalas y altamente automatizada.”* (19)

Su origen resulta ser nada más ni nada menos que la búsqueda de una mejora en la metodología TDD, con lo cual tiene una fuerte influencia de dicha metodología, pero se diferencia también ampliamente como veremos a continuación.

Si bien uno de sus orígenes fue TDD, a lo largo de los años la metodología ha evolucionado y ha ido tomando influencias de otras como son ATDD y DDD (también mencionaremos más adelante como BDD se diferencia de ellas).

Debemos tener presente desde este momento en el trabajo que BDD lo que hace es (19):

> *“Describir un ciclo de interacciones con resultados bien definidos, que resulta en una entrega del software trabajado, testeado, **software that matters**.”*

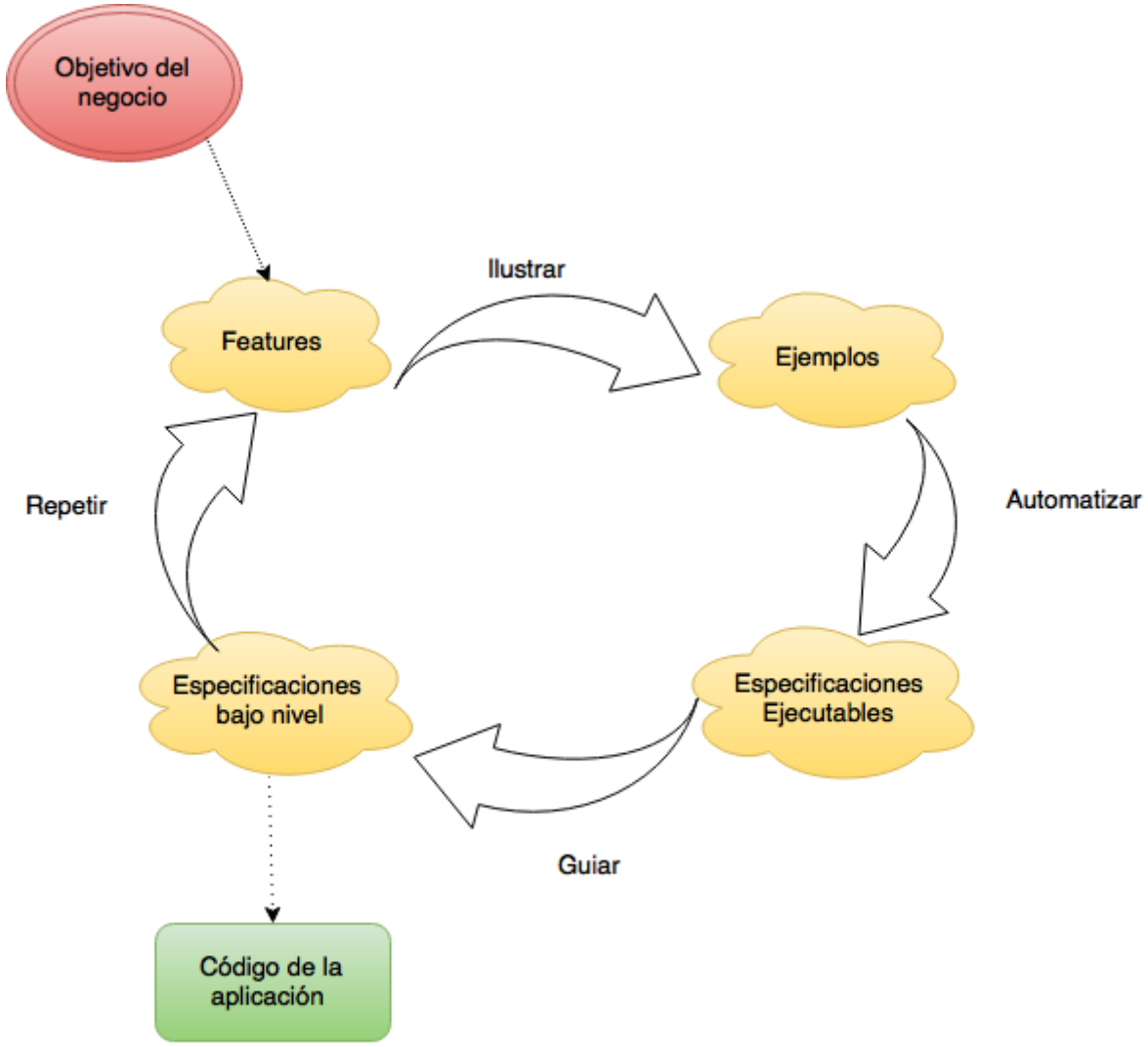


Figura 6. Actividades en BDD

BDD no hace más que hacer un buen uso de muchas prácticas ágiles que ya han sido probadas, a su modo, poniendo todas ellas bajo un fin en común y con una terminología consistente. Lo que busca, es a partir de ejemplos poder definir el comportamiento de la aplicación o sistema que estamos buscando desarrollar, siempre sin perder de vista que lo hacemos es software que importa, software que hace una diferencia, que tiene un propósito de ser, que le importa a quien sea nuestro cliente, en definitiva, ese es el fin primero y último para desarrollar software: **marcar una diferencia.**

2.4.3 BDD vs. TDD

Como mencionamos previamente, BDD no es una nueva metodología desde cero, sino que ha tomado de otras sus buenas prácticas, y la primera de ellas fue TDD. Curiosamente BDD en sus comienzos lo que hizo fue modificar la palabra "*test*" por "*should*", es decir en lugar de utilizar la palabra "*prueba*" usar la palabra "*debería*", con toda la connotación de valor agregado que esta palabra tiene, ya que en lugar de "*probar*" una funcionalidad lo que estamos haciendo es validando lo que la aplicación o sistema "*debería*" hacer. Esta es sin lugar a dudas una fuerte diferencia entre BDD y TDD.

Además, como BDD utiliza las buenas prácticas de TDD, si vemos el gráfico de la Figura 7, veremos que BDD incluye a TDD. Viéndolo desde la perspectiva de que tan abarcativas son ambas metodologías: TDD se centra y está pensada para los desarrolladores, son ellos quien ponen en funcionamiento la máquina iterativa que vimos en la Figura 1. En cambio, BDD se aboca a una tarea mucho más grande: crear software que importa, y para ello tiene en cuenta a todos los roles dentro de un equipo de desarrollo: BA, QA, DEV y además a los stakeholder como participantes activos del proceso.

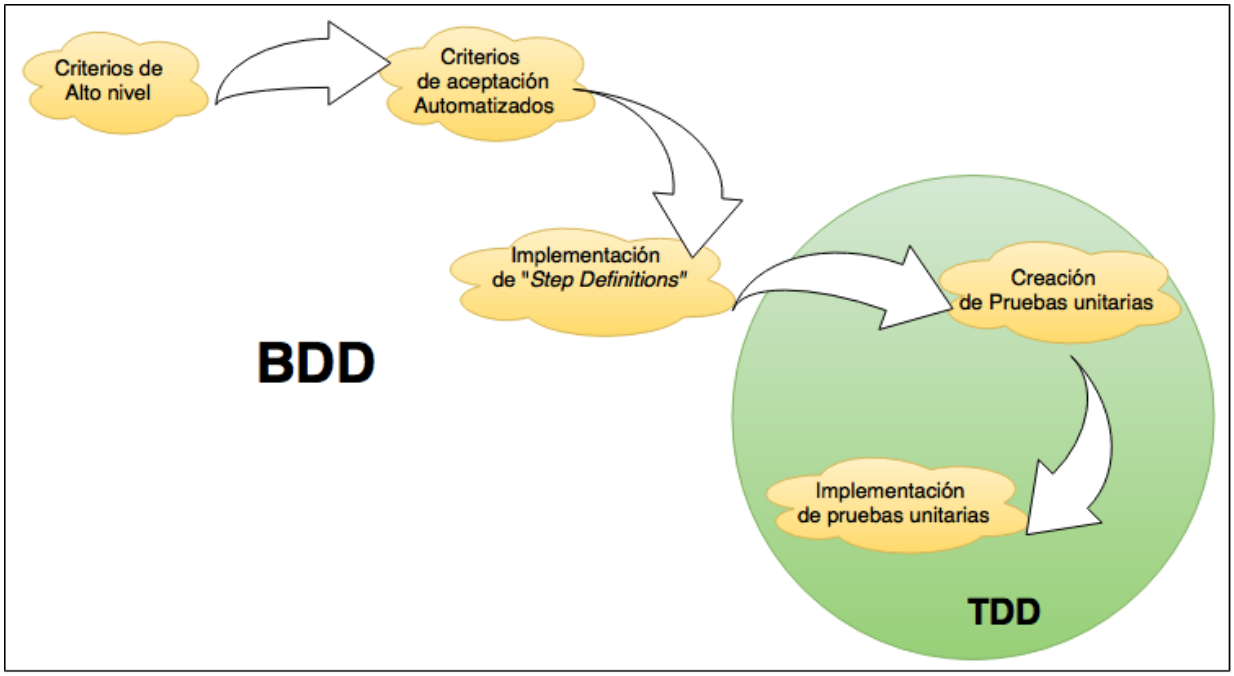


Figura 7. BDD vs TDD.

2.4.3 BDD vs. ATDD

Si bien muchos las consideran prácticas distintas, existen muchos otros que utilizan BDD y ATDD como sinónimos. Al igual que en muchas metodologías y prácticas ágiles, la diferencia entre ellas suele ser sutil o casi no existir, la cuestión es qué palabras se utiliza y cómo se utilizan. Teniendo en que cuenta que BDD surgió contemporánea de ATDD es claro que la influencias son mutuas. Según Liz Keogh (20) había algunos puntos que diferencian a BDD de ATDD:

- BDD cambia el panorama.

Creando ejemplos de cómo se comporta el sistema, y usándolos para definir el alcance, así como también las responsabilidades dentro del sistema, el punto es ir de afuera hacia adentro, es decir de lo más visible que es el comportamiento final hacía lo menos visible que es como llevarlo a cabo.

- BDD fomenta la conversación.

BDD no solo impulsa la creación de un lenguaje de comunicación entre la gente del "*negocio*" y la gente de "*sistemas*", sino que además plantea que dicho lenguaje se base en ejemplos y comportamiento en lugar de criterios de aceptación.

Sin embargo, debido principalmente a la evolución de ambas, la diferencia hoy en día radicaría simplemente en el nombre. Para los fines de este trabajo mencionaremos también que BDD pone foco más en el aspecto de **comportamiento** de un sistema a diferencia de ATDD que utiliza los criterios de aceptación para guiar el desarrollo. Además, BDD tiene una visión orientada al cliente mientras ATDD tiene una visión más orientada al desarrollador, más cercana a la visión de TDD.

2.4.4 BDD vs DDD

Tal como venimos hablando al comienzo de esta sección, esta metodología nació como un planteo de modificar TDD y a su vez se fue nutriendo de otras prácticas relacionadas y contemporáneas. Fue tomando de cada una de ellas algunos términos, fue modificando otros, recreando y evolucionando a lo largo de los años.

Una de las grandes influencias es DDD (la otra gran influencia principal es TDD). Si bien DDD también hace un fuerte hincapié en la comunicación, lo hace desde un ángulo distinto, de modo que podemos decir que DDD hace foco en el *modelo del dominio* (además no nos olvidemos que DDD es una filosofía de desarrollo para problemas de dominio complejos) como corazón del software que se utiliza para satisfacer un determinado comportamiento. En tanto, si hablamos de BDD hace foco en el *comportamiento del software* y en cómo el sistema se debería comportar; BDD pretende reducir la brecha de comunicación que existe comúnmente entre los usuarios de negocio (o *stakeholder*) y los miembros del equipo de

desarrollo (QA, BA, DEV) la idea es poder minimizar los obstáculos entre especificación, diseño, implementación y confirmación del comportamiento de un sistema.
Por estos motivos es que BDD al igual que DDD plantea la idea de un lenguaje ubicuo de comunicación basado más que nada en "**obtener la palabra indicada**".

Un detalle más evidente que diferencia a BDD de DDD, es que en el primero no estamos definiendo un modelo de dominio y que además BDD puede aplicarse a cualquier tipo de proyecto, no necesariamente a un proyecto que intente resolver un problema de dominio complejo.

2.4.5 Los pilares y principios de BDD

BDD pretende ayudar a poner foco en la entrega de valor de negocio, que ha sido priorizado y es verificable a través de la utilización de un vocabulario en común (el lenguaje ubicuo al igual que DDD).

Para alcanzar tal objetivo es que se construye a partir de los siguientes pilares (19):

- El análisis por anticipado, diseño y planeamiento, todo posee una utilidad decreciente. ("*Enough is enough*").

Con este primer pilar nos estamos refiriendo en cierta manera al viejo método de "*cascada*" en el cual TODO el relevamiento de requerimientos se hacía dentro de una primera etapa y no se podía modificar durante el desarrollo de las siguientes etapas, podríamos decir estaba *"escrito en piedra"*. Esta idea de no realizar demasiado o en exceso detallado relevamiento, planeamiento por adelantado nos habla de lo mismo; tal como muchas metodologías ágiles proponen: minimizar el tiempo y el alcance de los supuestos, y en cambio buscar una manera de verificar o refutar los supuestos tan pronto como sea posible, limitándonos al alcance actual, esto nos permite realizar estimaciones más certeras y en base a las condiciones actuales.

- Tanto el "*Negocio*" como los "*técnicos*" deben referirse al mismo sistema de la misma manera. (*"Deliver stakeholder value*").

Dicho de otra manera, todo es acerca de lo que el sistema "*debería*" estar haciendo. BDD es, tal como mencionamos antes acerca de "*obtener la palabra indicada*" y esto se logra utilizando un lenguaje consistente de comunicación de forma tal que las diferencias entre la gente de negocio y los técnicos (que en este caso incluiría también a los BA y QA, no solo a los DEVs) tiende a desaparecer. Definir un sistema en términos de su comportamiento, sirve porque posee distinta importancia en cada nivel de abstracción, sirve porque es lo que tiene en común el sistema que todos los participantes entienden de la misma manera.

- Todo sistema debería tener un valor de negocio identificable y verificable. ("*It's all behaviour*").

Cada comportamiento de un sistema, debiera estar allí, porque agrega concretamente un valor de negocio al sistema como un todo, es decir que es parte del comportamiento del sistema como un "*todo*" y tiene un motivo de "*ser*" dentro del mismo.

La idea es evitar incorporar nuevas funcionalidades o características a un sistema por el simple motivo que esté "*de moda*" o la competencia las brinden cuando en realidad dichas características no brindan al negocio un valor. Es por ello que en BDD es de suma importancia el valor asociado a cada historia de usuario, ya que esta información puede ser utilizada en la toma de decisiones de planeamiento y para guiar a todos hacia un buen entendimiento de los costos/beneficios.

Además de los pilares en los que se establece BDD, podemos mencionar adicionalmente algunos principios que surgen a partir de dichos pilares (4):

- Trabajar de manera colaborativa.

BDD es una práctica colaborativa tanto para con los miembros del equipo y el cliente, como para con los miembros del equipo internamente.

- Aceptar y trabajar la incertidumbre.

Es una realidad que los equipos que trabajan siguiendo las prácticas de BDD, saben que no pueden adelantarse a todos los imprevistos que conlleva un proyecto, por ello se ayudan con métodos tales como el feedback continuo del trabajo, que les indica que tan encaminados se encuentran en la solución y además brindan una herramienta para adelantarse a posibles inconvenientes.

- Utilizar ejemplos.

Los ejemplos son la herramienta más importante en BDD y cumplen un rol principal dentro de la metodología, ya que son una forma efectiva de comunicar los requerimientos de manera precisa, clara y sin ambigüedades. En BDD se escriben en el lenguaje ubicuo que mencionamos y que se conoce como Gherkin (Ver figura 8).

- Escribir especificaciones ejecutables y no pruebas automatizadas.

Esta diferencia es vital para el desarrollo de BDD como práctica en un equipo, si bien parece a primera impresión que ambas cosas son lo mismo o tienen el mismo valor, esto no es así. Pruebas automatizadas puede ser cualquier tipo de prueba que se ejecuta de forma automatizada utilizando alguna herramienta. En cambio, especificaciones ejecutables son pruebas automatizadas que ilustran y verifican como la aplicación entrega un requerimiento específico de negocio, involucran tanto la validación como la comunicación, ya que el propósito es que los reportes que se generen como resultado de la ejecución deben poder ser fácilmente entendibles por todos los involucrados en el proyecto.

- Escribir especificaciones de bajo nivel y no pruebas unitarias.

BDD no solo se trata de definir especificaciones ejecutables sino también de contar con calidad de código, es por ello que hablamos de especificaciones de bajo nivel. En este punto es donde entra en juego la integración con TDD, aunque con una pequeña variación, el DEV no piensa en término de pruebas unitarias, sino que escribe especificaciones

técnicas que describen como la aplicación se tiene que comportar; como dado un valor de entrada tiene que producir un valor determinado de salida.

- Entregar documentación "*viva*".

Los reportes que se obtienen de la ejecución de las especificaciones, no son simplemente reportes técnicos, ya que como mencionamos anteriormente estos reportes deben ser entendibles por todos, y convertirse en parte de la documentación. Decimos que esta documentación es "*viva*" porque se actualiza luego de cada ejecución, que al ser automatizada se ejecuta de manera continua y constante en el tiempo.

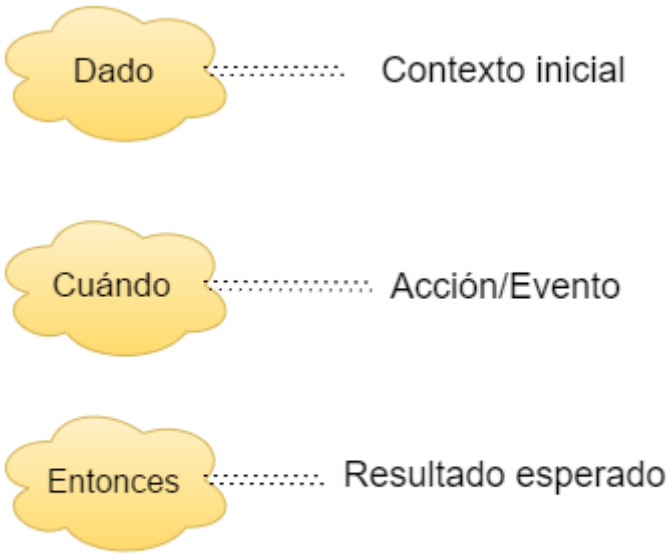


Figura 8. Gherkin Básico

2.4.6 Beneficios de BDD

Siendo BDD una metodología que toma buenas prácticas ya probadas provenientes de otras metodologías conocidas, es de asumir que los beneficios de utilizarla son muchos y variados. Aquí hay que tener en cuenta que, si bien es una metodología con sus principios definidos, todo lo que plantea BDD al igual que otras metodologías ágiles, son guías prácticas, buenas prácticas para equipos. Dependerá entonces de cada equipo, sus características y su puesta en práctica de las guías planteadas, los beneficios específicos que el equipo tendrá, que además dependerá también de las necesidades que tenga ese equipo.

A continuación, mencionaremos los principales beneficios que surgen de implementar BDD (cualquiera que se sea su puesta en práctica) en un proyecto de desarrollo de software (4):

- Reducción de costos.

Con este punto hacemos referencia tanto a los costos monetarios para el negocio que se reducen gracias a que existe un mayor foco en lo que el negocio necesita y espera del producto. Al comprender la visión y el objetivo del negocio, no se desperdicia tiempo (se prioriza de manera efectiva) en implementar cosas que no tengan un valor para el negocio y que le permitan alcanzar el objetivo. Pero también hacemos referencia a los costos en cuanto a una reducción en el porcentaje de errores y defectos que se producen en cada iteración de implementación, esto se debe principalmente a que todos los miembros del equipo de desarrollo saben desde el comienzo como es que el sistema debe *comportarse*.

- Los cambios son más fáciles y seguros de realizar

Gracias a que existe *feedback* constante, que permite llevar un seguimiento del estado del proyecto, los cambios cuando se producen, es en un momento en que realizarlo no es costoso, por dos motivos: no es un estado avanzado en el desarrollo y la implementación es lo suficiente flexible para adaptarse rápidamente a los cambios, como contamos con especificaciones ejecutables y especificaciones de bajo nivel, podemos inmediatamente luego de implementado un cambio volver a ejecutar todo y asegurarnos que el cambio no introdujo errores y se implementó con éxito.

- Releases más rápidos

Tal vez sea para los clientes el mayor aspecto positivo de BDD: tener rápidamente un requerimiento en funcionamiento. El hecho que los releases sean más rápidos se deben principalmente a las pruebas automatizadas que permiten acelerar el ciclo de los releases y también al hecho de saber y comprender qué es lo que se espera que el sistema haga.

2.4.7 Limitaciones/restricciones de BDD

Si bien hasta el momento no hemos hecho más que remarcar y resaltar todas las virtudes de BDD, esta metodología tiene sus limitaciones o mejor dicho restricciones que son necesarias para una buena puesta en práctica (4):

- BDD requiere una alto compromiso y colaboración por parte del negocio.

Uno de los puntos más importantes en BDD es la comunicación entre el equipo de desarrollo y el negocio, que está dado gracias a la utilización de un lenguaje ubicuo de comunicación. Pero más allá de ello, se requiere que el negocio (los expertos del dominio, los interesados y stakeholders) se comprometan a formar parte del proceso, formar parte de manera activa y colaborativa, sí esto último no es posible, ya sea porque esperan a último momento para hacer un feedback o porque no se encuentran disponibles para realizar workshops o reuniones de especificación, entonces será difícil poder obtener de BDD los beneficios esperados. Cabe destacar también que el compromiso y colaboración tiene que también darse dentro de los miembros del equipo, ya que en caso de no ser así también dificulta la puesta en práctica de BDD.

- BDD funciona mejor en un contexto Agile.

Si bien BDD se puede poner en práctica no sólo en proyectos ágiles, el marco que brindan Scrum[33], Kanban[34] u otra metodología iterativa de trabajo para la gestión de proyectos, es un marco lo suficientemente sólido para dar soporte al desarrollo e implementación de BDD.

- El costo de mantenimiento de las pruebas puede ser muy alto sí los mismos son escritos pobremente.

La creación de pruebas automatizadas de aceptación requiere de ciertas habilidades deseables por parte del equipo de desarrollo. Sí las pruebas no son correctamente y cuidadosamente diseñadas haciendo buen uso de las capas de abstracción y la expresividad, a lo largo del tiempo lo que producen es que el costo de mantenimiento crezca de manera exponencial. Si bien en BDD el foco es la utilización de ejemplos para definir el comportamiento del sistema y obtener una buena comunicación entre las partes involucradas en la creación del mismo, no debemos perder de vista que las pruebas automatizadas son una pieza muy importante para alcanzar el objetivo de negocio.

[33] El **Scrum** es un proceso de la Metodología Ágil que se usa para minimizar los riesgos durante la realización de un proyecto, pero de manera colaborativa.

[34] **Kanban** es una metodología ágil, cuyo objetivo es gestionar de manera general cómo se van completando las tareas. Se destaca por ser una técnica de gestión de las tareas muy visual, que permite ver a golpe de vista el estado de los proyectos, así como también pautar el desarrollo del trabajo de manera efectiva

2.4.7 Actividades principales de BDD

BDD posee un conjunto de actividades que pasaremos a explicar más en detalle. Tomando como referencia la Figura 6, las actividades de BDD son (4):

- Identificar y entender los objetivos del negocio.

Sí vemos en la Figura 6, los objetivos del negocio son el punto de inicio y principal input para dar comienzo a BDD. Para ello BDD hace buen uso de prácticas de relevamiento de requerimientos como: *feature injection*[35], *impact mapping* [36], *Purpose-Based Alignment Model*[37]. Lo que nos interesa en esta primera etapa es conocer y definir de manera clara: la visión (que es lo queremos lograr), los objetivos (como lo que queremos lograr beneficia al negocio), los stakeholder (a quienes beneficia), las capacidades (que es lo que necesitamos construir).

- Definir e ilustrar las features.

Una feature es un trozo de funcionalidad de software que ayuda a los usuarios u otros *stakeholder* a alcanzar un objetivo de negocio, para satisfacer una capacidad que el sistema debe cumplir. Una *feature* no es necesariamente una historia de usuario, sino que muchas veces la descripción de la feature está dada por un conjunto de historias de usuario (Ver figura 9). Para ilustrar las features nos hacemos uso de los ejemplos, que pasarán a ser descriptos como escenarios utilizando *Gherkin*.

- Crear especificaciones ejecutables.

Una vez que ya tenemos definidos los escenarios, entonces a partir de ellos definimos los "*steps definition*"[38] que pasarán a ser especificaciones ejecutables. Los step definitions son los intermediarios entre los escenarios que se encuentran definidos en alto nivel y las especificaciones de bajo nivel, son el nexo que conectan los mismos. Estas especificaciones ejecutables son automatizadas, ya que de esta manera nos permiten tener un *feedback rápido* y constante.

[35]**Feature injection** es una técnica que intenta identificar el valor que el proyecto intenta entregar y las features capaces de entregar dicho valor. El objetivo es complementar el conjunto de features que proveerán el mayor beneficio en pos de alcanzar los objetivos del negocio.
[36]**Impact Mapping** es una práctica que permite visualizar las relaciones entre los objetivos del negocio detrás del proyecto, los actores que se verán afectados y las features que permitirán al proyecto entregar los resultados esperados.
[37]**Purpose-Based Alignment Mode**l es otra técnica grafica que permite definir cuánto esfuerzo hay que invertir para cada feature. Los modelos hacen más sencillas las discusiones sobre los presupuestos que sustentan el valor propuesto para cada feature, y previenen que features de bajo valor pasen inadvertidas.
[38]**Step Definition** son trozos de código que interpretan el texto de la definición del escenario y que saben que especificación de bajo nivel llamar .

- Crear especificaciones de bajo nivel.

Las especificaciones ejecutables si bien se encuentran automatizadas y escritas en algún lenguaje de programación utilizando alguna herramienta de BDD, requieren de un paso más para poder conectarse con la aplicación. Las especificaciones de bajo nivel (en donde mencionamos se aplica TDD) son el último eslabón y más técnico de todos para llegar al sistema. En este punto se define el "*Cómo*" en lugar de el "*Que*" que definen las especificaciones ejecutables de alto nivel.

En la Figura 10 podemos ver como cada una de estas actividades están relacionadas mientras que en la Figura 11 vemos como se relaciones con las distintas capas de abstracción.

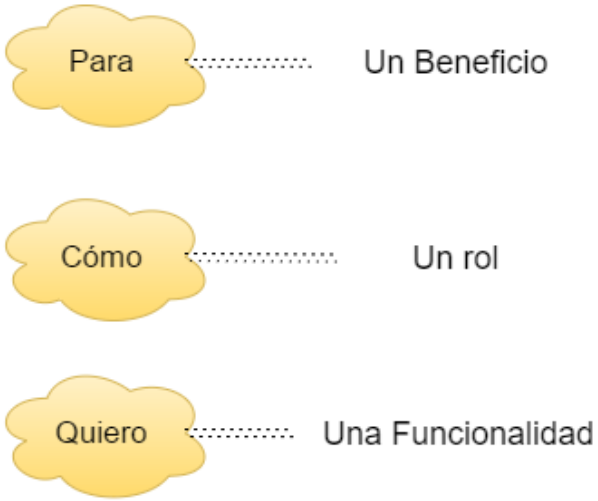


Figura 9. Historia de usuario en BDD

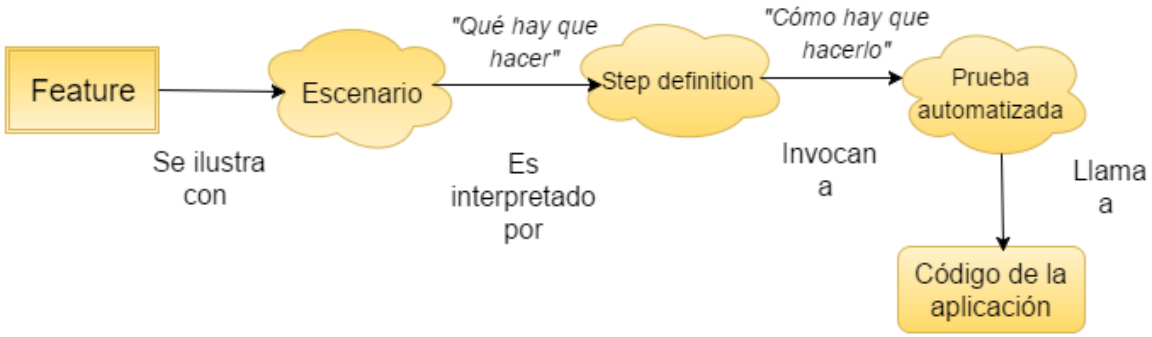


Figura 10. Actividades en BDD y sus relaciones

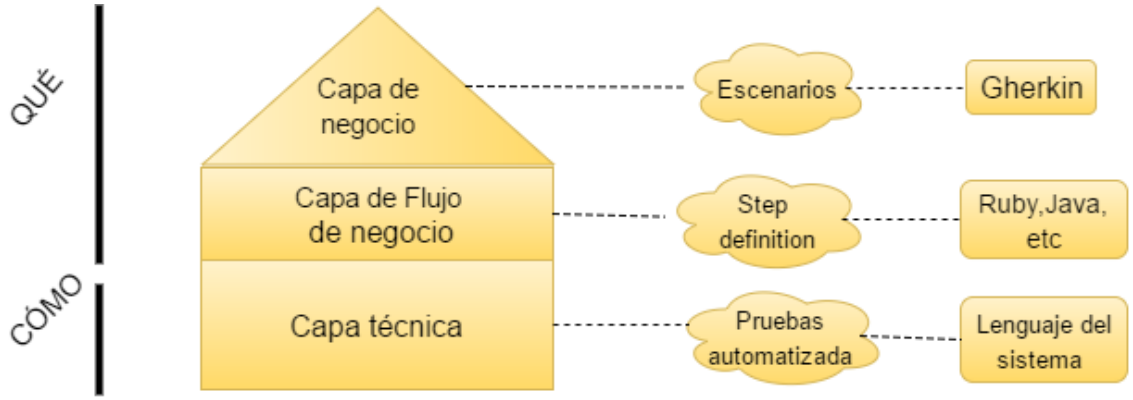


Figura 11. Capas de abstracción, el QUÉ y el CÓMO.

2.4.8 Herramientas

A continuación, enumeramos herramientas BDD disponibles en el mercado:

- **Serenity BDD** [39]. Es una librería de código abierto que permite escribir pruebas de aceptación automatizadas de alto nivel y en menos tiempo. Permite la creación de pruebas de forma tal que sea fácil y flexible su mantenimiento. Además, provee reportes ilustrados, permite mapear las pruebas con los requerimientos o capacidades, entre otras funcionalidades.
- **Cucumber**[40] Es una herramienta de código abierto para especificaciones ejecutables, que fue diseñada específicamente para asegurar que las pruebas de aceptación son fáciles de leer por cualquier miembro del equipo. Actualmente brinda soporte para múltiples lenguajes de programación como son: Ruby, Java, .net, php, etc.
- **RSpec**[41] Es un framework BDD para el lenguaje Ruby que está inspirado en JBehave. Está compuesto por múltiples librerías que fueron diseñadas para trabajar en conjunto o que pueden utilizarse de manera independiente con otras herramientas como Cucumber.
- **SpecFlow**[42] Es un framework BDD de código abierto para .Net, así como también Cucumber utiliza el lenguaje Gherkin y provee integración para Silverlight, Window Phone y Mono.
- **JBehave**[43] Es un framework BDD para el lenguaje Java (creado por Dan North) Actualmente provee también un plumbagina para Eclipse, como así también integración con Ant y Maven.

39 http://www.thucydides.info/#/
40 https://cucumber.io/
41 http://rspec.info/
42 http://www.specflow.org/
43 http://jbehave.org/

2.5 Resumen del capítulo

En este capítulo hemos hecho un relevamiento del estado del arte de las distintas metodologías que han influenciado a BDD, partiendo de TDD, pasando por ATDD y mencionando también DDD. De todas ellas se describen básicamente 3 puntos:

- descripción de la metodología,
- beneficios y desventajas,
- herramientas disponibles.

Luego nos detuvimos en la descripción de BDD, haciendo hincapié en cómo se diferencia con respecto a las metodologías descritas anteriormente.

Capítulo 3. Desarrollo de guía práctica propuesta

Mediante el análisis de las metodologías descritas en el capítulo anterior, se desarrolla una propuesta de guía práctica para incorporar el uso de BDD en equipos de desarrollo de software.

Describiremos la guía práctica que es el foco central de la tesis, partiendo de un cuestionario inicial, cuya finalidad es determinar la conveniencia o no de aplicar la guía. Luego se describen cada una de las fases de implementación planteadas.

3.1 Cuestionario Previo

Antes de poner en práctica una nueva metodología la primera pregunta que debería surgir es: *¿es esta metodología la apropiada para mi equipo?*
Lo que sucede muy a menudo es que tanto equipos grandes que pertenecen a multinacionales como equipos de pequeñas pymes todos ellos quieren mejorar su forma de trabajo, mejorar la entrega al cliente, entenderse mejor y evitar malos entendidos y en pos de dicho afán, muchos de estos equipos caen en el mal uso de metodologías simplemente por el hecho de considerarlas "*soluciones mágicas*". Otra cosa que también sucede es que se dice: "*a todos los equipos que la aplicaron les funciona bien*", esta afirmación en particular es bastante peligrosa, ya que es demasiado genérica y además el simple hecho de que a otros equipos les haya sido beneficioso el uso de tal o cual práctica, no quiere decir para nada que la misma sea aplicable al equipo actual.

Esto hace que debamos evaluar si nuestro equipo cumple con las condiciones necesarias para aplicar BDD, así como también detectar cuales son los puntos en los cuales habrá que hacer foco en la incorporación de la metodología. Recordemos que BDD es un conjunto de prácticas y que no es algo inflexible que no puede cambiarse, todo lo contrario, cada equipo debe decir cuál es la mejor manera de aplicar las prácticas.

Teniendo esto en cuenta a continuación se describe un cuestionario que nos ayudará a determinar si la incorporación de BDD es recomendable o no para un equipo en particular. La idea es que una vez obtenidas las respuestas y contrastadas las mismas con las respuestas planteadas, se pueda decidir si continuar o no con la fase 1. En otras palabras, este cuestionario cumple la función de "*Filtro inicial*".

3.1.1 Cuestionario

1. ¿Es un proyecto nuevo? ¿Es un proyecto a largo plazo? (Mayor a 6 meses)
2. ¿Es un equipo nuevo? ¿Es grande? (Mayor a 10 personas) Está distribuido?
3. ¿Porque queremos usar BDD?
4. ¿En caso que los haya, cuales son los problemas actuales?
5. ¿Se utiliza Scrum, Kanban o alguna metodología ágil? ¿Se utilizó alguna en el pasado?
6. ¿El cliente participa del desarrollo del producto? ¿Estaría dispuesto a hacerlo?
7. ¿Se utiliza automatización en algún nivel? ¿Se cuenta con experiencia en automatización?
8. ¿Se utiliza alguna herramienta de documentación?
   ¿En caso que se utilice, todo el equipo hace uso de la misma?

3.1.2 Análisis de las respuestas

Las respuestas y análisis a continuación se deben tomar como una guía para determinar si continuar o no, una guía para evaluar y analizar la situación actual del proyecto en el cual se busca incorporar BDD; se debe evitar tomar dichas consideraciones como estrictas.

La pregunta del punto 1 nos hace reflexionar acerca del **proyecto**, si el proyecto es nuevo entonces es un empezar desde cero, lo cual es ideal y es el mejor caso de todos para la puesta en práctica de la metodología. De hecho, BDD está planteado idealmente para empezar un proyecto, ya que esto nos permite seleccionar desde el principio las pautas y prácticas que se tendrán en cuenta a lo largo del desarrollo del producto de software. La pregunta acerca de la duración del proyecto tiene más que ver con la evaluación de costo/beneficio. Si es la primera vez que vamos a utilizar la metodología (con lo cual existe una cuota de desconocimiento e incertidumbre bastante alta) y además el proyecto es corto, quizás no sea tan buena idea pensar en incorporar una nueva forma de trabajo, ya que es probable que los resultados obtenidos no sean lo suficientemente demostrativos.

La pregunta del punto 2 nos adentra en el **equipo**, si el equipo es nuevo hay que tener en cuenta dos cosas: por un lado, un equipo nuevo tiene como desafío conocerse y aprender a trabajar juntos, y por otro lado si el equipo ya viene trabajando junto es difícil cambiar formas de trabajo ya adoptadas. El hecho de que el equipo cuente con varios miembros también es relevante, es mucho más fácil llegar a un acuerdo entre 3 personas que llegar a un acuerdo en una reunión entre 15 personas. Sí además los miembros del equipo se encuentran distribuidos también hay que tener en cuenta qué medio utilizarán para comunicarse si es que ya no cuentan con alguno.

¿La pregunta del punto 3 nos hace cuestionar el motivo, es decir existe algún motivo particular por el cual estemos buscando incorporar BDD como metodología de trabajo? Si la respuesta es no, entonces no hay porque seguir. ¿Es el cliente quien solicitó que se implemente BDD? Si es ese el caso, entonces lo que debemos hacer es analizar si es conveniente o no para nuestro proyecto y en caso de no serlo informar al cliente con los motivos por los cuales no es conveniente. Si en cambio, tenemos motivos bien definidos por los cuales queremos incorporar BDD vamos por buen camino.

La pregunta del punto 4 profundiza un poco más acerca de los motivos, si tenemos **problemas** detectados que queremos mejorar y que pensamos que BDD nos puede ayudar a corregirlos, entonces debemos tomar esos problemas para poner especial foco en su solución a largo plazo. Alguno de los problemas que suele surgir y que son motivo válido para pensar en BDD como una buena forma de solucionarlos son: un creciente número de defectos encontrados constantemente, malos entendidos entre el cliente y los desarrolladores, etc.

La pregunta del punto 5 nos acerca un poco a las **metodologías ágiles**, si en el pasado el equipo (suponiendo que es no es un equipo nuevo) ya trabajó o tienen experiencia en alguna de esas metodologías esto es un punto a favor en cuanto a la curva de aprendizaje, ya que es probable que haya ciertas prácticas que el equipo ya conozca. Por otro lado si ningún miembro del equipo conoce ni tampoco utilizó antes metodologías ágiles, hay que tener en cuenta dos cosas: por un lado que el cambio de paradigma de metodología con respecto a *Waterfall* o *RUP*[44] es bastante considerable, y por otro lado que no existen preconceptos por parte de los miembros del equipo acerca de cómo implementar las prácticas.

La pregunta del punto 6 nos lleva a pensar en el otro ingrediente importante de nuestra receta: el **cliente**. Si el cliente viene participando activamente de otros proyectos o del proyecto actual (suponiendo que es un proyecto ya activo y no uno nuevo) este elemento es sumamente importante para BDD. Si en cambio aún no participa, pero es posible plantearlo y se encuentra dispuesto a participar en el futuro, es también una buena clave para continuar. Pero si el cliente, no participa actualmente y tampoco está dispuesto a hacerlo en el futuro entonces es probable que no alcancemos todos los beneficios esperados.

La pregunta del punto 7 nos habla de **automatización**, cuando hablamos de BDD se asume que un requisito necesario y obligatorio es la utilización de automatización, sin embargo, esto no es del todo, así como veremos más adelante en este capítulo. A pesar de ello sí ya existe algo automatizado porque el proyecto no es nuevo o se cuenta con la experiencia de algún miembro del equipo en este campo, es un buen punto que podemos usar a nuestro favor. Si por otro lado no existe nada automatizado o conocimiento casi nulo en cuanto a la automatización, debemos tenerlo en cuenta a futuro.

La pregunta del punto 8 nos menciona la **documentación**, si bien es sabido que muy a menudo los proyectos poseen documentación escasa, también se sabe que la documentación es un pilar importante con el cual debemos contar durante el desarrollo de un proyecto. Si bien contar con una herramienta en particular no es indispensable, hoy en día es bastante importante sobre todo para los casos de equipos distribuidos, donde uno de los puntos de comunicación está dado por el lugar donde se encuentra la documentación compartida.

---

[44] *"(..) es un proceso de desarrollo de software y junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos"* http://proceso-unificado-racional.blogspot.com.ar/

3.2 Introducción a las fases de la guía

La guía establece una serie de pautas a mejorar, recomendaciones para incorporar técnicas y prácticas y se encuentra divida en fases:

1. Comunicación.
2. Automatización.
3. Documentación.

El primer aspecto a tener en cuenta y al cual BDD le da vital importancia, es la **comunicación**, recordemos que es el fin primero y último de BDD achicar la brecha de comunicación que existe entre los desarrolladores y la gente de negocio. Lo que queremos ante todo es disminuir la cantidad de malos entendidos y malas interpretaciones que puedan surgir. Por ello en la primera fase nos vamos a focalizar en mejorar este aspecto, profundizando en cómo podemos mejorar dicha interacción de manera eficiente y prolongada en el tiempo.

El segundo aspecto es la **automatización**, si bien lo segundo importante en BDD son las pruebas automatizadas, esto no quiere decir que nos veamos obligados a automatizar absolutamente todo en post de creer que esa es la mejor manera de implementar BDD. Por el contrario, la dosis justa de automatización es la que nos brinda mayores beneficios, veremos entonces qué es razonable automatizar y qué no y además cómo hacerlo.

Por último, abordaremos la **documentación**, es muy común en metodologías como *Waterfall* que la misma se encuentre definida antes del comienzo del proyecto, es decir antes de que se empiece a implementar la solución. En BDD como mencionamos en el capítulo anterior la documentación esta "*viva*" esto quiere decir por un lado que se va generando al mismo tiempo que se implementan las funcionalidades, pero también implica que se va actualizando cada vez que una prueba es ejecutada, con lo cual la información de reportes no solo es creada bajo demanda, sino que también es actualizada de manera constante.

Un equipo que desee incorporar BDD comenzará con la fase de comunicación, para luego focalizarse en la fase de automatización y finalmente pasar a la documentación. Es posible que dadas las condiciones particulares de algunos equipos sucedan dos cosas: algunas prácticas recomendadas ya estén siendo aplicadas o que algunas prácticas no sean realizables de la manera en la cual se plantean en el presente trabajo; cualquiera sea el motivo dependerá del equipo la forma de adaptarla para lograr el objetivo deseado.

3.3 Fase 1 Comunicación

"*The most important thing in communication, is hearing what isn't said.*", Peter Drucker.

3.3.1 Entendiendo el valor del negocio

Objetivos SMART

Para poder embarcarnos en el camino hacia BDD, debemos empezar identificando y entendiendo cuáles son los objetivos que persigue el cliente. En general el cliente tiene definidos tanto los objetivos como la visión a nivel empresarial. Sin embargo, nuestro interés radica en la definición de los objetivos del proyecto en particular en el cual trabajaremos. Es necesario que todos los miembros del equipo tengan pleno conocimiento de estos objetivos y sean capaces de asimilarlos, esto dará una noción más completa de lo que se espera obtener.

Los objetivos son ideales que sean SMART (4):

- Específicos ("*Specific*").

Antes que nada, un objetivo tiene que decir a los lectores específicamente **qué** es lo que se está intentando hacer, describiendo también **porque** se quiere hacer.

- Medibles ("*Measurable*").

Un buen objetivo también debe ser medible, ya que esto nos dará una clara idea de lo que espera el negocio y nos ayudará a determinar si se alcanzó a satisfacer dichas **expectativas**.

- Alcanzables ("*Achievable*").

Para conocer si un objetivo es alcanzable o no es importante cuantificar, de manera tal que nos ayude a determinar qué estrategia es la mejor para resolver el problema y sí es posible alcanzarlo. Debemos poder conocer **qué** es lo que queremos hacer.

- Relevantes ("*Relevant*").

Quizás el punto más importante de un objetivo es saber si agrega una contribución positiva a la organización. Los objetivos relevantes son los que hacen una **diferencia** al negocio.

- Limitados en tiempo ("*Time-bound*").

Debe existir un **plazo definido** en el cual el objetivo se estima será alcanzado.

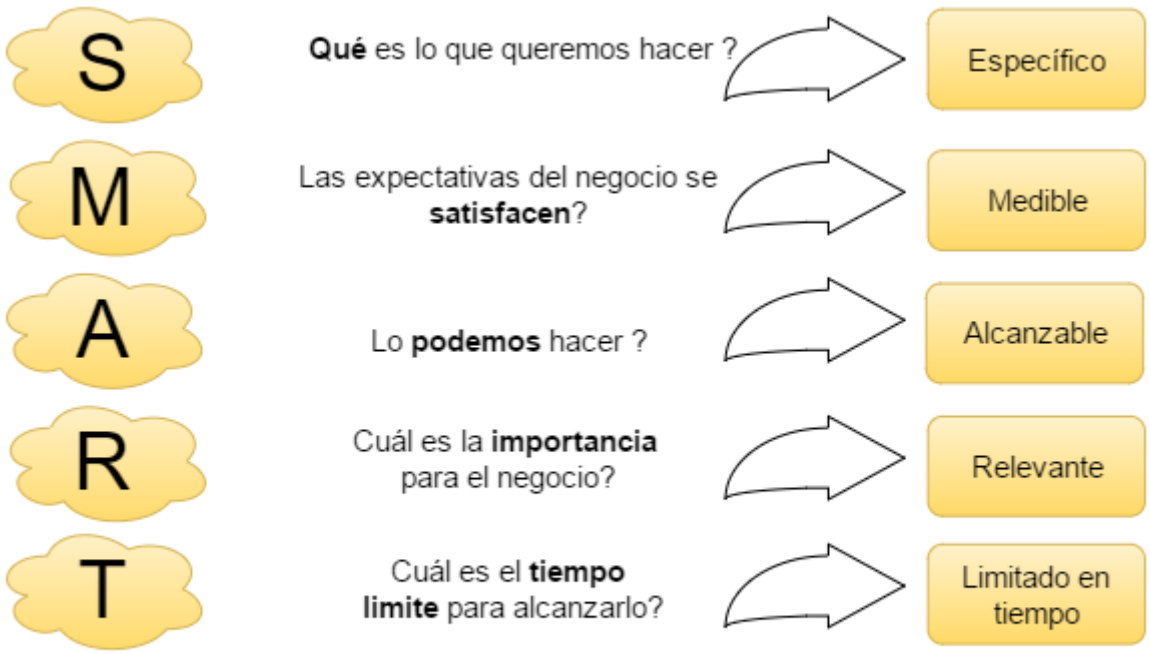


Figura 12. Objetivos SMART

Impact Mapping

"Impact Mapping es una técnica visual de planeamiento estratégico que ayuda a los equipos a alinear sus actividades con los objetivos del negocio y tomar mejores decisiones". (25)

Decimos que es una forma de visualizar la relación entre los **objetivos** detrás de un proyecto, los **actores** que se verán afectados por el proyecto y los **entregables** que permitirán al proyecto obtener los resultados esperados. (4)

Los "*Mapas de impacto*" son muy simples de crear y de comprender, y puede convertirse en una herramienta muy útil para documentar el proceso de análisis de requerimientos y obtener una visión de alto nivel acerca de lo que se está buscando lograr.

Los "*Mapas de impacto*" se generan a partir de conversaciones entre los miembros del equipo y los expertos del negocio o stakeholders y están basadas en 4 tipos de preguntas:

- ¿Porqué?
- ¿Quién?
- ¿Cómo?
- ¿Qué?

La primera pregunta (*¿Por qué?*) nos acerca al motivo por el cual estamos por construir el software, el objetivo por el cual el negocio nos solicita el proyecto.
La segunda pregunta (*¿Quién?*) nos acerca a quienes serán los afectados por el software, quienes obtendrán los beneficios de su creación.
La tercera pregunta (*¿Cómo?*) nos acerca a la forma en la cual pueden los actores comportarse para alcanzar el objetivo, que impactos son los que deben producirse.

La cuarta y última pregunta (*¿Qué?*) nos acerca a de qué manera el software que desarrollemos brindará soporte para los impactos que buscamos.

En la Figura 13 podemos ver estas 4 preguntas relacionadas con cada componente principal:

- El **objetivo** del cual partimos.
- Los **actores** que identificamos.
- Los **impactos** o las acciones que podrán hacer los actores.
- Los **entregables** que producirá el software.

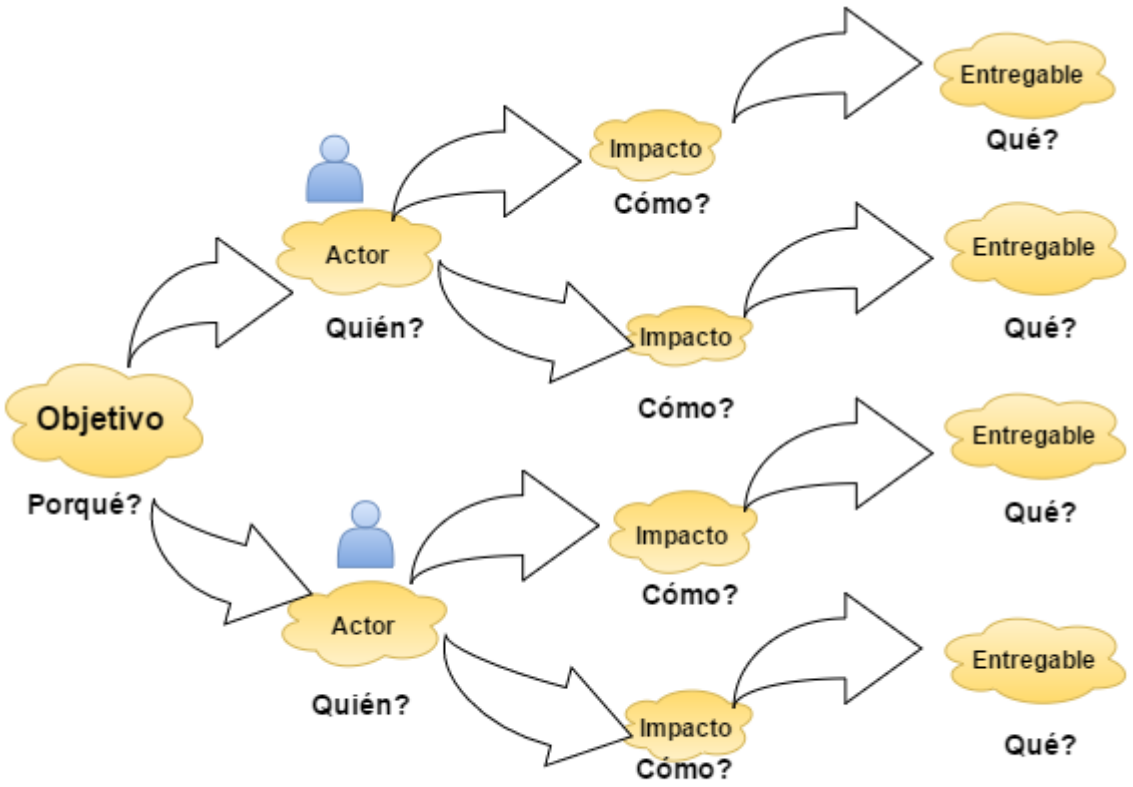


Figura 13. Impact Mapping

Ahora bien, la pregunta es: ¿cómo llevamos esta técnica a cabo?

La respuesta es bastante simple, por un lado, reunimos tanto a miembros técnicos del equipo como a expertos del negocio. Buscamos tener a disposición un espacio con mesas, sillas y una pizarra o pared blanca, fibras o marcadores y post-it de varios colores. También necesitaremos de una persona que esté encargada de llevar a cabo la reunión, esta persona puede ser un miembro del equipo, que se encargará de guiar al resto de los participantes y de colocar en forma de mapa los post-it en la pizarra o pared.

Las conversaciones y preguntas entre los participantes serán las fuentes de donde surgirán todas las respuestas presentes en el mapa. Hay que tener en cuenta que lo que se espera de esta reunión:

- al finalizar debe haber un mapa resultante de las conversaciones.
- el tiempo que se haya dedicado dependerá de la cantidad de personas, y de la habilidad de quien guíe la misma. En general se estima una reunión de entre 2 a 4 horas.
- los participantes deben irse con una clara visión a alto nivel de lo que se espera del proyecto y sobre todo por qué se está haciendo el proyecto y con qué fin.

A partir del *impact mapping* podemos definir las features que vamos a tener en nuestro proyecto, siguiendo el formato utilizado en BDD (4):

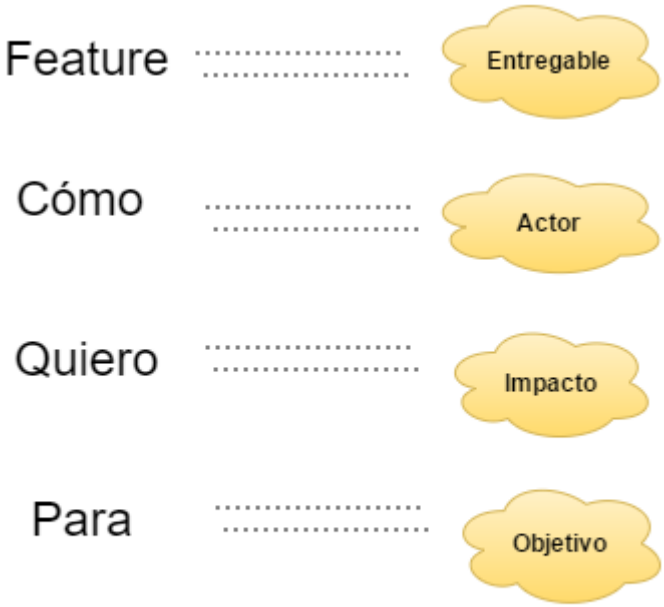


Figura 14. Feature en BDD

Este formato nos permite hacer foco en el valor del negocio que la *Feature* o entregable intenta satisfacer. Si bien el formato es muy similar a la historia de usuario, una feature puede ser una historia de usuario o contener a más de una. Recordemos que una feature es un **trozo de funcionalidad** que se entrega y que da soporte a un impacto que los actores necesitan para alcanzar un objetivo. En cambio, una historia de usuario es una **herramienta de planificación** que ayuda a complementar los detalles necesarios para entregar una feature en particular. (4)

3.3.2. Planificando con ejemplos.

Visual Story Planning

"Es una herramienta que permite generar una representación visual del sistema completo. Ofrece una vista general de todas las funcionalidades que lo componen (the big picture) de punta a punta. Permite identificar historias de usuario faltantes en el Backlog, planificar Releases partiendo en rebanadas (Slicing), visualizar cómo se distribuyen las funcionalidades de acuerdo a las diferentes áreas del sistema." (26)

Al igual que en la técnica anterior, será necesario contar con todos los miembros del equipo y con representantes del negocio, mesas, sillas, fibras, post-it. Se determinan de antemano cuáles son las features a tratar, de manera de limitar el alcance de la reunión. También vamos a necesitar de una persona encargada de guiarla, se suele llamar a esta persona el "*facilitador*"[45] y es deseable que no sea un miembro del equipo simplemente por el hecho de no viciar el proceso y poder tener una visión más objetiva.

Cómo se construye:

1. Identificar las distintas grandes funcionalidades en las que podemos dividir nuestro sistema (Actividades) y situarlas en un orden lógico secuencial.
2. Descomponer esa actividad en las tareas que el usuario hace (Tareas de usuario) y ordenarlas secuencialmente.
3. Identificar las sub-tareas que deben ser desarrolladas, es importante tener en cuenta que cada una de ellas debe entregar valor.

Cómo lo haremos para adaptarlo a BDD:

1. Utilizar las **features** obtenidas del impact mapping según un orden cronológico secuencial.
2. Descomponer esa feature en **historias de usuario** y disponerlas secuencialmente.
3. Identificar los **ejemplos** de cada historia de usuario que nos ayudarán a describirla.

[45] (..) es quien vela por el proceso, la conversación, la interacción, pero sin influencia directa sobre el contenido y resultado de la toma de decisiones del equipo.
http://www.martinalaimo.com/es/blog/siendo-facilitador

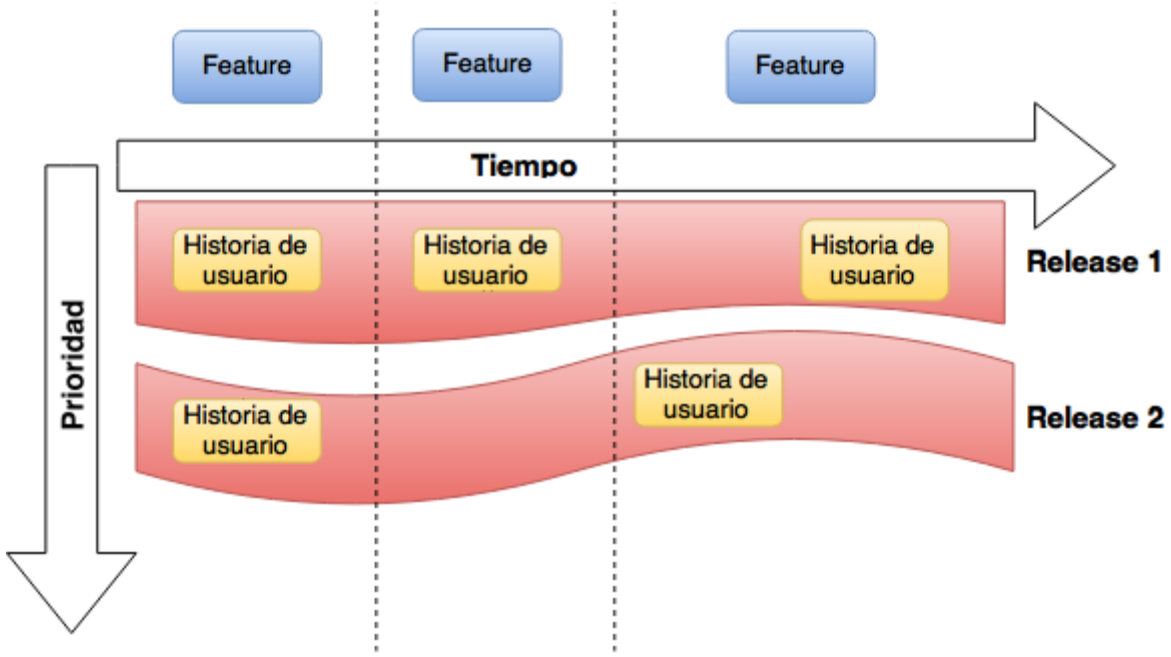


Figura 15. Visual Story Mapping

Esta técnica brevemente se resume en (Ver Figura 15):

- Las historias de usuario se ordenan de izquierda a derecha contando una historia "…*y entonces / después* ... "describiendo la secuencia de eventos que deben darse.
- De arriba a abajo el mapa contiene variaciones y alternativas.
- Las tareas se agrupan en Actividades (en nuestro caso será en features) que forman la columna vertebral ("*backbone*") del mapa.
- El mapa se puede cortar para agrupar todas las tareas necesarias para un objetivo específico para armar Plan de Releases.

Lo que obtendremos como resultado será:

- Una visión gráfica de las historias de usuario que forman parte de las features que queremos llevar a cabo.
- Ejemplos claros para cada historia
- Un plan de releases.

Historias de usuario INVEST

Al igual que detallamos cuando hablamos de objetivos y dijimos que los mismos es deseable que sean SMART, al momento de hablar de historias de usuario es recomendable que las mismas sean INVEST (ver Figura 16).

Como describe Martín Alaimo (27) INVEST son 6 características deseables de toda historia de usuario:

- Independiente ("*Independent*")

Las Historias de Usuario deben ser independientes de forma tal que no se superpongan en funcionalidades y que puedan planificarse y desarrollarse en cualquier orden.

- Negociable ("*Negotiable"*)

No es un contrato explícito por el cual se debe entregar todo-o-nada. Por el contrario, el alcance de las Historias podría ser variable: pueden incrementarse o eliminarse con el correr del desarrollo y en función del feedback del usuario y/o la *performance* del Equipo.

- Valorable ("*Valuable*").

Deben tener un valor para el negocio.

- Estimable ("*Estimable*").

Se deben poder estimar cual es el costo de la historia, para ello debemos contar con cierto conocimiento técnico y funcional mínimo suficiente.

- Pequeña ("*Small*").

Debe ser lo suficientemente pequeña de forma tal que permita ser estimada por el equipo de Desarrollo.

- Verificable ("*Testable*").

Se espera que la historia se pueda probar para validar que la misma se comporta de manera esperada.

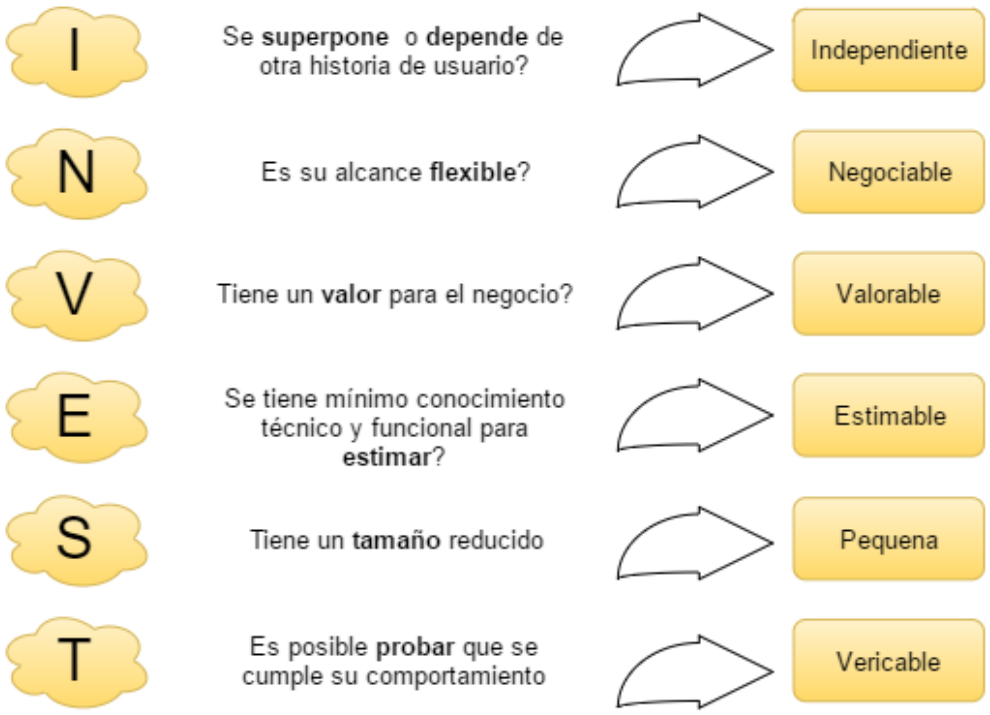


Figura 16. Historias de usuario INVEST.

Reunión de 3 Amigos

Una vez que contamos con las historias de usuario identificadas, pasaremos entonces a refinarlas y para ello utilizaremos la técnica conocida como "*Reunión de 3 amigos*" que fue descripta originalmente por Gojko Adzic (*"Three Amigos"*). Si bien esta técnica la podemos utilizar directamente en lugar de realizar primero una "*Story planning*", en nuestro caso la usaremos para refinar el conocimiento de las historias de usuario.

Lo que buscamos con esta técnica es obtener los criterios de aceptación de las historias (que serán descriptos en forma de escenarios) de forma colaborativa.

Esta estrategia consta de 3 figuras o roles que participarán (4):

- Un DEV (que nos dará la perspectiva de cómo se **hacen** las cosas). Que pondrá especial énfasis en las consideraciones y limitaciones técnicas.
- Un QA (que nos dará la perspectiva de cómo **romper** las cosas). Que pondrá especial énfasis en detallar cómo se podrán validar el comportamiento
- Un BA (que nos dará la perspectiva como **deben** funcionar las cosas). Que pondrá especial énfasis en el valor para el negocio.

Juntos conversarán acerca de la historia de usuario que tengan asignada, tomando los ejemplos que se han descrito en el Story mapping y describirán los criterios de aceptación en forma de escenarios utilizando Gherkin (Ver Figura 8).

En la Figura 17 podemos ver gráficamente como la participación y perspectiva brindada por cada uno de los participantes influye en la generación de los criterios de aceptación.

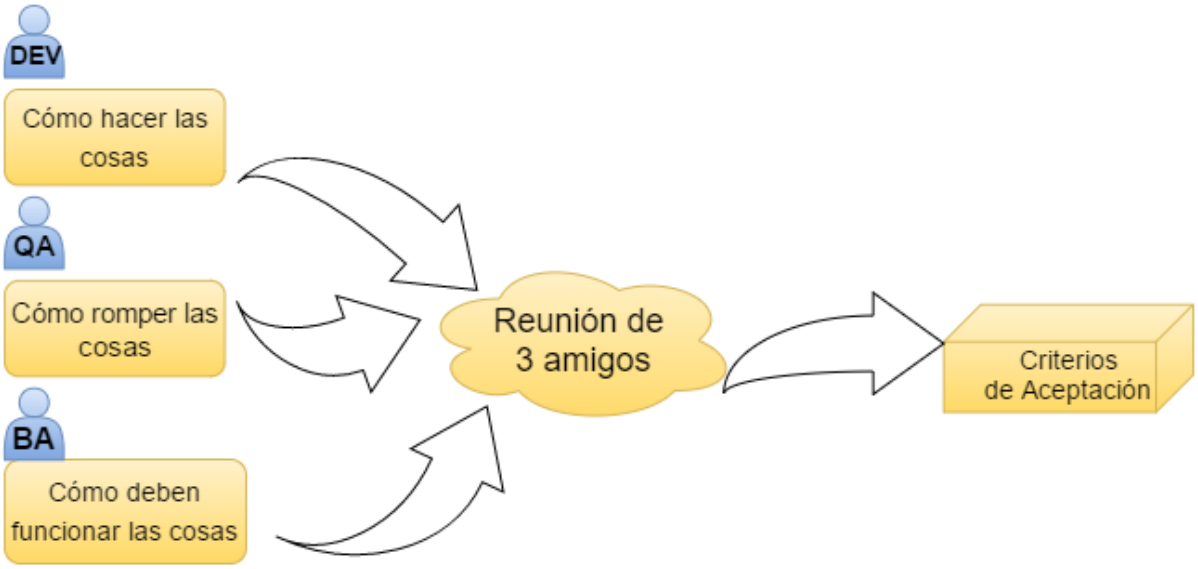


Figura 17. Reunion de 3 amigos

El resultado de esta reunión serán entonces nada más y nada menos que los criterios de aceptación para una historia de usuario (pueden ser más de una). Pero algo que es importante destacar en este punto es que como resultado: el DEV posee un conocimiento más profundo de la historia que tiene que desarrollar, el QA tiene los escenarios principales que sabe deberá validar, y el BA se asegura que los principales escenarios han sido planteados.

3.3.3 Mejorando el proceso de comunicación: feedback

“Generating high-quality relevant feedback, as far as possible from within rather than from experts, is essential for continuous improvement, at work, in sport and in all aspects of life”, John Whitmore.

La importancia del feedback

> *“(..)feedback es el proceso mediante el cual alguien que recibe un servicio demuestra su nivel de satisfacción al respecto, pudiendo hacer sugerencias para mejorarlo”.* (28)

En todo proyecto, sea ágil o no, el feedback es una de las herramientas de las que más tendríamos que hacer uso, ya que nos permite detectar tempranamente errores o desviaciones hacia el objetivo que puedan producirse, nos permite conocer cuál es el nivel de satisfacción del negocio con respecto al trabajo que estamos llevando a cabo.

Contar con feedback no sólo constante sino también rápido es esencial para el desarrollo de software. El feedback es el medio con el cual contamos para saber “*que tan bien venimos*” que además nos da una pauta de cómo mejorar lo que pueda estar fallando.

A continuación, mencionaremos algunas técnicas que vistas desde el punto de vista del feedback que brindan, son útiles para su incorporación en un proyecto.

Daily meeting -Feedback diario

Una “*Daily meeting*” o su traducción *“Reunión diaria”* consiste en reunir todos los días al equipo de desarrollo, donde cada miembro secuencialmente habla durante 2-3 minutos respondiendo a estas 3 preguntas:

- ¿Qué hiciste **ayer**?
- ¿Qué harás **hoy**?
- ¿Hay algún **impedimento**?

La “*Daily meeting*” también es conocida con otros nombres como “*Stand up*” (que hace referencia al hecho de realizar esta reunión encontrándose todos sus participantes parados físicamente) es una práctica que se plantea en SCRUM.

Estas reuniones nos permiten conocer cuál es el estado actual del equipo con respecto al objetivo del proyecto. Asimismo, le brinda al equipo una identidad y sentido de pertenencia al mismo. Como podemos ver en el artículo de Jason Yip (29), existen más beneficios del uso de esta práctica, consideraciones y recomendaciones para realizarla de manera eficiente, no haremos una descripción detallada de la práctica en el presente trabajo.

Desde el punto de vista del feedback, las Daily meeting no hacen otra cosa que brindar un **feedback diario del equipo hacia el equipo**, ya que nos permite detectar tempranamente cualquier problema o posible problema a futuro que pueda afectar al trabajo comprometido. Este es el principal motivo por el cual hacemos mención de esta práctica y por el cual es tan importante incorporarla como práctica a nuestro proyecto.

Sprint Review - Feedback del Sprint

Una "*Sprint Review"* o su traducción "*Revisión de sprint"* proporciona un punto de inspección para el progreso del proyecto al final de cada Sprint. El propósito de esta reunión es que el equipo presente de manera informal lo que se estuvo trabajando durante el sprint, esta presentación puede darse en algunos casos en forma de "*Demo*[46]".

¿Quienes participan de esta reunión? En principio todo el equipo de desarrollo debe participar, así como también *Scrum master*[47] y *Product Owner* (en el caso que dichos roles se encuentren definidos), caso contrario es recomendable que algún representante del negocio o stakeholder esté presente.

Durante esta reunión el Product Owner o representante del negocio evaluará el trabajo realizado durante el sprint por parte del equipo, y aquí es donde viene lo más importante y el motivo de esta reunión: dará su feedback al respecto. Con esto lo que queremos decir es que determinara si las soluciones desarrolladas son válidas o no, si acaso satisfacen las necesidades planteadas por el negocio, sí es que hay que realizar cambios, si algo hay que volver a hacer, etc. Es por eso que este tipo de reuniones son de gran valor para obtener un feedback, nos señalan el margen de error existente en el trabajo que venimos realizando.

Retrospective Meeting- Feedback general

Una "*Retrospective meeting"* o su traducción "*Retrospectiva*" es una reunión que se lleva a cabo luego de finalizada una iteración (puede ser un Sprint o un Ralease) donde el equipo reflexiona acerca de qué cosas estuvieron bien y que cosas no. Lo más importante de esta reunión es que se obtiene un feedback y se elabora un plan de acción para modificar/mejorar aquellas cosas que no estuvieron bien durante la iteración.

Como bien explica Rishi Devendra (32) un "*sprint Review"* mira que es lo **que** el equipo está construyendo, mientras que una "*retrospective meeting*" mira **como** está siendo construido.

Existen 3 preguntas/puntos principales que se deben incluir:

- ¿Qué cosas anduvieron **bien** durante la iteración(sprint/release)?
- ¿Qué cosas anduvieron **mal** durante la iteración(sprint/release)?
- ¿Qué podemos hacer diferente para mejorar?

---

[46] *"Actividad de una revisión de sprint donde los ítems completados del product Backlog son mostrados con el objetivo de obtener un feedback*"
http://www.innolution.com/resources/glossary/sprint-demo

[47] "(...) *es la figura que lidera los equipos en la gestión ágil de proyectos. Su misión es que los equipos de trabajo alcancen sus objetivos hasta llegar a la fase de "sprint final", eliminando cualquier dificultad que puedan encontrar en el camino"*
http://comunidad.iebschool.com/iebs/agile-scrum/definicion-y-caracteristicas-del-scrum-master/

Una "*Retrospective meeting"* se puede estructurar con las siguientes partes (31):

- Preparar el escenario => lograr que las personas se focalicen en los objetivos de la reunión, en el tiempo estipulado y con una dinámica productiva.
- Recabar datos => lograr una visión común de la situación a analizar, tanto con datos objetivos como subjetivos. Es la base común de hechos, eventos y sentimientos que permitirá tener una comunicación efectiva durante el resto de la reunión.
- Generar entendimiento profundo => entender el porqué, tanto lo que anduvo mal como lo que anduvo bien. Ir más allá de la primera apariencia, para encontrar las causas profundas que hay que sostener y mejorar o cambiar.
- Decidir qué hacer => teniendo una lista de las posibles acciones que el equipo puede realizar para mejorar, es el momento de elegir cuales se llevarán a cabo.
- Cierre => finalizar claramente la retrospectiva, con una nota positiva y ganas de realizar las acciones que se encontraron

3.3.4. Resumen Fase 1

Hasta este punto hicimos hincapié en la comunicación, indicando como ciertas técnicas nos ayudan a mejorar la comunicación, señalando además cuál es la entrada necesaria de cada una de ellas y que salida o resultado se obtienen luego de realizarlas.

Como se puede ver en la Figura 18, partimos de los objetivos del negocio que deberían ser **SMART**, con ellos podemos plantear entonces un ***Impact Mapping*** que tendrá como resultado final la obtención de las **features**, dichas features serán el punto de partida en el ***Visual Story planning*** que nos permitirá especificar las historias de usuario y además planificar los releases, con dichas historias de usuario que es deseable cumplan con **INVEST** llevaremos a cabo la ***reunión de 3 amigos***, en la cual describiremos los ejemplos en de las historias en forma de escenarios que serán nuestros **criterios de aceptación** para la historia de usuario.

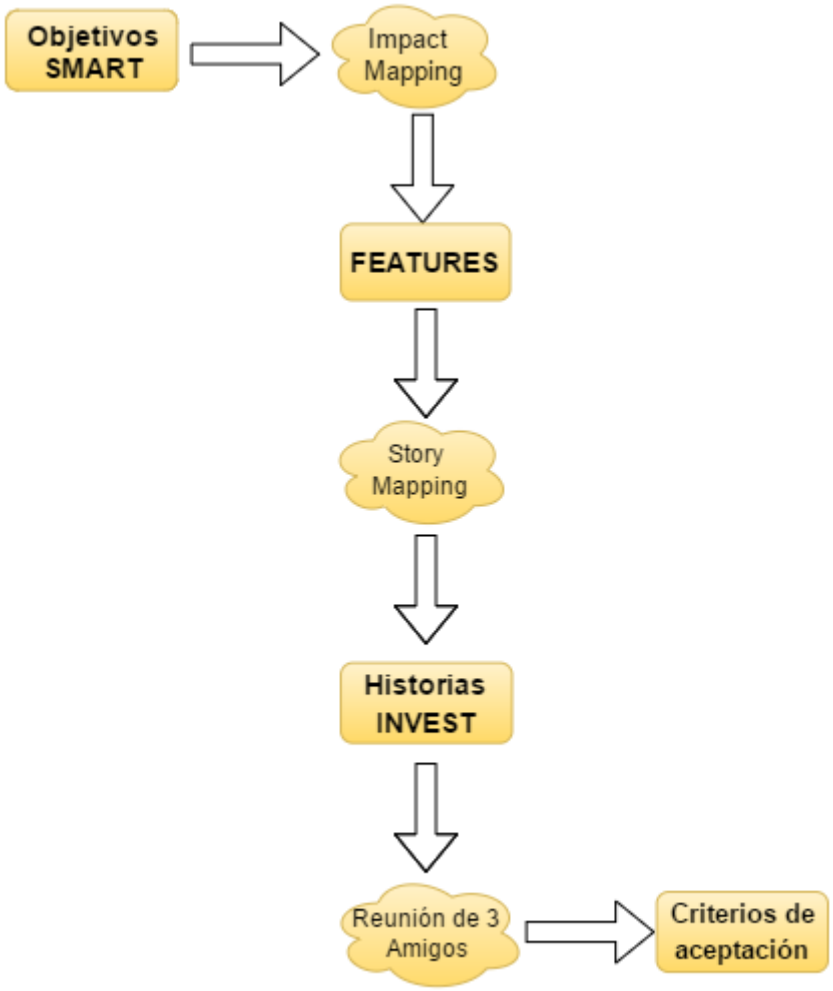


Figura 18. Resumen Fase 1-Comunicación.

Por lo otro lado en esta sección también hablamos de la importancia del feedback y planteamos 3 prácticas a considerar para obtener un feedback que nos permita tomar decisiones; la primera de ellas la ***Reunión diaria***, la cual nos brinda un feedback día a día y principalmente nos ayuda a detectar problemas que afecten al trabajo comprometido, la segunda es la ***revisión de sprint,*** momento en el cual el trabajo realizado es presentado y se recibe el feedback del mismo, y por último la ***retrospectiva*** , que le brinda al equipo un espacio donde poder analizar qué cosas se hicieron bien, y qué cosas se hicieron mal, y cómo determinar qué acciones se pueden llevar a cambio para revertirlas en las próximas iteraciones. En la Figura 19 podemos ver gráficamente como estas 3 prácticas alimentan el feedback del proyecto.

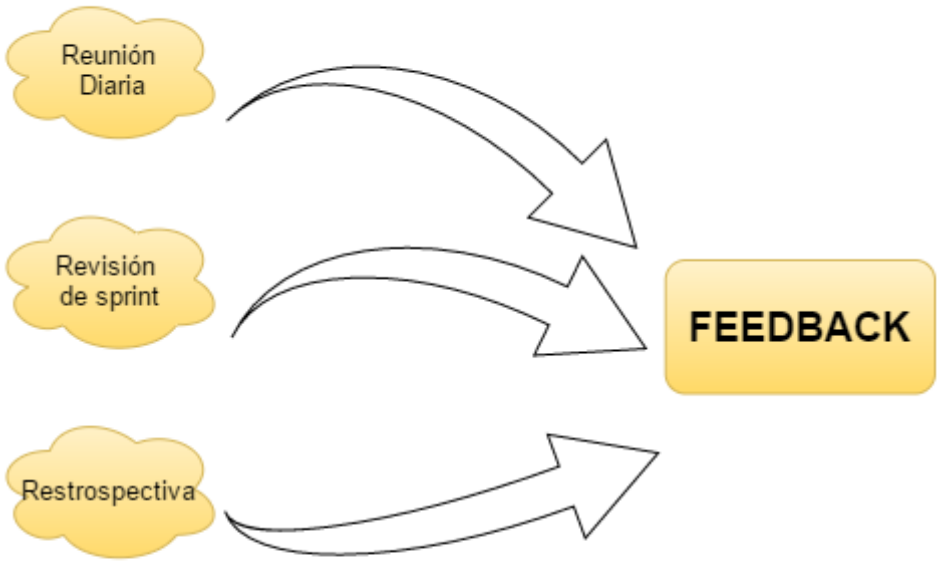


Figura 19. Resumen Fase 1-Feeback.

3.4 Fase 2 de Automatización

"The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency.", Bill Gates

3.4.1. Emprendiendo el camino de la automatización

¿Porque automatizar? ¿Qué hay que tener en cuenta?

*"Automatización es el uso de una máquina o mecanismo diseñado para seguir un patrón determinado y una secuencia repetitiva de operaciones respondiendo a **instrucciones predeterminadas**, sustituyendo el **esfuerzo físico humano** o la rutina por la observación o toma de decisiones"*[48]

Como podemos ver de la definición de automatización lo que estamos haciendo es sustituyendo el **esfuerzo físico humano** (con ello hacemos referencia a la ejecución manual de pruebas) por un conjunto de **instrucciones predeterminadas** que una máquina se encargará de ejecutar (en nuestro caso hacemos referencia a alguna herramienta especializada en las pruebas que debamos llevar a cabo).

La automatización en el proceso de desarrollo nos permite:

- Aumentar la calidad del desarrollo.

Cada vez que contamos con una persona que se encarga de realizar las pruebas sobre un software, siempre estamos teniendo en cuenta que una persona como tal puede cometer errores y eso no sucede con las herramientas de automatización, ya que no cometen errores. Por ende, la calidad podemos decir que aumenta ya que estamos confiando el resultado de las pruebas a una máquina que ejecuta y hará exactamente lo que le indiquemos.

- Realizar regresiones en menor tiempo.

Una de las mayores ventajas que poseen las herramientas que ejecutan pruebas automatizadas es que a diferencia de un ser humano no se cansan, no se estresan y siempre son capaces de obtener el mismo resultado para un mismo valor de entrada: son consistentes, confiables y repetibles lo cual resulta perfecto para la ejecución de pruebas de regresión. Las regresiones suelen ser tareas muy monótonas para las personas encargadas de realizarlas lo cual puede llevar a fallas en su ejecución, cosa que no pasa en la automatización, además que la misma se realiza mucho más rápido.

[48] http://automatizarimportancia.blogspot.com.ar/

- Detectar tempranamente errores.

Uno de los mayores inconvenientes en el desarrollo de un producto de software en hallar fallas o errores en un momento tardío, por ejemplo, cuando se está a 2 días de salir a producción, el costo de solucionar un problema aumenta casi exponencialmente a medida que se avanza en los tiempos de desarrollo, cuanto antes se encuentren los errores siempre será más sencillo solucionarlos, ya que además contaremos con más tiempo para obtener una versión estable y evitaremos el re trabajo. Ejecutar pruebas automatizada frecuentemente nos permite detectar errores más rápidamente.

- Reutilizar casos de prueba.

La automatización de pruebas conlleva a una mayor organización, ya que sin la misma no es posible obtener resultados certeros que brinden un *feedback* al equipo, al tener una organización de las pruebas es posible detectar aquellas pruebas o métodos que son utilizados con mayor frecuencia y de esa manera reutilizarlas en lugar de generar nuevas pruebas.

- Reducir los tiempos de ejecución y entrega.

Como decíamos anteriormente una herramienta de automatización no se cansa ni se estresa al igual que un humano, ni tampoco le preocupa estar ejecutando una prueba a las 2 AM. con lo cual los tiempos de ejecución se reducen ya que podemos tomar la ventaja de ejecutar en horarios que una persona no trabaja, la gran ventaja de ello es que al otro día la persona encargada de analizar los resultados, tendrá un informe de las ejecuciones y podrá poner foco sólo en aquellas pruebas que estén fallando. Así mismo debido a que el tiempo de validación y verificación del software es menor, el tiempo de entrega del mismo también se reduce en consecuencia.

Con lo cual uno podría pensar que la automatización es la solución mágica a todo, sin embargo, tenemos que tener en cuenta:

- Curva de aprendizaje de la herramienta.

Toda herramienta conlleva un aprendizaje, siempre que consideremos el peor caso: vamos a suponer que ningún miembro del equipo conoce cómo funciona la herramienta que se quiere utilizar, con lo cual la curva de aprendizaje será pronunciada. En el caso que el equipo ya tenga experiencia en la herramienta que se utilice aun así siempre habrá un aprendizaje faltante, ya que todo proyecto es un mundo nuevo y tendrá necesidades diferentes. Además, la curva de aprendizaje se verá particularmente afectada por las capacidades técnicas que posean los miembros del equipo y del tiempo destinado a dicho aprendizaje.

- Costo monetario de la herramienta.

La mayoría de las herramientas disponibles hoy en el mercado brindan dos opciones: una opción gratuita por tiempo limitado o ilimitado con funcionalidades limitadas y una versión paga con funcionalidades completas. Uno de los mayores inconvenientes el momento de decidir qué herramienta utilizar es que probablemente el equipo de desarrollo seleccione una herramienta que se adapte a las características del proyecto, pero el negocio no tenga el dinero suficiente para solventarla a lo largo del tiempo.

- Tiempo de arranque o startup de la herramienta.

Como mencionamos antes con respecto al aprendizaje, tiene un costo en tiempo, sobre todo al principio cuando es necesario dedicar el tiempo apropiado para obtener una versión estable y confiable del uso de la herramienta.

- Limitaciones de la herramienta.

La mayoría de las herramientas brindan una opción con funcionalidad limitada, si es nuestro caso entonces habrá ciertas funcionalidades que nos veremos no capaces de realizar. Y aun cuando se tenga la posibilidad de contar con una herramienta paga, suele ocurrir que haya que utilizar más de una herramienta para poder satisfacer todas las necesidades del proyecto.

- Cambios en la cultura de testing.

Este punto afecta sobretodo a equipos que recién comienzan a incorporar la automatización y que por años se han dedicado a realizar todas sus pruebas de forma manual. El cambio de paradigma y de cultura suelen ser costosos de aceptar, ya que el cambio que se produce es bastante grande y no todas las personas incorporan los cambios de la misma manera. Por eso en este punto es crucial la comunicación dentro del equipo (lo que profundizamos en la sección anterior de este capítulo).

- Costo de mantenimiento.

Al igual que cualquier herramienta, la automatización lleva un mantenimiento, ya que con el tiempo las pruebas que hayamos creado pueden o bien convertirse en obsoletas porque la funcionalidad ha cambiado o se ha eliminado o bien puede haber variado demasiado a lo largo del tiempo. Por eso es sumamente importante que las pruebas automatizadas están actualizadas ya que de eso depende el valor que sus resultados puedan brindarnos.

Todos estos puntos se deben evaluar con respecto a la elección de la herramienta que pondrá en marcha la automatización en nuestro equipo; el costo monetario depende de los costos que pueda solventar el negocio para nuestro proyecto, mientras que la curva de aprendizaje dependerá enteramente de las capacidades de los miembros del equipo.

Cuándo Si automatizar o QUÉ automatizar

A grandes rasgos podemos decir que cualquier tarea que sea **tediosa o repetitiva** involucrada con el desarrollo del software es candidata para automatizar. Claramente esa aseveración es demasiado inclusiva y genérica, mencionaremos a continuación un listado que describen Crispín y Gregory (6) acerca de qué es lo que podemos automatizar en un proyecto. Tengamos en cuenta que dadas las condiciones del proyecto algunos de estos puntos serán viables y otros no.

Que podemos automatizar:

- Integración, despliegue y entrega de código. Hablaremos más en detalle en la sección próxima (Automatización continua)
- Pruebas unitarias. En nuestro caso las llamamos especificaciones de bajo nivel y también más adelante hablaremos en detalle de las mismas.
- Pruebas de API[49] o servicios web.
- Pruebas detrás de la UI o interfaz de usuario.
- Pruebas de UI o interfaz de usuario.
- Pruebas de rendimiento y de carga de datos.
- Tareas repetitivas.
- Creación de datos y configuración

Cuándo NO automatizar o qué NO automatizar

Si bien es muy común que en algunos proyectos cuando se plantea la idea de automatizar se comete el error de intentar automatizar TODO, esto no es necesario principalmente porque el costo tanto en tiempo y en esfuerzo es alto y además porque el mantenimiento debe ser considerado también. Requerimientos no funcionales como la *escalabilidad* y el *rendimiento* y *usabilidad* son también un claro ejemplo de cosas que no es conveniente automatizar.

Como describen Crispín y Gregory (6) algunas pruebas requieren oídos, ojos e inteligencia humana. Las pruebas de usabilidad y las de exploración son dos que caen en esta categoría. Otras pruebas que pueden no justificar la inversión en automatizarse son las pruebas excepcionales o aquellas pruebas que sabemos a priori que no fallará. Describamos un poco los motivos de no invertir en automatizar dicho tipo de pruebas:

- Pruebas de usabilidad.

Las pruebas reales de usabilidad requieren que "*Alguien*" real utilice el software. La automatización puede ser de ayuda en la configuración que subsecuentemente será examinada por la usabilidad. Observar usuarios en acción, conocer sus experiencias, y evaluar los resultados, es el trabajo de la persona que entiende que los aspectos de usabilidad del software no pueden ser automatizados. Si pensamos en cuanto al "*look and feel*" e intentamos automatizarlo, sabremos que una prueba automatizada solo podrá ver lo

[49]Es un conjunto de reglas (código) y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas: sirviendo de interfaz entre programas diferentes de la misma manera en que la interfaz de usuario facilita la interacción humano-software.
http://www.ticbeat.com/tecnologias/que-es-una-api-para-que-sirve/

que se indique que tiene ver, con lo cual es probable que pierda encontrar problemas que son evidentes para una persona.

- Pruebas de exploración.

Así como la usabilidad las pruebas de exploración se pueden agilizar mediante pruebas automatizadas para la creación de datos y configuraciones, pero requerirá de un QA habilidoso para diseñar y ejecutar dichas pruebas. El principal objetivo de las pruebas de exploración es aprender qué es lo que el producto está haciendo y utilizar dicha información para mejorar desarrollos futuros.

- Pruebas que nunca fallaran.

Se suele decir que un test que nunca fallará, no necesita ser automatizados, sí un requerimiento es tan obvio que solo existe una forma de implementarlo, las posibilidades de introducir un defecto en dicha parte del código son cercana a cero.

- Pruebas excepcionales.

La mayoría de las veces ejecutar de manera manual pruebas excepcionales es más que suficiente. Con lo cual automatizar este tipo de pruebas no genera un beneficio, y el sentido de tomarse el trabajo de hacerlo pierde validez. Las tareas tediosas y repetitivas son las que valen la pena automatizar, aun cuando dichas pruebas no se ejecutan de manera frecuente. Con lo cual sí es fácil hacerlo de forma manual y al automatizarlo no será más rápido, la respuesta es sencilla: dejémosla manual.

Si quisiéramos una forma rápida de determinar si automatizar alguna prueba pensemos en las siguientes preguntas:

- ¿Ganaremos tiempo al automatizar la prueba?
- ¿Podremos ejecutar la prueba de manera eficiente?
- ¿Es fácil realizar la prueba de forma manual?
- ¿Es una prueba repetitiva? ¿Se debe ejecutar frecuentemente?
- ¿Hemos seleccionado una herramienta para trabajar la automatización?

3.4.2 Automatizando especificaciones

“Focus on the process not on the tools” Gojko Adzic

Especificaciones de alto nivel

Hasta este momento en el capítulo lo que hemos hecho ha sido obtener los criterios de aceptación para nuestro proyecto, los cuales describiremos en forma de escenarios utilizando el formato GHERKIN y que son entonces *especificaciones ejecutables de alto nivel*.(ver Figura 6) .Ahora bien como explica Gojko (2) para que sean efectivas, tenemos que ser capaces de poder ejecutarlas *frecuentemente* y *rápidamente*, sin que nos tome demasiado tiempo hacerlo (sea tiempo del desarrollador, analista, etc.). La mejor manera de hacerlo es automatizando tanto como sea posible, ya que esto nos permite fácilmente chequear sí vamos por buen camino.

¿Y qué ganamos automatizando los escenarios entonces? Bueno cuando es posible automatizar un escenario y tiene sentido hacerlo entonces obtenemos algunos beneficios (4):

- Los QA pierden **menos tiempo** en regresiones repetitivas.
- Versiones nuevas del producto son liberadas **más rápido** y de forma **más confiable**.
- Los escenarios automatizados brindan una **visión más precisa** del estado actual del proyecto.

¿Y cuáles son los escenarios que automatizamos? Bueno podemos mencionar 2 grandes categorías:

- Escenarios con interfaz de usuario.

En este caso nos referimos principalmente a pruebas webs, aunque también podría ser cualquier otro tipo de interfaz con la que cuente la aplicación. Muchas veces los resultados visuales que podemos obtener son efectivos para describir CÓMO el usuario interactúa con la aplicación, además es una forma de hacer una “Demo” (ver la sección anterior) con los stakeholders. Y por otro lado estaremos reduciendo la cantidad de pruebas manuales para la interfaz de usuario siempre y cuando los tiempos de ejecución sean mejores que los manuales.

- Escenarios sin interfaz de usuario.

En este caso nos referimos a escenarios que no involucran la utilización de interfaces de usuario y se basan principalmente el validar la lógica y reglas de negocio de la aplicación, así como también todas las restricciones bajo las cuales funcionará. En esta categoría también se incluyen los requerimientos no funcionales que pueda tener la aplicación.

Entonces, volviendo al principio: tenemos nuestros escenarios descritos en GHERKIN y ahora qué sigue?

A grandes rasgos diremos que lo queremos hacer es:

1. **Definir** los "*steps definitions*"
2. **Implementar** los "*steps definitions*"

Para el punto 1 lo que haremos es tener los escenarios escritos en alguna herramienta de BDD qué puede ser Cucumber, RSpec o cualquier otra del mercado que hayamos seleccionado. Una vez que tengamos los escenarios en la herramienta lo que haremos será entonces desglosar a partir de ellos los "*steps definitions*".

*Los "**steps definitions**" son trozos de código que interpretan el texto de la definición del escenario y que saben que especificación de bajo nivel llamar.*

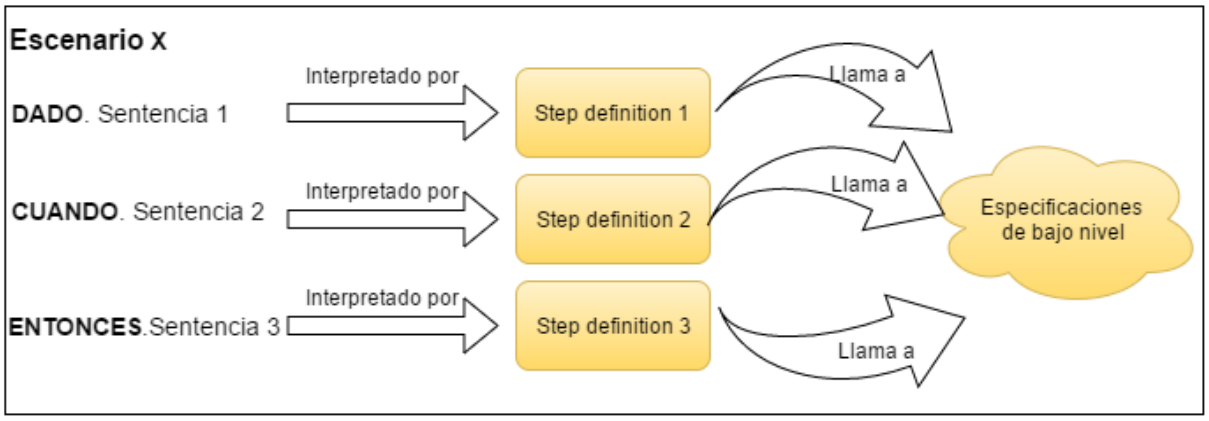


Figura 20. Steps Definitions

Cabe aclarar que el lenguaje en el que se encuentre escrito este trozo de código NO es necesariamente el mismo lenguaje en que se encuentre escrito el código de nuestra aplicación, ya que de hecho varía según la herramienta que hayamos elegido (Si usamos RSpec por ejemplo el lenguaje de programación será Ruby, sí en cambio usamos Cucumber tenemos una amplia gama de lenguajes de programación que podemos utilizar para escribir los *"steps definitions*").

En cuanto al punto 2, implementar los "*steps definitions*" es el corazón de toda herramienta de automatización de BDD. Los "*steps definitions*" le dicen a la herramienta BDD exactamente qué código se debe ejecutar para cada paso o sentencia de los escenarios, lo que hacen es comunicarse con las especificaciones de bajo nivel que saben cómo llamar directamente al código de la aplicación.

Para implementar los "*steps definitions*" vamos a tener en cuenta:

- Envío parámetros.

Cuando definimos nuestros escenarios es muy probable que en algunos casos hayamos incluido algún dato de entrada que es requerido como condición para que se ejecute el escenario. Dichos datos son parámetros que vamos a necesitar que los "*steps definitions"* se encargan de interpretar y utilizar para enviar al código que se debe llamar.

- Estados.

Entre las distintas sentencias a menudo es necesario que comparta cierta información (recordemos que todas las sentencias forman parte de un mismo escenario y son necesarias para la ejecución del mismo) con lo cual en la implementación se debe proveer una manera de hacerlo.

- Datos de entrada.

Los escenarios surgen a partir de ejemplos y los ejemplos como tales cuentan con datos de entrada que pueden variar, o que puede tener distintos valores válidos. Debemos proveer una forma sencilla de contar con distintos valores de datos de entrada, tanto validos como no válidos. En general esto suele hacerse utilizando tablas con datos de entrada.

- Resultados posibles.

Tradicionalmente los resultados posibles de cualquier tipo de prueba son: Passed or failed. Pero en BDD lo que ejecutamos es un escenario, y cada escenario posee varios "*steps definitions*" cada uno de ellos puede tener varios resultados posibles (ver Figura 21), el valor que obtendrá el escenario dependerá del resultado de cada uno de ellos. Un "*step definitions*" que aún no fue implementado tendrá como resultado cuando se ejecute "*Pendiente*", una vez que haya sido implementado entonces los posibles resultados serán: "*Exitoso*" (se ejecutó de manera exitosa), "*Error*" (se produjo algún error al momento de ejecutarse puede ser en la implementación de la sentencia o en la aplicación) "*Fallo*" (se produjo algún error al momento de ejecutarse en la aplicación) "*Salteado*" (la sentencia no se ejecutó porque la anterior falló).

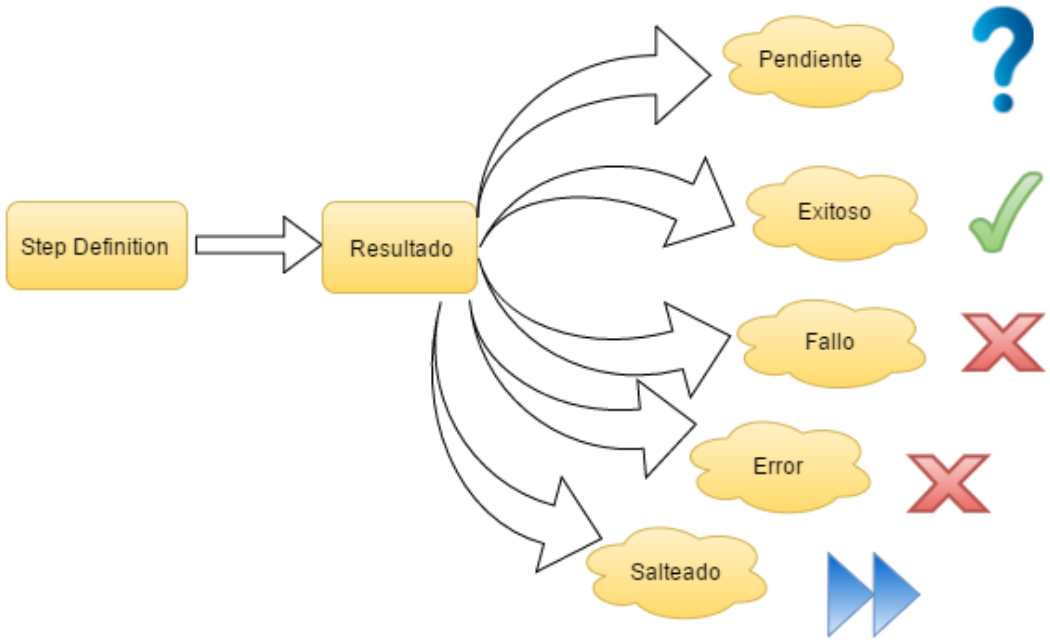


Figura 21. Resultados de "*step definition*".

Especificaciones de bajo nivel

Una vez que automatizamos los escenarios y sus "*steps definitions*", aún nos queda un tramo más: automatizar también las especificaciones de bajo nivel o "*implementar siguiendo ejemplos*" como lo llama North (19), que no es ni más ni menos que TDD al modo de BDD veamos un poco como hacer esto (ver Figura 6 y 7).

Si bien como mencionamos en la sección anterior a partir de los escenarios obtuvimos los "*steps definitions*" los mismo estarán en el estado "*pendiente*" hasta que desarrollemos el código necesario de la aplicación para que satisfaga la ejecución de dichos steps y como consecuencia estaremos buscando satisfacer los escenarios propuestos. Una vez que sabemos que es lo que el "*step definition*" debe hacer, nos guiaremos a partir de los mismos para desarrollar el código necesario y para ello lo que haremos será:

1. **Entender** el modelo de dominio
2. **Aplicar** TDD con comportamiento.

En el caso del punto 1, esta es una idea que BDD trae de uno de sus principales influencias: DDD. A la manera de BDD lo que haremos será realizar una reunión de diseño o de modelo (el nombre en sí mismo no debería hacer ninguna diferencia) entre los DEV principalmente y los QA que deseen participan, en la cual se establezca un diseño y arquitectura de alto nivel antes de comenzar a crear el código de la aplicación, Además este punto inicial hace que todos los DEV que participaron estén alineados y todos compartan un mismo entendimiento del comportamiento esperado de la aplicación a bajo nivel o como se suele decir en la jerga informática "para asegurarse que todos se encuentren en la misma página" ("to make sure we all are in the same page"). Esta reunión es un espacio para plantear preguntas acerca del cómo se llevará a cabo esta etapa de codificación, es por eso que puede ser de valor contar con la presencia de algún QA que proporcione una perspectiva distinta y ayude a complementar el buen entendimiento.

Para el punto 2 lo que haremos es basándonos en los "*steps definitions*" y tomándolos como guía, crearemos las pruebas unitarias o *especificaciones de bajo nivel*, generando luego el código necesario para que dicha prueba se ejecute de manera exitosa y posteriormente realizando el refactoring que sea necesario (Ver Figura 1). ¿Y cómo sabemos cuándo terminamos? Cuando todos los "*steps definitions*" tienen el estado "*exitoso*" cuando se ejecutan, para todos los escenarios definidos para una feature.

Si bien este puede ser considerado el punto principal de cualquier proyecto, ya que el código mismo de la aplicación se está creando en este punto, no nos detendremos demasiado ya que usaremos los mismos principios bien conocidos estipulados por TDD.

Aun así, mencionaremos algo tips a tener en cuenta:

- Pensar en las pruebas unitarias como **trozos de funcionalidad**, que están explicitando comportamiento de la aplicación.
- Al momento de crear el código necesario para una prueba unitaria crear las clases o métodos simples que surjan de manera inmediata.
- Utilizar una **implementación mínima**
- Utilizar **Mocks** cuando sea conveniente, sobretodo en los casos en los que el comportamiento sea complejo, de manera de poder descomponerlo de afuera hacia adentro.
- Recordar que BDD es acerca de **comportamiento y de comunicación** y que este lineamiento no debe perderse de vista en este nivel, más allá que sea un nivel bajo en la aplicación, con lo cual el vocabulario que se utilice para las pruebas unitarias debe ser fácil de entender y el código generado fácil de leer.

3.4.3 Automatización continua

"*A working program is one that has only unobserved bugs*", Ley de Murphy.

El fin último de BDD es entregar software de mayor valor para el negocio de la forma más eficiente posible. Uno de los mayores inconvenientes es que el verdadero valor de las features para el negocio no es tangible hasta que es llevado a producción donde los usuarios reales lo usan. Teniendo esto en cuenta lo que queremos lograr es entregar al negocio valores lo más rápido posible de manera de poder tener un feedback de los usuarios reales, y para ello será necesario que seamos capaces de desplegar features no solo más rápido sino también más eficientemente.

Desplegar features en producción es típicamente un proceso complejo. Es necesario construir la aplicación a partir del código fuente y ejecutar todas las especificaciones que venimos creando hasta este punto de forma de asegurarnos que cumplimos con el comportamiento esperado de la aplicación que ha sido definido en las features y luego crear un paquete que se sea desplegable en producción.

En pos de mejorar y acelerar este proceso es que explicaremos brevemente las prácticas: Integración Continua, Entrega Continua y Despliegue Continuo. Todas ellas prácticas deseables para aplicar en un proyecto de desarrollo de software.

Integración Continua ("*Continuous integration*")

Siendo Integración Continua una práctica ampliamente conocida dentro de las prácticas ágiles, existen varias definiciones, dos de ellas son las que se mencionaremos a continuación:

> *Integración continua es una práctica que involucra la construcción y prueba automáticos de un proyecto cada vez que un cambio en el código se incorpora al código fuente del repositorio. La integración continua es un mecanismo de feedback muy valioso, que mantiene alerta a los desarrolladores de potenciales problemas en las integraciones o regresiones lo antes posible. Pero para que sea realmente efectivo, la integración continua depende fuertemente de un conjunto robusto de pruebas automatizadas.* (4)

> *Integración continua es una práctica de desarrollo de software donde los miembros de un equipo integran su trabajo frecuentemente, usualmente cada persona íntegra al menos una vez al día. Cada integración es verificada por un "build" que se encuentra automatizado y que incluye pruebas, para detectar posibles errores en la integración tan rápido como sea posible. Muchos equipos encuentran esta estrategia los guía hacia una reducción en los problemas de integración y permite al equipo desarrollar software cohesivo mucho más rápido.* (34)

Podemos ver que en ambas se hace principal foco en dos cosas: por un lado, la ejecución de pruebas cada vez que realiza una **integración** o un cambio en el código y por otro lado se resalta la **frecuencia** con que esto se realiza. Ambos puntos son sin dudas los ejes principales de esta práctica, ya que ambos en conjuntos son los que permiten detectar errores lo antes posible de manera de minimizar los riesgos.

El principal objetivo de la integración continua es entonces proveer un **rápido feedback** del estado del proceso de construcción de una aplicación. En general un servidor de integración continua es una aplicación que monitorea continuamente cambios en el código fuente del proyecto y se configura de manera tal que cada vez que se genera una nueva versión del código se ejecuten todas las pruebas sobre esa nueva versión; en el caso de haber errores se notifica de inmediato a los desarrolladores.

Desde el punto de vista de BDD lo ideal sería que todas las especificaciones que venimos creando tanto las de alto nivel como las de bajo nivel sean parte del "*build*" automatizado, esto claramente dependerá tanto de las especificaciones que hayamos creado, como de la herramienta que seleccionemos para la integración continua, así como también el tiempo de ejecución que lleve ejecutar todo, recordemos que el principal objetivo es obtener un feedback rápido.
Para ser parte de la ejecución automatizada, las especificaciones deben:

- Ser auto suficientes. => Es decir que debe ser posible aislarlas de manera tal que si se detecta un error sea identificable y rápido de solucionar.
- Estar almacenados en un versionador de control. => De esta forma podemos tener acceso a las distintas versiones y podemos corroborar que sí lo que introdujo el error es el nuevo código u otra cosa.
- Se deben poder ejecutar desde consola. => Siguiendo con la misma idea de los dos puntos anteriores, debe ser posible una vez detectado el error ser capaces solo de ejecutar la especificación en la cual se está produciendo el error.

Debemos destacar que lo importante es aplicar la práctica más allá de la herramienta que sea seleccionada para tal fin, ya que existen actualmente varias opciones y entre algunas de ellas podemos mencionar las más conocidas como **Jenkins**,[50] **CruiseControl**[51] y **GO** [52].

Entrega Continua ("*Continuous delivery*")

Entrega continua es una extensión de integración continua, donde cada "*build*" es un potencial entregable. Cada vez que un desarrollador introduce nuevo código en el repositorio, se compila un nuevo "*build*" en el servidor que es candidato para la versión actual. Si el candidato pasa una serie de verificaciones de calidad automatizadas (pruebas unitarias, criterios de aceptación automatizados, pruebas de desempeño, etc.) puede ser enviado a producción tan pronto como el negocio brinde su aprobación para continuar.

Los principales beneficios de la entrega continua son:

- Se reduce el riesgo de despliegue y se obtiene más flexibilidad.

Dado que se realizan despliegues frecuentemente para cambios pequeños, existe menos probabilidad de que las cosas vayan mal (ya que no sea harán muchas integraciones de pronto) y es más fácil de solucionar problemas cuando aparecen.

[50] https://jenkins.io/
[51] http://cruisecontrol.sourceforge.net/
[52] https://www.go.cd/

- Se produce un progreso creíble.

Muchas veces suele darse que el concepto de "*done*" o terminado está dado por: "*el desarrollador dice que está terminado*" lo cual es mucho menos creíble que sí el código está integrado exitosamente.

- Se obtiene feedback de usuario.

El mayor riesgo de cualquier software es que el producto final sea algo que no le sirve al usuario. Cuando antes y más frecuentemente tengamos un producto que está funcionando frente a usuarios reales, es que más rápido se tendremos feedback de los mismos y se podremos saber realmente el valor que tiene para ellos el producto.

- Se tiene más tiempo para resolver errores

Los errores aparecen y son abordados en fases más tempranas. Están acotados en un mismo proceso, cuanto antes se detectan, antes se solucionan lo que a su vez genera que haya menos errores en producción.

Despliegue Continuo ("*Continuous deployment*")

Despliegue continuo es similar a integración continua, pero puede no existir ninguna etapa de aprobación manual. Cualquier "*build*" candidato que pase los controles de calidad será automáticamente desplegado en producción. El proceso de despliegue en sí mismo es automatizado con herramientas especializadas.

Ambos integración continua y despliegue continuo alientan a la racionalización del proceso de despliegue y ambos requiere de un alto grado de confianza en las pruebas automatizadas de la aplicación.

Despliegue continuo suele confundirse con entrega continua, la diferencia radica en que cuando hablamos de despliegue continuo una vez que se realizan cambios y se atraviesa exitosamente lo que se denomina "*Pipeline*"[53], automáticamente se introducen los cambios en el ambiente de producción, como resultado es posible realizar varios despliegues a producción al día. Despliegue continuo significa entonces que se tiene la capacidad de realizar despliegues frecuentemente, aunque en general esta acción debe pasar primero por una aprobación por parte del negocio, quien determina cuántos y cuándo se hacen efectivos los despliegues al ambiente productivo. Decimos que despliegue continuo es el siguiente paso a la entrega continua y que puede ser considerado también como una extensión, ya que la entrega continua es un paso necesario para poder realizar despliegues de manera continua.

[53]Son cada uno de los pasos que conforman el flujo automatizado de construcción y entrega del software. Cada uno de estos pasos se ocupa de realizar una tarea concreta, por ejemplo: un primer paso podría ser la descarga de código, un segundo paso la compilación del código descargado, etc. http://atsistemas.com/wp-content/uploads/downloads/2014/04/articulo_integracion_continua_entrega_continua.pdf

Para poder lograr despliegue continuo, el equipo debe confiar en la infraestructura de pruebas automatizadas (ya mencionamos antes que nos referimos a las especificaciones tanto de alto nivel como de bajo nivel) y en la ejecución de las mismas, ya que debe ser posible que la misma herramienta que utilicemos sea capaz de detectar problemas, abortar la ejecución en el punto que se encuentre, volver atrás los cambios que sean necesarios y disparar la intervención de personas (en general eso sucede mediante la notificación vía email a los desarrolladores y demás personas que se hayan configurado en la herramienta).

El principal beneficio del despliegue continuo es que obtenemos como resultado una reducción del tiempo existente entre que el código es creado por los desarrolladores y el usuario final puede ejecutarlo usando la aplicación.

3.4.4 Resumen Fase 2

Antes de poner manos a la obra en esta fase nos detuvimos brevemente a hablar un poco de las **razones** por las cuales es conveniente automatizar, puntualizando que es recomendado automatizar y que no. En resumidas cuentas, automatizamos por 2 motivos: ganar **tiempo** y mejorar la **calidad** de lo que hacemos. Ganar tiempo para poder realizar entregar al negocio de manera más rápida y mejorar la calidad con que realizamos dichas entregas.

Luego tomando como base los **criterios de aceptación** que obtuvimos en la fase 1 describimos los mismos en forma de **escenario**, para luego definir los "***Step Definitions***" y su implementación, lo que para esta fase llamamos **especificaciones de alto nivel** (Ver Figura 22).

Para finalizar con la automatización de especificaciones tuvimos en cuenta entender el **modelo de dominio** que presenta la aplicación y luego aplicando **TDD,** automatizar las **especificaciones de bajo nivel** que son quienes finalmente se comunicaran con el código de la aplicación.

Por otro lado, para completar la fase de automatización, hablamos resumidamente de 3 prácticas sumamente importantes y complementarias a la automatización de las especificaciones. Mencionamos en primer lugar que es y cuál es la importancia y beneficios de la **Integración Continua**, como la **Entrega Continua** es su extensión y finalmente cómo es ideal que se complete utilizando Despliegue **Continuo**.

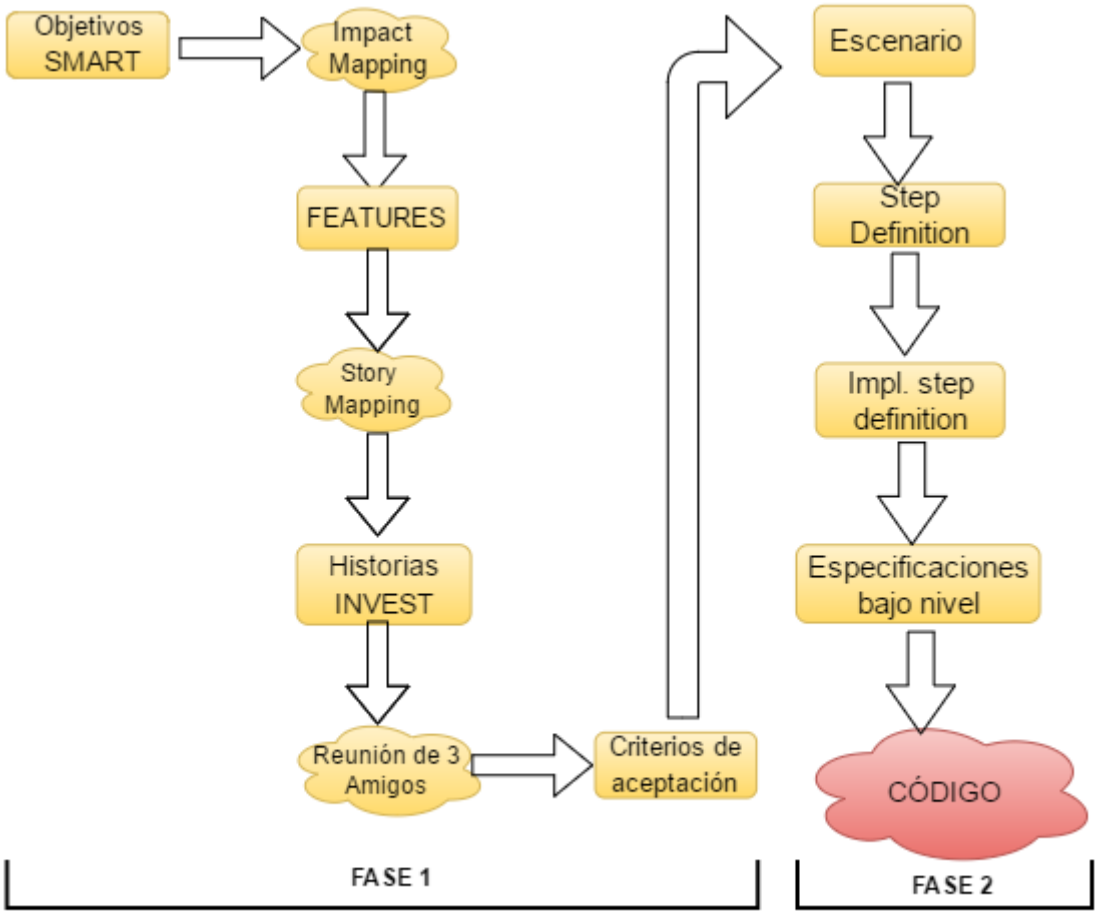


Figura 22. Fase 1 y 2

3.5 Fase 3 de Documentación

“The good code is his best documentation”, Steve McConnell.

3.5.1 Documentación viva

¿Qué es la documentación “viva”?

*Es documentación funcional disponible para **todos los stakeholders** de un proyecto que se nutre de las especificaciones automatizadas y se encuentra **actualizada** siempre y en **constante crecimiento**.*

Veamos rápidamente los puntos principales de esta definición:

- ***todos los stakeholders*** esto quiere decir que la documentación viva no solo es útil para el negocio que podrá tomar decisiones a partir de ella, sino que también es útil para todos los miembros del equipo de desarrollo y todos los stakeholder involucrados en el proyecto, cualquiera de ellos debe poder acceder a esta documentación.
- ***especificaciones automatizadas*** con esto hacemos referencia a las especificaciones de alto nivel y bajo nivel de las cuales venimos hablando durante el capítulo y que en la fase anterior nos encargamos de automatizar.
- ***actualizada*** sin lugar a dudas este punto es sumamente importante, la documentación debe estar actualizada, y que mejor que se encuentre actualizada a partir de las especificaciones ejecutables?
- ***constante crecimiento***. BDD al igual que las metodologías ágiles en las cuales se ha basado, es una práctica iterativa de desarrollo de software, la documentación viva crece en función de la incorporación de nuevas features, como así también de los cambios que se producen en la aplicación.

En este capítulo hemos recorrido un largo camino desde entender cuáles son los objetivos del negocio, pasando por convertir esos objetivos en features, para luego describir esas features en forma de escenarios, formando así las especificaciones ejecutables de alto nivel, quienes luego se comunicarán con las especificaciones de bajo nivel y finalmente con el código. En particular las especificaciones tanto de alto nivel como las de bajo nivel en la fase 2 describimos como y porque automatizarlas; las ejecuciones de las mismas generan reportes en la herramienta BDD que hayamos seleccionado. Los reportes combinan los resultados de todo lo que venimos trabajando y nos vuelven a nuestro punto de origen: las features. Estos reportes que combinan las especificaciones de alto y bajo nivel, junto con los resultados de las ejecuciones son lo que llamamos “*Documentación viva*”. Esta documentación es “*viva*” porque evoluciona y crece a incrementalmente a medida que incorporamos nuevas features a nuestra aplicación o realizamos cambios sobre la misma.

Los reportes no solo proveen un conjunto de resultados, que son obtenidos de la ejecución de las especificaciones, sino que:

- Describen **QUÉ** se espera que la aplicación haga.
- Reportan si es que la aplicación desempeña lo esperado de forma **correcta** o no.
- Ejemplifica **CÓMO** una feature se lleva a cabo desde la perspectiva del usuario.

Como podemos ver los reportes de BDD son el corazón mismo de la "*Documentación viva*", ya que de ellos obtenemos la información que nos indica no sólo el estado de avance de las features, sino que también como la aplicación está funcionando **como un todo**, en este momento.

Ventajas de la documentación viva

La "*Documentación viva*" nos provee de una serie de beneficios a saber:

- Una de las mayores ventajas de contar con una documentación "*viva*" es que el estado actual de la aplicación lo tenemos al alcance de la mano.
- Ayuda a los QA a identificar áreas en donde poner el foco para realizar pruebas de exploración.
- Ayuda a acelerar la incorporación de nuevos miembros al equipo de desarrollo, ya que no solo describe que hace la aplicación, sino que es posible también como lo hace actualmente. La documentación es ante todo una transferencia de conocimiento, un conocimiento que tiene que tener un valor y que debe ser posible transferirlo hacia distintas personas, tanto ahora y como en el futuro.
- Ayuda realizar cambios en el código con una mayor seguridad, ya que, si estos cambios modifican el comportamiento del producto, esto se refleja en la propia documentación.
- Los *stakeholders* del negocio puede revisar la documentación para asegurarse que describe el comportamiento deseado desde el punto de vista del negocio.

¿Porqué BDD cuadra con la documentación viva?

BDD ha demostrado que es posible tener una documentación precisa, siempre en armonía con el código, creando la especificación de manera más cuidadosa. BDD cuadra con la documentación viva ya que cumple por naturaleza misma de BDD con varios principios de este tipo de documentación (38):

- Conversaciones sobre documentación.

La herramienta primordial de BDD son las conversaciones que tienen lugar entre las personas, asegurando además que todos los roles están presentes, por ejemplo, en la reunión de 3 amigos.

- La audiencia a quien se dirige.

BDD se basa por sobretodo en entender el valor para el negocio, con lo cual la documentación está dirigida en primer lugar a la gente del negocio, y por lo tanto en un lenguaje no técnico.

- La idea de sedimentación.

Las conversaciones en general son suficientes, y no todo merece ser escrito. Solo los escenarios claves son los que se escriben y forman parte de la automatización.

- Documentos de texto plano.

Simplemente porque el texto plano es más conveniente que el hipertexto para manipular cambios y para un mejor mantenimiento con un mecanismo de control de versiones.

- Mecanismo de sincronización.

Porque el comportamiento del negocio describe tanto escenarios como la implementación del código, herramientas como Cucumber aseguran que ambos se encuentran sincronizados.

Una documentación viva: ECO y CCIL

La "*Documentación viva*" es mucho más que un conjunto de especificaciones ejecutables,para poder obtener los beneficios de una documentación viva, tenemos que organizar las especificaciones para que tengan sentido como conjunto y agregar información contextual que a su vez nos permita entender cada una de las partes individualmente.

Idealmente un sistema de *documentación viva* debe ayudarnos a entender que hace nuestra aplicación, lo que significa que tiene que ser **ECO** (1) (Ver figura 23):

- Fácil de entender ("*Easy to understand*")
- Consistent ("*Consistent*")
- Organizada para el fácil acceso ("*Organized for easy access*")

Para que nuestra documentación sea **fácil de entender** podemos:

- No crear especificaciones muy largas.

Esto es porque cuanto más larga sea la especificación más compleja será entenderla.

- No usar varias especificaciones pequeñas para describir una feature.

A medida que la aplicación evoluciona, también así nuestro entendimiento del dominio, suele suceder muchas veces que varias historias de usuario provenientes de una misma feature con el tiempo terminan siendo muy parecidas y lo conveniente entonces sea reorganizarse y solo dejar una que sea la que mejor represente la feature.

- Evitar usar conceptos técnicos de automatización en las especificaciones.

Las especificaciones descritas con un lenguaje técnicas son indicaciones herramientas de comunicación. Recordemos que BDD se basa en la comunicación y lo que buscamos es que todos los stakeholder pueden entender fácilmente las especificaciones, con lo cual deben estar escritas como mencionamos antes en un lenguaje Ubicuo.

Para que además la documentación sea consistente lo que se puede hacer es:

- Evolucionar el lenguaje oblicuo.

La evolución del lenguaje oblicuo ayuda a reducir el costo de mantenimiento de la automatización, porque reutiliza frases existentes para describir nuevas especificaciones.

- Definir el lenguaje de manera colaborativa.

Sobre este punto fue que hicimos énfasis en la fase 1 cuando plantemos las distintas reuniones (Story *planning*, *retrospective meeting*, etc.) que nos ayudan a trabajar de forma colaborativa en la definición de las especificaciones de nuestra aplicación.

Por último, para asegurarnos que la documentación se encuentra organizada de forma tal que es de fácil acceso podemos:

- Organizar el trabajo actual por historias de usuario
- Re-organizar las historias por áreas funcionales
- Organizar a través de las rutas de navegación de GUI
- Organizar por los procesos de negocio.

Figura 23. Documentación ECO.

Una documentación "*viva*" es también considerado actualmente como un conjunto de técnicas y principios para una documentación de alta calidad a bajo costo y que gira en torno a 4 principios o **CIRL** (38) (Ver Figura 24):

- Colaborativa ("*Collaborative*")
- Reveladora ("*Insightful*")
- Confiable ("*Reliable*")
- De bajo esfuerzo ("*Low-Effort*")

Una documentación **colaborativa** promueve la comunicación y por ende las conversaciones, estas tienen que ser interactivas, fluidas donde haya un intercambio de conocimiento eficiente.

Una documentación es **reveladora** porque brinda feedback que permite al negocio tomar mejores decisiones o visualizar situaciones que de otro modo no se evidencian.

Una documentación es **confiable** cuando se asegura que toda la documentación es precisa y en armonía con el código que se entrega en cualquier punto.

Una documentación es **de bajo esfuerzo** cuando la cantidad de trabajo que se debe realizar sobre la documentación cuando hay cambios, eliminaciones o incorporaciones es mínima.

Figura 24. Documentación CIRL.

3.5.2 Métricas

“Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted.”, Albert Einstein

¿Qué es una métrica?

*Una métrica es una **medición recurrente** que posee un poder informacional, diagnóstico, motivacional o predictivo de algún tipo.* Dave Nicolette.

Una métrica puede cumplir dos tipos de propósitos; por un lado, puede ser una métrica de conducción que se utiliza para determinar que tan bien se desarrolla el proyecto en post de los objetivos. Por otro lado, una métrica puede ser de mejora, y se utiliza para determinar si se obtiene una mejora en las entregas al cliente.
Es decir que una métrica puede ayudar a conducir el trabajo en progreso y/o a guiar las mejoras en progreso.

Las métricas también cumplen 3 diferentes funciones, pueden ser:

- Informativas. Nos **indican** que es lo que está ocurriendo.
- Diagnosticas. **Identifican** las áreas de mejora.
- Motivacionales. **Influencian** comportamiento.

Una métrica puede cumplir una o más de estas funciones.

Las métricas nos ayudan a medir cosas que ya han sucedido como también a medir o predecir cosas que pueden suceder en el futuro. Cualquier métrica que nos provea de información acerca de cosas que ya han sucedido es lo que se considera un indicador retrospectivo. Cualquier métrica que nos ayuda a predecir cómo suceden las cosas en el futuro es considerado un indicador principal. Usualmente un indicador principal se compone de una serie de indicadores retrospectivos junto con un cálculo de tendencia que sugiere cómo pueden suceder las cosas en el futuro, mientras las circunstancias se mantengan estables.
Las buenas métricas deberían:

- Ser lo suficientemente **precisas** para permitir que se tomen mejores decisiones.
- Permitir mejores acciones y mejoras importantes.
- **Motivar** al equipo
- Ser lo suficientemente **simple**
- Permitir la optimización

Métricas de conducción

Tengamos en cuenta con respecto a las métricas en general:

- Las métricas son utilizadas por el gerenciamiento del proyecto => es decir son gestionadas por roles como *Scrum Master.*
- Las métricas deben ser fácilmente entendibles por todos => debemos tener en cuenta que nuestro principal objetivo cuando usamos métricas es aportar al negocio datos fehacientes del trabajo que estamos haciendo y cómo lo estamos llevando a cabo.
- Más allá de quién sea el rol encargado de la gestión de las métricas, las mismas involucran siempre a todo el equipo => con lo cual es deseable que todo el equipo tenga conocimiento de las mismas.
- No es necesario implementar o llevar una métrica simplemente porque siempre se hizo o porque es recomendada, siempre hay que tener en cuenta si la métrica es útil para las características propias del proyecto=> Es decir puede ser la mejor métrica del mundo, pero sí existe un motivo por el cual no es útil para el equipo, entonces no es viable que se utilice.

Si bien existen múltiples y variadas métricas que pueden aplicarse a los distintos proyectos, se mencionan a continuación algunas especialmente seleccionada para ser utilizadas en coordinación con BDD.

Running tested features

RTF es el recuento del número de features actualmente desplegadas al ambiente de pruebas y que han pasado todas las pruebas automatizadas de manera exitosa:

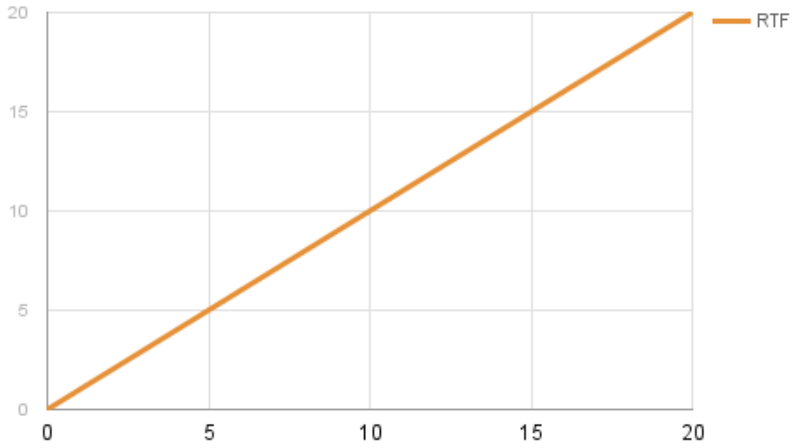
- Indica la **cantidad** de features planeadas de la solución están listas para producción.
- Indica si se crean **regresiones** cada vez que se incorporan nuevas features.
- Indica si es posible terminar suficiente **funcionalidad** en tiempo para proveer de suficiente valor al negocio que justifique continuar con el proyecto.
- Indica cuánto **tiempo** necesitaremos para completar un conjunto dado de features.

A lo largo del desarrollo el equipo realiza entregas incrementales que son partes de la solución (el software que estamos creando, la aplicación que será el producto final) en un ambiente donde se ejecutan regularmente pruebas automatizadas sobre las features entregadas.

Para poder entender mejor esta métrica veamos dos ejemplos: en el primero vamos a ver un **crecimiento lineal** esperado que indica que un proyecto se mantiene estable y saludable, en el segundo vamos a ver un **crecimiento no-lineal** que indica que surgió algún problema con las pruebas.

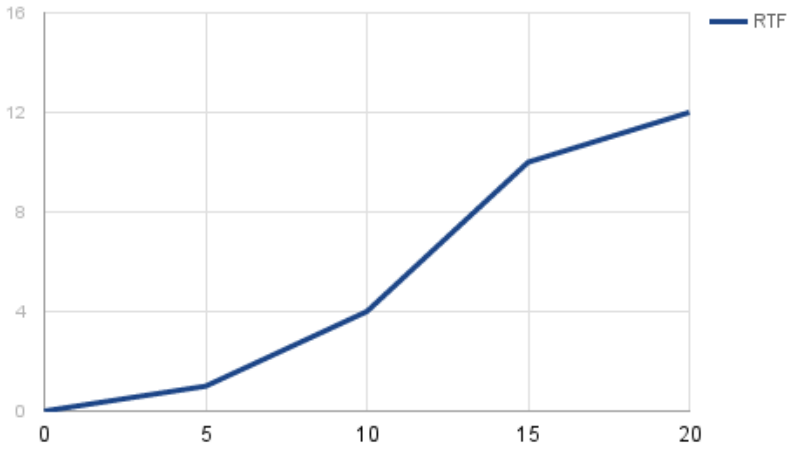
Veamos primero el ejemplo en la Figura 25 RTF Crecimiento Lineal (39), podemos ver en el mismo que en eje de las X señalamos los días del sprint (suponiendo que nos encontremos en un proyecto ágil) que también podríamos señalar con fechas exactas para mayor precisión. Mientras que en el eje de las Y señalamos la cantidad de historias de usuario que se despliegan en el ambiente seleccionado de pruebas y que satisfacen todas las pruebas automatizadas de manera exitosa. Este caso de crecimiento lineal nos indica que a medida que se introdujeron más features, las nuevas features y sus correspondientes pruebas y regresiones, no rompieron ninguna feature desarrollada con anterioridad y el proyecto se mantiene entonces en un crecimiento estable.

Figura 25. RTF Crecimiento Lineal



Ahora veamos el ejemplo dos en la figura 26 RTF Crecimiento No- Lineal (39), en este caso podemos ver que a diferencia del ejemplo anterior el crecimiento no es lineal en el tiempo, se pueden ver unos puntos de caída del eje de las Y, lo que significa que alguna feature que hasta días anteriores funcionaba correctamente, luego de la incorporación de un cambio o nueva feature, las pruebas dejaron de funcionar, esto puede suceder porque hubo algún cambio en la feature o porque no se actualizaron correctamente las pruebas automatizadas, de cualquier manera nos indica que la feature con la que contábamos ahora debe ser arreglada antes de continuar.

Figura 26. RTF Crecimiento No- Lineal



Algunos de los factores que contribuyen al crecimiento linear de RTF:

- Pruebas automatizadas
- Automatización de Builds
- Pruebas de aceptación automatizadas
- Integración continua
- Los desarrolladores crean pruebas unitarias
- Los QA crean pruebas de features
- Desarrolladores y QA trabajan juntos en la cobertura de pruebas unitarias

Burn charts

Es una proyección de cómo el equipo realizará entregas en el futuro basada en una medición empírica; muestra la cantidad de trabajo que se ha realizado, lo cual se conoce como **burn-up** chart o puede mostrar la cantidad de trabajo restante que se conoce con el nombre de **burn-down** chart.

Un burn chart:

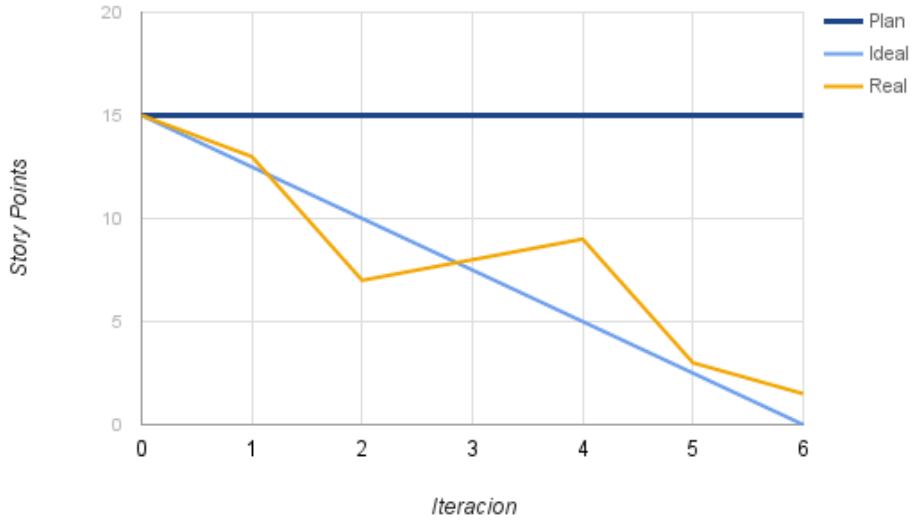
- Indica si el equipo cumplirá con los **objetivos de entrega** del producto.
- Indica cuánto tiempo necesita el equipo para completar el **alcance planeado**.
- Indica cuanto del alcance planeado el equipo puede completar para una **fecha determinada.**

Nos provee de un indicador principal del desempeño en cuanto a la entrega que tiene el equipo y a su vez también nos puede proveer de una advertencia temprana de posibles riesgos en las entregas.

Al igual que hicimos con RTF, veamos dos ejemplos de Burn chart para entenderlo mejor. En el primer ejemplo veremos un burn down chart, mientras que en el segundo veremos un burn up chart. Para ambos casos, sobre el eje de las x señalaremos las iteraciones que se pautaron (siempre teniendo en cuenta que nuestro proyecto sea ágil e implemente iteraciones o sprints, si otro fuese el caso podríamos simplemente señalar sobre este eje fechas precisas) mientras que en el eje de las y disponemos los *story points*[54] estimados para cada feature (esto también es suponiendo un proyecto ágil, si no fuese el caso podemos simplemente utilizar cantidad de horas) .

Veamos entonces la Figura 27 Burn Down Chart, donde podemos ver 3 cosas a describir: en primer lugar, el plan acordado con el negocio, este indica lo esperado. Por otro lado, tenemos una línea punteada que es el ideal esperado, en este caso que las features disminuya linealmente a medida que se acerca la fecha final, ya que recordemos el burn down nos muestra la **cantidad de trabajo que resta** del día actual hasta el día pautado, lo que se espera es que a medida que llegue el día pautado esa cantidad disminuye. Por último, tenemos la línea que indica el decremento real, es decir lo que el equipo realmente está haciendo con respecto a lo esperado, esta curva nos permite conocer si existen desviaciones lo antes posible, para poder tanto recordar fechas o tomar alguna otra medida que permita al equipo llegar con lo pautado al día acordado.

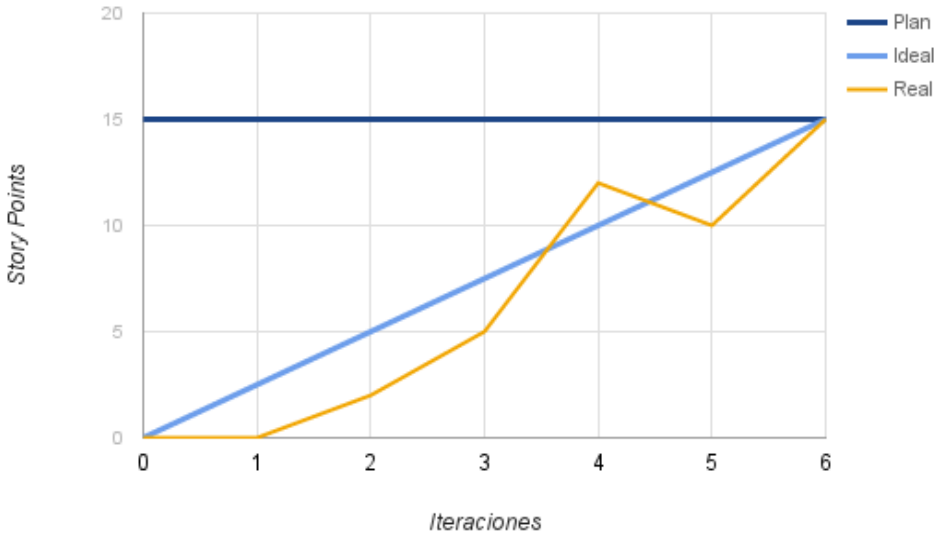
Figura 27. Burn Down Chart.



[54]"*Un Story Point es la medida de esfuerzo mediante la cual un equipo evalúa la complejidad, el tamaño, el nivel de incertidumbre y el riesgo de realizar una historia en base a las destrezas, conocimiento y la información actual que poseen los miembros que conforman ese equipo, entendiendo que esfuerzo no es igual a tiempo"*
https://johnnyordonez.com/2014/11/25/que-mismo-es-un-story-point/

Vayamos ahora a la Figura 28 Burn Up chart, sobre este también veremos 3 cosas a mencionar, el primer lugar el plan acordado también se dibuja en este gráfico, en segundo lugar, la línea punteada que indica el ideal también está presente. Por último, lo que diferencia a este gráfico del anterior es la dirección y el significado de la línea real ya que el burn up nos muestra la **cantidad de trabajo realizado**, y nos muestra también cuál es el restante, para llegar al objetivo.

Figura 28. Burn Up Chart



Para que realmente funcione, debe haber un entendimiento consistente acerca de qué constituye un ítem de trabajo y debe haber una definición explícita, demostrable de que significa que un ítem esté "*completo*".

Cumulative flow

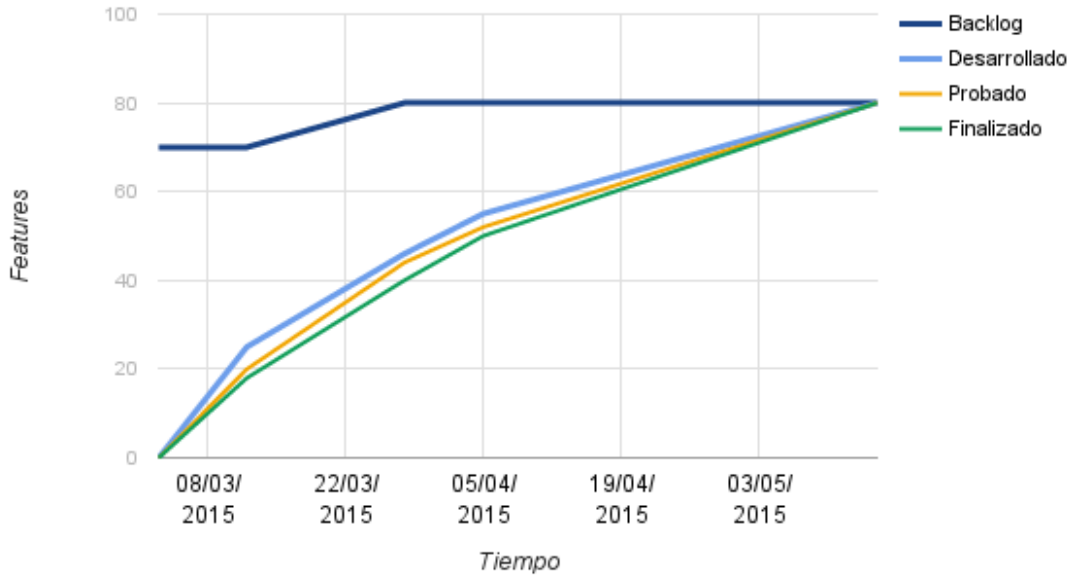
Cumulative Flow o CFD es una representación visual de todo el trabajo realizado y en proceso a la fecha. Expone las oportunidades de mejora del proceso y cuáles son los ítems entregados a simple vista, además:

- Indica dónde se encuentran los **cuellos de botella** en el proceso.
- Indica en qué puntos es posible realizar **entregas parciales**.
- Indica dónde se encuentran los componentes más grandes
- Indica en qué punto la carga de trabajo se encuentra **desbalanceada**.

Veamos dos ejemplos simples de CFD, en uno veremos un gráfico de un caso ideal, mientras que el segundo veremos un ejemplo de un equipo que no utiliza entrega continua. Como podemos ver en la Figura 29 CFD Caso Ideal (40), sobre el eje de las x disponemos las fechas establecidas con el negocio, mientras que en el eje de las y señalamos las features comprometidas. Sobre el mismo gráfico también marcamos el *backlog*[55] (nuevamente estamos suponiendo en este punto que el proyecto es ágil y cuenta con un *backlog* establecido) lo señalamos en el gráfico por un motivo principal: se puede modificar a lo largo del desarrollo, es decir es posible que se incorporen nuevas features o que alguna sea eliminada. Por otro lado, tenemos 3 líneas indicadoras: la primera nos indica lo que ha sido **desarrollado**, la segunda lo que ha sido **probado**, y la tercera lo que se considera que está **terminado**. Para poder dibujarlas es imprescindible que contemos con un tablero que nos indique en qué estado está cada feature (en el *backlog*, desarrollada, probada, terminada) para poder establecer el número y dibujar la línea en el gráfico.

En el primer ejemplo podemos ver que la línea de desarrollo comienza a crecer apenas comienza el gráfico, mientras que la línea de pruebas se mantiene en 0 hasta que luego comienza, esto se debe a que hasta que no haya algo desarrollado y desplegado en un ambiente de pruebas, no es posible probarlo. Por último, lo mismo sucede con la línea de terminado, al comienzo se mantiene en 0 hasta que luego de aprobadas las features cumplen con lo estipulado como criterio de finalizado y recién ahí es posible contabilizar dichas features.

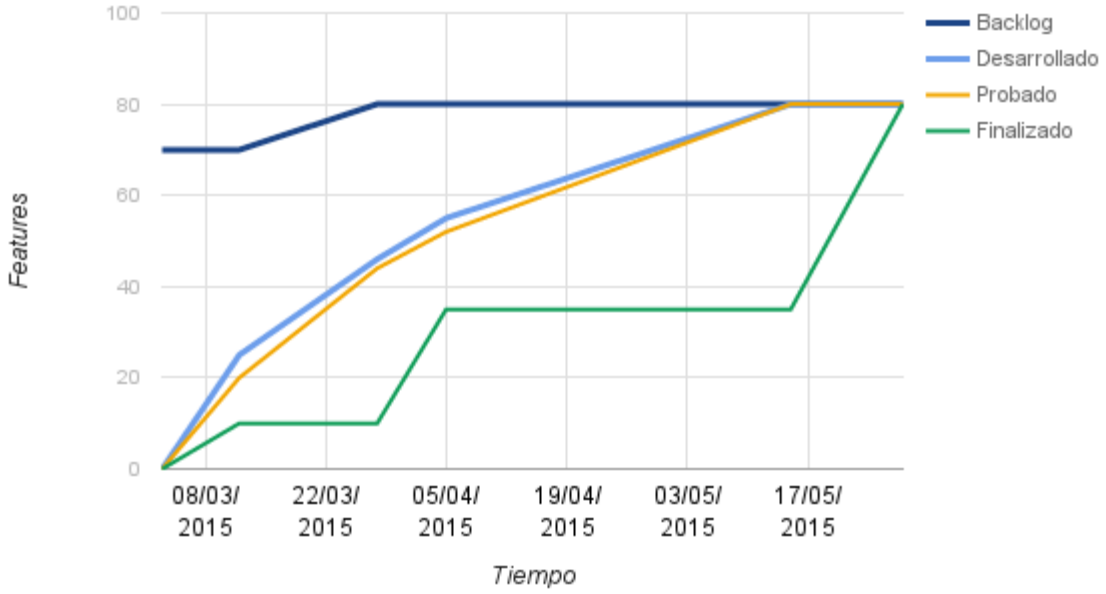
Figura 29 CFD Caso Ideal



[55] *"Es el instrumento metodológico del marco de trabajo Scrum, que se usa para listar las características (Features) o funcionalidades del software a desarrollar, para priorizarlas de acuerdo a las necesidades del área de negocio"*
http://www.pmoinformatica.com/2013/11/plantillas-scrum-pila-producto-product.html

En el segundo ejemplo en la Figura 30 CFD Ejemplo sin integración continua, vemos que la diferencia con el primero es simplemente la línea de finalizado que se encuentra en forma de escalera, esto se debe a que hasta que no se realiza una entrega en algún ambiente y puede darse la feature como finalizada, la curva no crece. Claramente esta situación puede mejorarse utilizando integración continua, práctica que explicamos en la fase anterior.

Figura 30 CFD Ejemplo sin entrega continua



Métricas de mejora

Velocity

Esta métrica mide la cantidad de trabajo completado durante una iteración, ayudando a reducir las variaciones y mejorar la predictibilidad del planeamiento para las iteraciones; asimismo asegura que las entregas realizadas en cada iteración incrementan el producto final y maximizan el valor de negocio. Además:

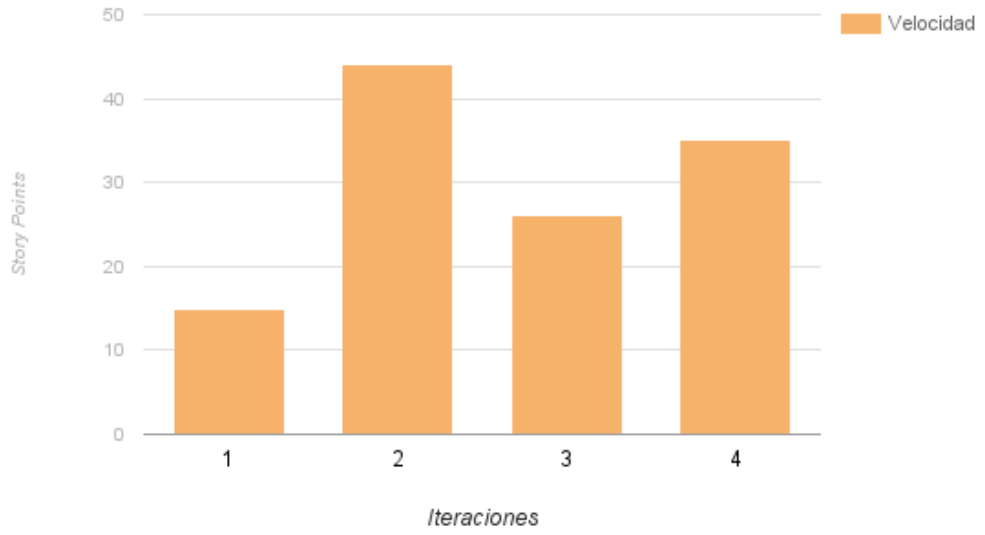
- Indica si las entregas del equipo para una iteración son **incrementales**
- Indica si el índice de entregas de un equipo se mantiene **consistente** y **predecible**.

Esta métrica es utilizada típicamente como base para predecir el cumplimiento de entregas que tendrá un equipo en el futuro cercano basado en una observación empírica del cumplimiento en el pasado reciente. Provee una base de predicciones para planificaciones a corto plazo, porque se basa en observaciones del cumplimiento actual del equipo en lugar de estimaciones o promesas. Una velocidad consistente de un equipo indica que se planifica y se ejecuta el trabajo correctamente.

Veamos cómo se puede graficar esta métrica con un par de ejemplos. Veremos en los ejemplos dos casos distintos, en el primero un equipo con una velocidad errática y en el segundo un equipo con una velocidad constante. En primer lugar, explicaremos que en el eje de las x dispondremos las iteraciones del proyecto, mientras que sobre el eje de las y los story points de las features que se entregan en cada iteración. Veremos entonces que se generan barras con la velocidad del equipo para cada iteración, lo cual en la jerga se conoce como "*cantidad de puntos quemados por iteración*".

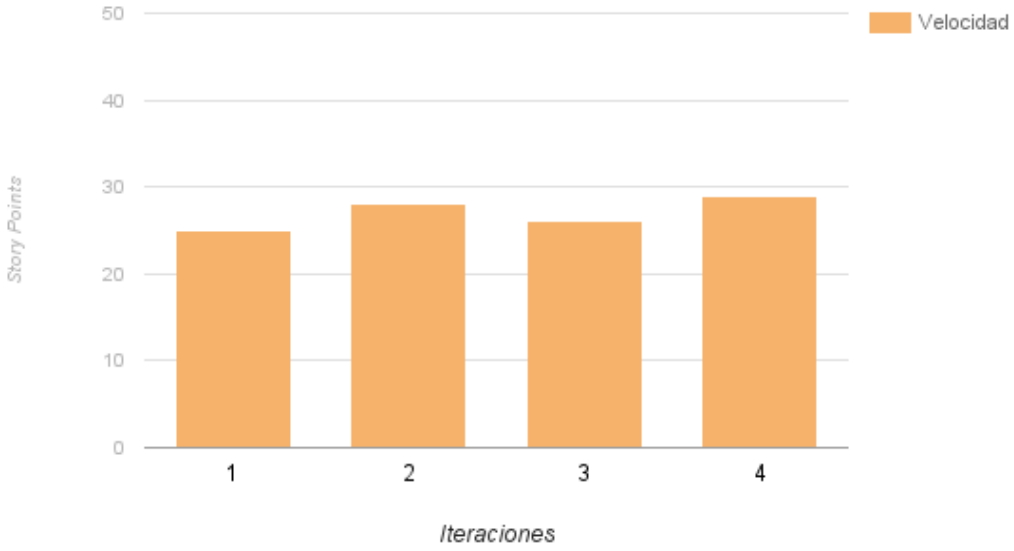
Vayamos entonces a la Figura 31 Velocidad Errática. Podemos ver a simple vista que la velocidad del equipo vario bastante entre las distintas iteraciones, esto puede darse por varios motivos, uno de ellos es que en una iteración se cuente con 5 miembros y a la siguiente iteración por algún motivo queden 3 miembros, la velocidad claramente se verá afectada. Otro motivo de esta variación entre iteraciones es que nuevos miembros se incorporen, con lo cual la curva de aprendizaje de los nuevos, detiene en parte a los miembros anteriores del equipo que debe tomarse el tiempo de explicarles cómo funcionan las cosas.

Figura 31. Velocidad errática



Un caso diferente es el que vemos en la Figura 32 Velocidad Constante. Si bien hay una diferencia pequeña entre la primera iteración y la segunda (usualmente la primera iteración de un equipo se denomina "*Iteración 0*" y se la toma como referencia inicial) es lo que se considera el punto de estabilización, ya que podemos ver que luego en el resto de las iteraciones la velocidad se mantiene constante. Esto nos indica varias cosas, por un lado, que la cantidad de trabajo y entrega al negocio se mantiene, por otro lado, que los problemas que surjan sí es que surgen son absorbidos por el equipo y se llega a los plazos sin modificaciones.

Figura 32 Velocidad constante



Como en la mayoría de las métricas, sobre todo aquellas que se encuentran directamente relacionadas con los miembros del equipo, hay factores que pueden influenciar la velocidad del equipo:

- Impedimentos
- Motivación
- Conocimiento del negocio
- Ambiente de desarrollo
- Transparencia del proyecto
- Objetivos del proyecto

Niko Niko calendar

Esta métrica Indica cómo el estado emocional del equipo cambia a lo largo del tiempo. Basado en un simple chequeo diario de cada miembro del equipo, es un calendario que registra el estado de ánimo usando 3 estados: **positivo**, **neutral**, **negativo**. Lo que nos permite es levantar una advertencia de posibles problemas sistémicos que afectan la moral del equipo, puede también brindar una advertencia de posibles problemas del proceso o técnicos que puedan darse, ya que una baja moral usualmente conlleva a otros problemas. Tal vez el punto más importante a resaltar es que esta métrica es para el equipo y por el equipo, tal como mencionamos en la fase de comunicación, esta métrica nos brinda un feedback que nos permite tomar decisiones a nivel de equipo que pueden influenciar el trabajo que se realiza y las entregas al cliente. Todos los miembros del equipo deben **voluntariamente** participar, de manera que la información sea fidedigna y se pueden obtener datos reales.

Veamos un par de ejemplos de cómo un calendario Niko Niko nos revela información del equipo. Como ya mencionamos esta métrica se obtiene en base a un calendario que diariamente los miembros del equipo completan con un estado, con lo cual nuestro gráfico tendrá por un lado en el eje x los días del año y en el eje x las personas que forman parte del equipo.

En la Figura 33 Calendario Niko Niko- El campista feliz, en este caso vemos que hay un miembro del equipo que increíblemente todos los días se siente positivo, esto podría estar acertado simplemente porque esa persona tiene un buen humor siempre, o podría estar indicando otra cosa: podría ser que para contentar al resto en lugar de completar el calendario con su sentir actual esa persona lo complete con un valor que no es real.

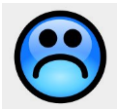
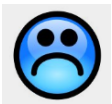
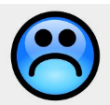
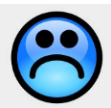
	Lunes	Martes	Miércoles	Jueves	Viernes
Simón					
Juana					
Pedro					
Apolo					

Figura 33 Niko Niko Calendar -El Campista Feliz

En cambio, en la Figura 34 Calendario Niko Niko- Estándar, vemos que la mayoría de los miembros la mayor parte del tiempo se sienten positivos, esto nos indica por un lado que están satisfechos con el trabajo que vienen realizando y que además no hay demoras o inconvenientes mayores en el desarrollo del producto. A menudo si los estados negativos coinciden esto puede deberse a que hubo algún inconveniente ya sea personal entre esos dos miembros o puede haber sido por situaciones personales o relacionadas con el proyecto.

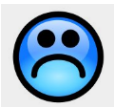
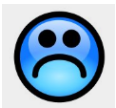
	Lunes	Martes	Miércoles	Jueves	Viernes
Simón					
Juana					
Pedro					
Apolo					

Figura 34 Niko Niko Calendar –Estándar.

La lectura que realicemos del calendario Niko Niko dependerá en 80% del equipo mismo y cuanto lo conozcamos para obtener información de valor del mismo.

3.5.3 Backlog Digital

En proyectos ágiles, los tableros de tareas son frecuentemente usados para llevar un registro de la actividad del proyecto. Un tablero suele ser una *pizarra física o una pared* que contiene una lista de historias de usuario, tareas, defectos, y otras actividades que el equipo debe realizar para completar el proyecto.

La disposición exacta del tablero de tareas es muy flexible y es en general diferente para cada equipo y proyecto, pero el principio general siempre es el mismo. Durante la planificación inicial, el trabajo se divide en historias de usuario (en nuestro caso en features, que como mencionamos antes no necesariamente se relaciones uno a uno con una historia de usuario, ya que una feature puede contener a más de una historia de usuario), tareas, etc. que deben ser resueltas en la iteración y entregadas a su finalización. Cada tarea u historia se dispone en una columna del tablero basado en su estado actual (no comenzada, en curso, terminada, estos estados también dependen de lo que estipule el equipo, ya que por ejemplo algunos equipos solo utilizan: none, in progress, done. Mientras que otros equipos pueden llamarlo de otro modo o utilizar más cantidad de estados). Podemos ver gráficamente un ejemplo en la Figura 35.

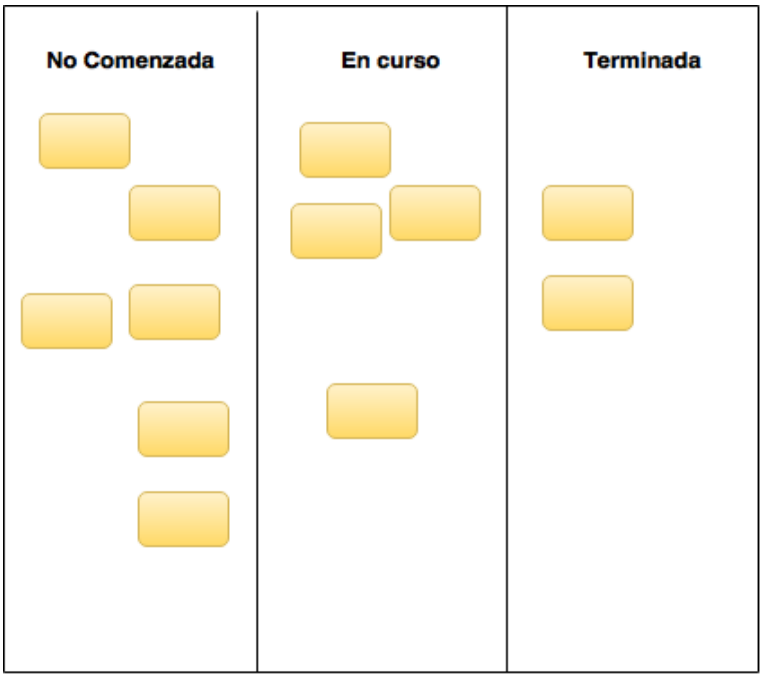


Figura 35. Tablero de tareas.

Todos los días los miembros del equipo durante la **reunión diaria** que explicamos en la fase 1, brindan su estado al resto del equipo y el tablero se actualiza según corresponda. La gran ventaja de este tipo de formatos es que todo el equipo tiene un vistazo general de lo que el equipo como un todo está trabajando, lo cual hace más sencilla la coordinación del equipo.

La principal desventaja de este formato es que para obtener métricas es necesario que algún miembro del equipo vuelque esa misma información en otro medio, además siempre y cuando todos los miembros del equipo se encuentren en el mismo espacio físico es sencillo llevar a cabo la utilización de un tablero de tareas físico, pero hoy en día los equipos se encuentran en su gran mayoría **distribuidos** geográficamente, lo cual hace más difícil esta comunicación.

Debido a estas desventajas es que muchos equipos toman la decisión de llevar un registro del trabajo que se realiza en alguna herramienta de gestión de proyectos que permita esta función, se le suele llamar a eso backlog digital. La gran ventaja que tiene es que es posible generar por ejemplo gráficos **burn down** (los cuales explicamos en la sección anterior) automáticamente a partir del input que se genera diariamente sobre los tableros de tareas. Pero ahí no terminan los beneficios que tiene utilizar una herramienta que permite tener un backlog digital, además es podemos integrar con alguna herramienta de BDD (como por ejemplo Cucumber). Esto nos ayudará por ejemplo a no duplicar información ya que la definición de una historia de usuario puede estar definida en la herramienta de backlog digital y la herramienta que seleccionemos de BDD simplemente la tomará de allí.
A continuación, mencionaremos algunas herramientas de gestión de proyectos que permiten llevar un backlog digital, así como tablero de tareas virtual: **Jira**[56], **VersionOne**[57], **ScrumDo**[58].

[56] https://www.atlassian.com/software/jira
[57] https://www.versionone.com/
[58] http://www.scrumdo.com/

3.5.4 Resumen Fase 3

En esta fase hablamos principalmente de 3 cosas: la documentación viva, las métricas y el backlog digital. En cuanto a la primera, nos detuvimos a explicar un poco qué es la **documentación viva**, mencionamos sus ventajas y porque BDD es una metodología que cuadra muy bien con la documentación viva. Además, detallamos como una documentación viva debe ser **ECO** y **CCIL**, especificando todas las características deseables.

Luego hablamos de las **métricas**, describimos en primer punto que es una métrica y luego detallamos 5 métricas recomendadas 3 relacionadas con las features (**Running Tested Features, Burn Charts, Cumulative Flow**) y 2 relacionadas con el bienestar del equipo (**Velocity, Niko Niko Calendar**). De Cada una de ellas se dio una breve explicación y se detallaron 2 ejemplos contrapuestos para comprenderla mejor.

Por último, en esta fase hablamos de lo que es un **backlog digital** y de cómo las herramientas de backlog digital y de BDD se pueden integrar para generar mejores reportes, que forman parte de la documentación viva, como explicamos antes.

En la Figura 36, podemos ver en conjunto como todas las herramientas que hemos hecho referencia a lo largo del capítulo forman parte activa de la documentación viva, por un lado, las herramientas de BDD que seleccionemos para realizar nuestras features, como explicamos en la Fase 1 y en parte de la Fase 2 al automatizar las especificaciones. Por otro lado, las herramientas relacionadas a la integración y entrega continua como detallamos en la Fase 2 cuando hablamos de automatización del proceso de build. Por último, las herramientas que permiten tener un backlog digital y que son denominadas herramientas ágiles.

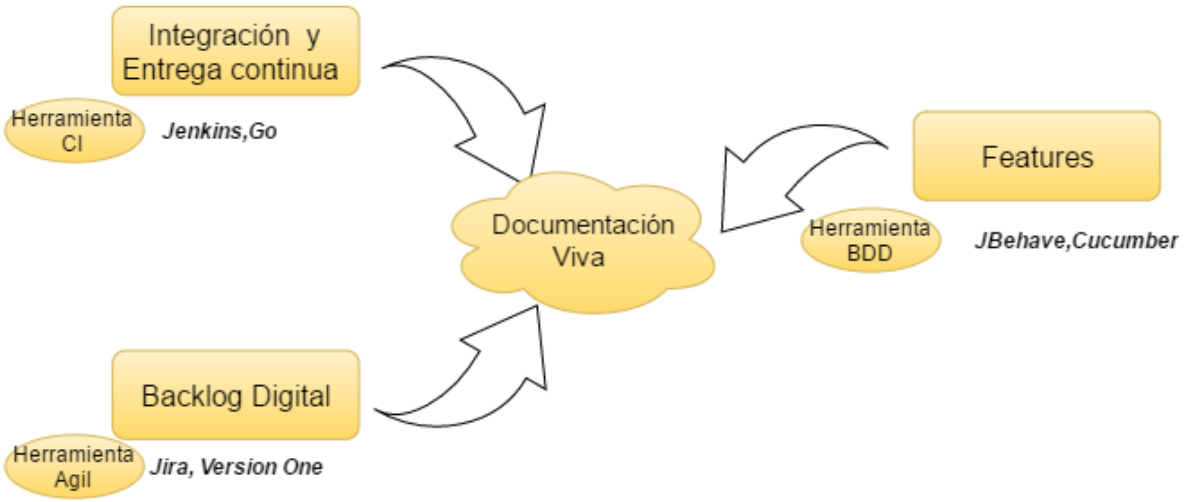


Figura 36. Documentación Viva

3.6 Resumen del capítulo

En este capítulo desarrollamos la guía práctica propuesta en detalle, describiendo en primer lugar un cuestionario previo que nos incentiva a analizar los motivos por los cuales queríamos llevar a la práctica la guía en un proyecto. Luego nos adentramos en cada una de las fases, a modo de resumen se enumeran a continuación los puntos relevantes de cada una de ellas:

- Fase 1- **Comunicación**
  - Entender el negocio. Impact mapping. Objetivos SMART.
  - Story planning. Reunión de 3 amigos. Historias de usuario INVEST.
  - Daily Meeting. Sprint Review. Retrospective Meeting.
- Fase 2- **Automatización**
  - Automatización especificaciones de alto nivel. Scenarios. Step definitions. Implementación de steps definitions.
  - Automatización de especificaciones de bajo nivel. Pruebas unitarias.
  - Integración continua. Entrega Continua. Despliegue Continuo.
- Fase 3 - **Documentación**
  - Documentación viva.
  - Metrics. Running Tested Features. Burn Chart. Cumulative Flow. Velocity. Niko Niko Calendar.
  - Backlog Digital.

Capítulo 4. Caso de estudio (Simulación)

En el desarrollo de este capítulo veremos cuál es nuestro caso de estudio[59] y cuáles son sus características particulares, siendo de mayor importancia aquellas que lo hacen un proyecto semi- ágil.

Además, simularemos la puesta en práctica de la guía propuesta en el capítulo 3 y hablaremos acerca de los resultados obtenidos de la simulación.

4.1 Descripción del caso de estudio

Para describir nuestro caso de estudio, partiremos dando un breve resumen de las características generales de la empresa a la cual pertenece.
A modo de simplificar para futuras menciones, llamaremos a la empresa en la cual nuestro proyecto se desarrolla: **ATLab**.

ATLab es una empresa que se dedica al e-commerce y que cuenta con variados proyectos en los cuales se realiza tanto mantenimiento de funcionalidad actual, se modifica e incorpora nueva funcionalidad y también se realiza re-arquitectura de código heredado.

ATLab es una empresa multinacional con lo cual es muy común que sus equipos se encuentren **distribuidos** a lo largo de distintos países y distintas zonas horarias, lo que suele a menudo complejizar la comunicación.

ATLab cuenta con una herramienta desarrollada internamente que permite la **automatización de pruebas**, la herramienta utiliza Selenium WebDriver[60] y se encuentra desarrollada en lenguaje JAVA. Además, se cuenta con **4 ambientes:** el primero es para desarrollo y es utilizado por los desarrolladores únicamente, el segundo es un ambiente de pruebas en donde los QA realizan sus verificaciones, el tercero es un ambiente similar a producción que se utiliza para realizar pruebas específicas y reproducir errores que se den en producción y por último el ambiente productivo que da soporte a la plataforma de e-commerce.

Actualmente cada proyecto y, por ende, cada equipo dentro de un proyecto sigue un patrón ***Waterfall*** para el desarrollo: se realiza el análisis de requerimientos, luego se desarrollan y finalmente se prueban para ser entregados al ambiente productivo. Además, todos los meses se cuenta con un esquema de entregas (denominado "*Release Milestone*") fijo a nivel corporativo para todos los proyectos y desarrollos vigentes, que define las fechas en las cuales se debe finalizar el análisis, cuando se debe finalizar el desarrollo, cuando se deben realizar las pruebas y cuando se realiza la entrega productiva.

[59]Cabe aclarar que el caso de estudio está basado en un proyecto real, por cuestiones de confidencialidad, todos los nombres han sido adaptados y el alcance fue acotado para los fines de este trabajo.
[60]Es una de las herramientas parte del conjunto de herramientas provistas por Selenium que permite crear pruebas automatizadas para navegadores web. http://www.seleniumhq.org/projects/webdriver/

Todos los proyectos de ATLab además cuentan con el uso de una herramienta licenciada de Backlog digital, una herramienta de documentación, y además una herramienta de Integración continua en donde se administran los ambientes.

Vayamos entonces a la descripción del proyecto dentro de ATLab que será nuestro caso de estudio:

- El proyecto en cuestión se tratará de un cambio de arquitectura sobre funcionalidad existente. El cambio es desde una arquitectura web con Base de Datos Oracle hacia una arquitectura moderna con servicio Rest[61] y base de datos distribuida. La funcionalidad en particular se refiere a los códigos de acceso y seriales para productos digitales que son otorgados ante la compra de uno o más productos digitales.
- El equipo cuenta con **7 miembros**: 1 líder técnico, 4 desarrolladores, 2 QA. Además, cuenta con la participación momentánea de 1 BA (momentánea porque solo participa cuando se requiere aclarar algún aspecto de la re - arquitectura planteada)
- El equipo se encuentra **distribuido** en 2 locaciones distintas.

4.1.1 Por qué es un desarrollo semi-ágil?

ATLab es una empresa que se auto proclama como una empresa "*Ágil*" o que sigue corrientes y prácticas ágiles. Esto se debe al hecho de que cierto conjunto de prácticas como por ejemplo las reuniones diarias han sido implementadas. Sin embargo, veremos que no son aplicadas de la manera que se esperaría o en la manera correcta.

Si hacemos un repaso de las características que mencionamos de ATLab podemos destacar que:

- Cuenta con una herramienta privada de automatización de pruebas.
- Tiene una licencia para una herramienta de backlog digital.
- Posee 4 ambientes: Desarrollo, Testing, Integración, Producción.
- Utiliza una herramienta de integración continua.
- Aplica algunas prácticas ágiles.

¿Pero entonces por qué decimos que es un desarrollo semi-ágil sí todo lo mencionado en el párrafo anterior parece cumplir con un desarrollo ágil? Principalmente por estos puntos:

- Existen prácticas Ágiles que se utilizan en el desarrollo y de ahí el término de semi-ágil.
- No todas las practicas Agiles implementadas, de la forma en la que se llevan a cabo son por definición prácticas ágiles que promuevan lo declarado en el Manifiesto Ágil. Algunas son, malas interpretaciones que solo conservan el nombre y no tanto así el concepto de lo que involucra.
- Aún se mantiene el proceso *Waterfall* para la entrega de funcionalidad. Existe un esquema estricto de fechas de entrega, lo que contradice a la flexibilidad planteada por las metodologías ágiles. ("*Release Milestone*")
- Se utilizan herramientas ágiles (para gestionar el backlog, para automatizar ciertas pruebas, para permitir integración continua) pero no se comunican entre sí, lo cual

[61] "REST *(Representational State Transfer) es un estilo de arquitectura para desarrollar servicios.*" http://www.arquitecturajava.com/servicios-rest/

hace que la información y por tanto la documentación se encuentra fraccionada y repetida en algunas ocasiones.

Existe una mínima diferencia entre un desarrollo ágil y uno semi-ágil. Las metodologías ágiles plantean buenas prácticas que cada empresa, proyecto, equipo debe adaptar a las necesidades particulares que se presenten. Esto sigue valiendo para las semi-ágiles; sin embargo, basándonos en la experiencia adquirida en el mercado laboral actual de desarrollo de sistemas en una empresa multinacional y además considerando el marco teórico adquirido durante la elaboración del presente trabajo, brindamos la siguiente definición de desarrollo semi-ágil:

*Un **desarrollo semi-ágil** es aquel que mantiene procesos y metodologías estructuradas junto con la utilización de un número reducido de buenas prácticas recomendadas por las metodologías ágiles, pero en donde la implementación de las mismas no se corresponde con la filosofía u objetivos para los cuales fueron creadas.*

Basándonos en esta definición es que podemos decir que nuestro caso de estudio sigue un ***desarrollo semi-ágil***.

4.1.2 Problemas actuales

En ATLab como ya bien mencionamos, aún se mantiene el proceso *Waterfall*, mediante el cual la comunicación entre desarrolladores y QA se comienza a entablar recién cuando el desarrollador ha terminado su tarea y es el QA que debe comenzar a crear sus pruebas para verificar la funcionalidad. Este punto de partida no es eficiente, ya que genera un claro desfasaje entre ambos miembros de un mismo equipo, el desarrollador ya ha terminado su labor con respecto a la tarea o podríamos decir ya ha hecho su parte, mientras que el QA recién comienza, lo que hace que el trabajo entre ambos no se realice de manera colaborativa ni en paralelo.

Sumado a lo anterior y en parte como consecuencia, no se cuenta con un tiempo exclusivo dedicado a la resolución de problemas (o como se suele llamar en la jerga informática: *bug fixing time*) sino que una vez que termina la implementación del código el desarrollador toma otra historia para continuar trabajando. El problema de esto es que mientras los desarrolladores continúan el desarrollo de nuevas historias, también deben solucionar cualquier problema o defecto que los QA encuentren durante la ejecución de sus pruebas. Este cambio constante que deben hacer los desarrolladores entre lo que ya han implementado y lo que se está actualmente implementando propicia aún más los errores tanto en la historia desarrollada con anterioridad como en la actual.

ATLab lleva más de 7 años en el mercado y su código heredado no es sólo antiguo, sino que su documentación suele no estar actualizada correctamente para todas y cada una de las funcionalidades y servicios que brinda. A lo largo de los años algunas de las personas que participaron de distintos proyectos ya no trabajan más en la empresa y como la documentación es escasa, cierto conocimiento del negocio y de cómo fue implementado el mismo, de decisiones de diseño, se perdió con la partida de dichas personas, siendo la

única forma de generar esta documentación realizar ingeniería inversa[62] del código existente.
Por otro lado, hace 2 años se decidió introducir algunas prácticas de Scrum y fue entonces cuando se decidió crear una herramienta propia de automatización de pruebas, la idea surgió de un grupo de ingenieros QA que crearon una versión beta en 2 meses, esa misma idea se propagó por las distintas áreas de la empresa, con lo cual cada área se encarga hoy en día de crear, corregir y mantener un conjunto de pruebas automatizadas con esta herramienta, que se ejecuta diariamente.
Relacionado también a la incorporación de prácticas Scrum es que se decidió hacer uso de una herramienta de backlog digital. Esta decisión estuvo principalmente basada en tratar de reducir la brecha de comunicación entre las diferentes locaciones (ya que algunas locaciones se encuentran no solo en distintas partes del mundo, sino que también en diferentes zonas horarias).

¿Cuáles son los **problemas actuales** en el desarrollo entonces? Explicaremos brevemente los que resultan de interés para el presente trabajo:

Uso incorrecto de SCRUM
Si bien como se comentó anteriormente se incorporó la utilización de la metodología SCRUM, esta solo se implementa siguiendo los protocolos de nombre de reuniones y sin cumplir con el motivo para el cual han sido pensadas. Por ejemplo: realizar una reunión diaria en donde en lugar de describir simplemente el estado actual de cada miembro del equipo se utiliza ese tiempo para discusiones técnicas, haciendo que dicha reunión se extienda más de 25 minutos.

Ausencia de trabajo colaborativo/paralelo entre los distintos roles.
Dado que aún se lleva a cabo un esquema similar a Waterfall para las entregas, el trabajo de una persona termina siendo el punto de partida del siguiente. Por ejemplo, dado un requerimiento el BA lo procesa y obtiene una historia de usuario, dicha historia de usuario es entregada al DEV quien implementará su solución, dicha solución será aprobada por el QA, una vez aprobada por el QA se entregará al cliente la funcionalidad abarcada por dicha historia de usuario.

Mala comunicación entre los roles.
Relacionado con el punto anterior, dado que cada uno de los roles trabaja de manera aislada, la comunicación existe, pero no es la adecuada, principalmente porque al dar por terminada o casi terminada su tarea, la comunicación suele ser negativa. Por ejemplo, en el caso del DEV con el QA, cuando el QA se pone en contacto con el DEV es porque encontró algo que no funciona como es lo esperado y por supuesto que esto denota un punto negativo para el DEV ya sea que ha cometido un error o existe un error acarreado desde la definición de la historia de usuario.

[62] *"(...) es el proceso de descubrir los principios tecnológicos de un objeto, herramienta, dispositivo o sistema, mediante el razonamiento abductivo (haciendo conjeturas) de su estructura, función y operación*."
http://blogingenieria.com/general/ingenieria-inversa/

Falta de tiempo exclusivo de "*bug fixing*"

Así como explicamos antes un DEV en el mismo periodo de tiempo que se encuentra implementando una nueva historia, debe también solucionar los errores que un QA encuentre sobre lo implementado con anterioridad. Siendo así y dado el hecho que la mayoría de los errores no se encuentran tempranamente, sino sobre el final y cerca a la fecha de entrega, este es un punto de inflexión y de máxima carga de trabajo para los desarrolladores.

Falta de feedback con respecto al bienestar del equipo.

Si bien cada equipo es libre de optar si realiza reuniones tales como la retrospectiva, realizar un debido seguimiento para detectar malestar en el trabajo diario del equipo es una práctica no fomentada ni tampoco practicada, lo cual hace muy difícil detectar posibles problemas de satisfacción en el equipo y descontentos individuales respecto al trabajo diario, descontentos que además tienen un impacto en el trabajo realizado.

Documentación escasa y no actualizada

Como bien se sabe la informática no es un campo en donde abunde la documentación escrita, ya que lo primordial es que "*funcione*" y bajo ese lema es que en nuestro caso de estudio la documentación de la funcionalidad es muchas veces escasa y otras tanta no se encuentra actualizada, lo cual hace muy difícil su mantenimiento y seguimiento.

Falta de integración entre herramientas

Relacionado al punto anterior de la documentación, no se cuenta actualmente con una integración entre la herramienta de backlog digital utilizada y la herramienta de integración continua. Esto lo que provoca es, por un lado, información no relacionada, ya que no hay manera de relacionar cierto código con una historia en particular y, por otro lado, genera inconsistencias e información repetida.

Las pruebas automatizadas no son confiables.

Debido en parte a su falta de mantenimiento constante y a la inestabilidad del ambiente de pruebas que tiene multipropósito (ya que sirve de ambiente de pruebas tanto para DEV y QA de todas las áreas), las pruebas que existen no son confiables, al menos no tal como deberían serlo. Muchas veces encontrar el motivo por el cual falla una prueba conlleva una elevada cantidad de tiempo, para al final de cuentas enfrentarse al mismo motivo: la prueba está desactualizada.

4.2 Puesta en práctica de la guía (Simulación)

"We are what we repeatedly do. Excellence then, is not an act, but a habit." , Aristoteles.

Dentro de esta sección realizaremos la simulación de la puesta en práctica de la guía planteada en el capítulo 3. Seguiremos las fases 1, 2 y 3 luego de responder el cuestionario inicial y en cada fase nos detendremos a considerar lo propuesto frente a las características del caso de estudio al igual que lo haría un equipo de desarrollo en un escenario real. En cada fase también decidiremos cómo lo pondremos en práctica y los motivos para hacerlo de tal manera.

4.2.1 Respondiendo el cuestionario inicial

A continuación, dispondremos de las respuestas al cuestionario inicial.

1. Sí, es un proyecto nuevo. El plazo estimado de duración es de 9 meses, teniendo en cuenta que se trata de un cambio de arquitectura de implementación para una funcionalidad ya existente.
2. El equipo en su conformación total es nuevo, sin embargo, los desarrolladores ya han trabajado en un proyecto anterior junto con 1 de los QA. La cantidad de miembros es un total de 7, de los cuales 5 se encuentran en 1 locación y los 2 restantes en otra locación con una diferencia horaria de 2 horas.
3. El motivo principal de utilizar BDD es el de poder comenzar a realizar un cambio a nivel empresarial a partir de este proyecto, podríamos decir que este proyecto pretende ser "*conejillo de indias*". Este cambio busca solucionar o minimizar los problemas de comunicación y mejorar la velocidad y calidad de las entregas.
4. Algunos de los problemas actuales son: la cantidad de defectos encontrados se incrementan a medida que llega la fecha límite de entrega, no hay un tiempo dedicado para resolución de defectos, existe una brecha de comunicación entre los desarrolladores y los QA, las pruebas automatizadas con las que se cuenta no son confiables.
5. Se utilizan algunas de las prácticas planteadas por Scrum: *Daily meetings, sprint planning* y se usa un tablero digital para las tareas y/o historias que se encuentran en desarrollo.
6. Los distintos clientes no participan directamente en contacto con el equipo, sino que la comunicación con ellos es mediante el rol de BA que se encarga de funcionar como intermediario, por el momento no se plantea una posible incorporación de los clientes activamente en el desarrollo.
7. Hay una herramienta de documentación que se utiliza en la empresa y que se encuentra disponible para todos los proyectos. Esta herramienta la utilizan principalmente los desarrolladores para exponer el trabajo que se realiza y también los BA para describir lo que se requiere de cada historia. Todos los miembros del equipo hacen uso de la herramienta de forma directa o indirecta en algún momento.

4.2.2 Recorriendo la fase 1

Recordemos primero qué nos propone la fase 1 de la guía:

1. **Entender** el negocio con Impact mapping y Objetivos SMART.
2. **Planificar** con ejemplos con story planning, reunión de 3 amigos e historias de usuario INVEST.
3. **Mejorar** el feedback con *daily meeting, sprint review, retrospective meeting.*

Si bien en nuestro caso de estudio la funcionalidad ya existe en el sistema, el hecho de entender al negocio y saber cuál es el motivo detrás del proyecto y cuáles son los objetivos que se persiguen, es de gran importancia para un correcto desarrollo.
Debido a que no es posible realizar un ***impact mapping*** en ese punto con el cliente, lo que se hará será llevar a cabo una reunión de todo el equipo con el BA, quién cumplirá el rol de Product Owner, mediante la cual se buscará entender cuáles son las features que debe satisfacer el proyecto.
Se espera que luego de esta reunión, el equipo tenga una idea precisa de cuáles son los motivos de este proyecto (en particular el motivo de la reestructuración) y tener claramente identificadas las features que se llevarán a cabo.

Con respecto al punto 2 se decide que la reunión de story ***planning*** se realizará con todo el equipo y el BA quien también en este caso cumplirá el rol de Product Owner. Se buscará a partir del funcionamiento existente y de las features identificadas en la reunión anterior, poder identificar las historias de usuario que formarán parte del proyecto. Para poder alcanzar una visión más objetiva se solicitará la participación de un miembro de otro equipo que cumpla el rol de facilitador. Además, dichas historias de usuario serán escritas en el formato de historia de usuario planteado por BDD (Ver Figura 9).
Hasta el momento y como parte del proceso *Waterfall* existente, cuando el desarrollador terminaba de escribir el código y el mismo pasaba a estar disponible en el ambiente de pruebas, se realizaba una reunión entre el DEV y el QA. En dicha reunión el DEV explicaba el trabajo que había hecho, mostrándolo con un ejemplo en vivo y además explicaba al QA cuáles pruebas había hecho y de esta manera el QA obtenía la información necesaria para realizar sus pruebas y verificar el correcto comportamiento de la funcionalidad. Irónicamente el nombre de dicha reunión era: "***dev exit***", es decir la puerta de salida del desarrollador, el punto a partir del cual dejaba todo en manos del QA para que este realice las pruebas pertinentes.

Teniendo en cuenta esto, el siguiente cambio o medida a tomar siguiendo la guía práctica, será incorporar la reunión de 3 amigos. En lugar de reemplazar la "*dev exit*" que recién mencionamos, lo que haremos es: antes de comenzar el desarrollo realizar la **reunión de 3 amigos** entre el DEV, el QA y se pedirá la participación del BA. De esta manera se podrá alinear lo que se espera, junto con cómo se probará y como se desarrollará, en otras palabras, obteniendo los criterios de aceptación de manera colaborativa y escribiendo los mismos en formato de escenarios. Pero, además, luego de finalizado el desarrollo se realizará la reunión de *"dev exit"* pero con el nombre de "***checkpoint meeting***". La idea será que en la misma se puedan reunir nuevamente el DEV y el QA para corroborar que lo planteado en la reunión de 3 amigos se cumplió.

Con respecto a las distintas reuniones para obtener feedback planteadas en la guía: la reunión de ***Daily meeting***, es una práctica que ya se encuentra adoptada por el equipo, ya que algunos de sus miembros han trabajado juntos anteriormente. La reunión de ***sprint Review,*** se decide que no se llevará a cabo principalmente porque el cliente no participa activamente del desarrollo, sino que lo hace a través del BA y además al ser una funcionalidad ya existente no se encuentra particularmente útil esta reunión. La reunión ***retrospective meeting*** será una nueva incorporación para el equipo, la misma se decide que se realizará 1 vez por mes y se pedirá la participación de un miembro de otro equipo para que cumpla el rol de facilitador. Si bien parte del equipo ya ha trabajado junto en el pasado, debido a todos los cambios que tendrán lugar con la implementación de la guía práctica, se considera que con esta reunión se podrá hacer un buen seguimiento del bienestar del equipo y poder detectar tempranamente cualquier inconveniente que pueda surgir o desvío en la planificación.

A modo de resumen podemos ver en la Figura 37 por un lado que es lo que se propone en la guía para la fase 1 y por otro lado su correspondencia con las decisiones tomadas para la implementación de la misma. Hagamos entonces un repaso de las correspondencias una a una:

- *Impact Mapping* => en su lugar se realiza una reunión con la figura de Product Owner.
- *Features* => se utilizará el formato planteado por BDD en el capítulo 3. (Ver Figura 14)
- *Story Mapping* => se realizará con un facilitador externo.
- Historias INVEST=> se utilizará el formato de historias planteado por BDD en el capítulo 2 . (Ver Figura 9)
- Reunión de 3 amigos =>se realizará solicitando la participación del BA.
- Criterios de aceptación. => se utiliza el formato Gherkin para describir los criterios en forma de escenarios como explicamos en el capítulo 2. (Ver Figura 8)
- *Checkpoint meeting*. => se mantendrá la reunión de "*dev exit*" con este nuevo nombre como punto de corroboración para lo pautado en la reunión de 3 amigos.

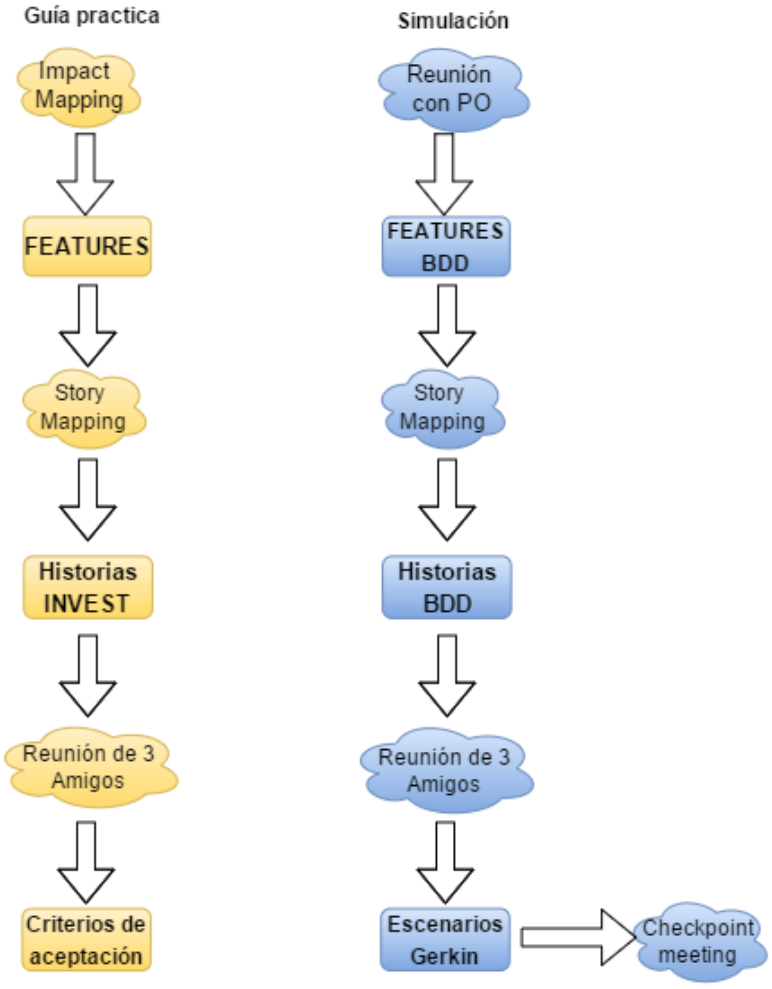


Figura 37. Recorriendo la Fase 1

4.2.3 Recorriendo la fase 2

Recordemos lo que nos propone la guía en la fase 2:

- **Automatizar** las especificaciones de **alto nivel**, partiendo de los escenarios, definiendo los step definitions y creando la implementación de dichos steps.
- **Automatizar** las especificaciones de **bajo nivel** con la utilización de pruebas unitarias.
- **Automatizar** el proceso de **construcción, entrega y despliegue** del producto de software con herramientas de integración continua, entrega continua y despliegue continuo.

Como mencionamos anteriormente, en ATLab se cuenta con una herramienta de automatización interna, que ha sido desarrollada en los últimos años. Se realizó una reunión con el equipo encargado de su mantenimiento para poder tomar una decisión respecto a cómo automatizar especificaciones de alto nivel en el marco del proyecto, que es nuestro caso de estudio. Actualmente esta herramienta que pasaremos a llamar "**LiveSel**" se encuentra desarrollada en lenguaje Java con **TestNG**[63] y **Selenium WebDriver**, posee integración con **Jenkins**[64] y **Maven**[65] utilizando como repositorio **GitHub**[66]. Aunque la herramienta de automatización se encuentra funcionando de manera apropiada, es constante la cantidad de pruebas que fallan por distintos motivos entre ellos, la falta de actualización de las pruebas. Lo que se hará será seguir utilizando la herramienta "**LiveSel**" y se incorporará **Cucumber JVM**[67] para la definición de los escenarios.

Con respecto a las automatizaciones de **especificaciones de bajo nivel** se decide que se continuarán realizando de la misma forma que se hacen hasta el momento: creando pruebas unitarias que forman parte del pipeline de Jenkins. En este punto tenemos que tener en cuenta que ya existen algunas pruebas para la funcionalidad y que es probable que se necesario actualizarlas y además crear nuevas. Dichas pruebas unitarias se encuentran creadas en **JUnit**[68].

Por último, en cuanto a automatizar el **proceso de construcción, entrega y despliegue,** ya se cuenta con un proceso bien definido que utiliza Jenkins para la construcción y entrega, con lo cual sobre este punto no se realizará modificación alguna.

---

[63] Es un framework Java inspirado es JUnit que permite la creación de pruebas unitarias, funcionales, entre otras. http://testing.org/doc/index.html

[64] Es un servidor de integración continua, gratuito, open-source. https://jenkins.io/

[65] Es una herramienta de software para la gestión y construcción de proyectos Java. https://maven.apache.org/

[66] Es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. https://github.com/

[67] Es la implementación para Java de Cucumber. https://cucumber.io/docs/reference/jvm

[68] Es un framework de Java que nos permite escribir pruebas repetibles. http://junit.org/junit4/

A modo de resumen podemos ver en la Figura 38 la correspondencia entre lo que plantea la guía en la fase 2 y las decisiones tomadas en esta sección para su implementación. Repasemos cada una de ellas:

1. Escenarios => se escribirán en la herramienta Cucumber JVM.
2. *Step definitions* => se crearán en la herramienta LiveSel.
3. *Step definition* Implementación= > se realizarán con Selenium WebDriver.
4. Especificaciones de bajo nivel => se actualizarán las pruebas unitarias implementadas en JUnit.

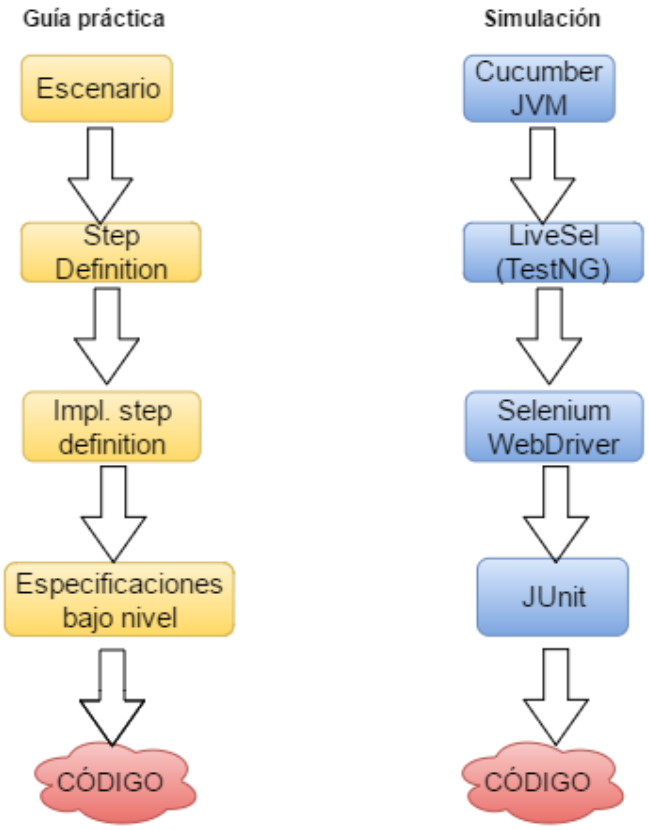


Figura 38.Recorriendo la Fase 2

4.2.4 Recorriendo la fase 3

Recordemos lo que nos propone la guía en la fase 3:

- **Crear y mantener** una documentación viva.
- **Implementar métricas**: running tested features, burn chart, cumulative flow, velocity, Niko Niko Calendar.
- **Utilizar** una herramienta de Backlog Digital.

Para poder crear y mantener una buena documentación viva se decide llevar a cabo una integración entre **Jenkins** y **VersionOne**[69]. Ambas herramientas ya se están utilizando desde algún tiempo, pero de manera separada, se buscará con la integración que la herramienta VersionOne pueda nutrirse de información actualizada de ejecución de pruebas proveniente de Jenkins. Esta tarea de integración la realizará el equipo que se encarga de mantener la configuración de Jenkins en conjunto con los desarrolladores del proyecto.
En lo que respecta a las métricas propuestas se utilizarán 3 de ellas: **burn chart, cumulative flow y velocity**, todas ellas disponibles en VersionOne[70]. El encargado de realizar un seguimiento de las mismas será el BA, en tanto que todos los miembros del equipo podrán ingresar a la sección de métricas para consultarlas.

Por último, en lo que respecta al **Backlog Digital**, como venimos mencionando anteriormente se utiliza la herramienta VersionOne, que también posee una sección de tablero virtual, el cual se encontrará disponible para la comunicación entre los miembros del equipo al momento de las reuniones diarias. Asimismo, VersionOne se integrará con Jenkins para obtener información de los resultados de ejecución de las especificaciones automatizadas. Como podemos ver esta herramienta pasará a contener nuestra "*Documentación viva*".

A modo de resumen vemos en la Figura 39 la correspondencia entre lo propuesto por la guía en la fase 3 y lo que se decide implementar. Repasemos entonces cada punto:

1. Integración y entrega continua => usaremos Jenkins que actualmente ya está configurado y en funcionamiento. Integraremos esta herramienta con VersionOne [71].
2. *Backlog Digital* => mantendremos el uso de la herramienta VersionOne como backlog digital.
3. *Features* => utilizaremos Cucumber JVM para la definición de los escenarios y AutoSel para los steps definitions tal como se describe en la sección 4.2.3.

Como podemos observar en el gráfico de la Figura 39, nuestra documentación viva quedará centralizada en VersionOne.

[69]Es una herramienta de gestión de proyectos ágiles. https://www.versionone.com/
[70]Podemos ver cuáles son las métricas que brinda VersionOne en:
https://www.versionone.com/product/lifecycle/agile-reporting/
[71]Podemos ver cuáles son las integraciones con otras herramientas que brinda VersionOne en:
https://www.versionone.com/product/connect/integrations/platform/

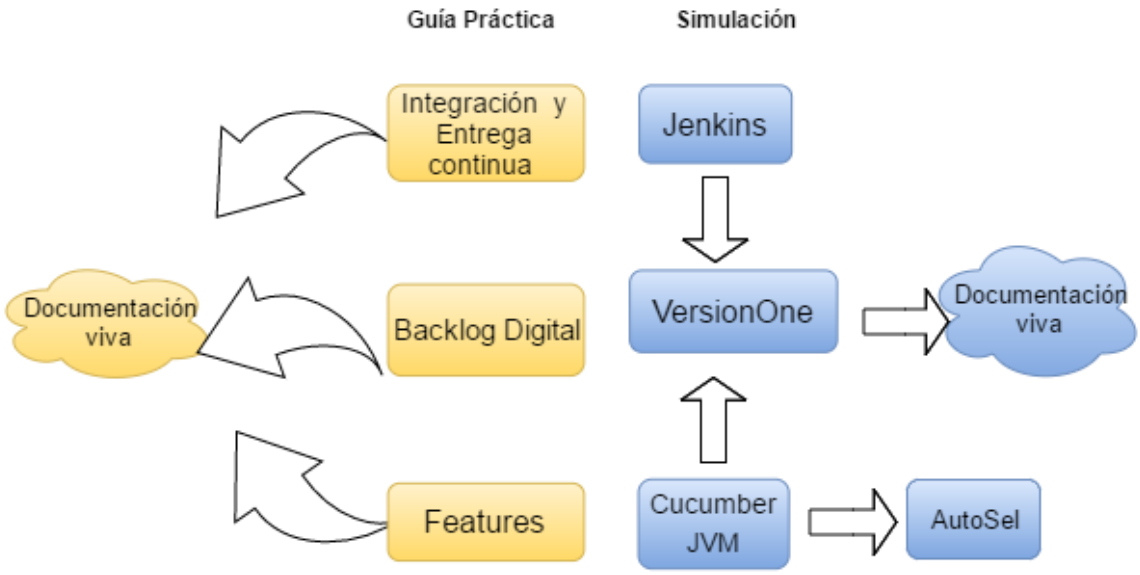


Figura 39. Recorriendo la Fase 3

4.3 Resultados obtenidos

"If you want different result, do not do the same thing", Albert Einstein.

Recapitulemos que he hemos hecho en el desarrollo de este capítulo:

En primer lugar, hicimos una descripción de las características particulares que posee la empresa donde se desarrolla nuestro proyecto; dimos una definición de desarrollo semi-ágil y explicamos por qué nuestro caso de estudio cumple con dicha definición, además hablamos de los problemas actuales que se presentan y algunos de los cuales queremos dar solución con la aplicación de la guía propuesta en el capítulo 3.

Luego como parte de la simulación, respondimos el cuestionario y recorrimos las fases 1,2 y 3. En cada una de ellas dado lo aconsejado en la guía decidimos cómo se implementaría y explicamos los motivos. Además de cada fase describimos un gráfico comparativo con lo recomendado y como se llevaría a cabo.

Ahora entonces presentaremos algunos de los resultados obtenidos simulando la aplicación de la guía y una vez finalizado el proyecto, para ello haremos uso de tablas y gráficos cuyos contenidos iremos explicando de a uno.

4.3.1 Los releases

Antes de entrar en los detalles de la información obtenida como resultado, cabe mencionar que el proyecto tuvo una duración de 9 meses. Siguiendo con la metodología de Scrum (recordemos aquí que uno de los problemas planteado era: Uso incorrecto de SCRUM) y alineados con las expectativas empresariales (recordemos los "*Release Milestones*") se llevaron a cabo 9 Releases, es decir 1 Release por mes, compuesto por 2 sprints, donde cada sprint tuvo una duración de 2 semanas.

El conjunto de Features surgió a partir de la reunión con el rol de PO (Ver Figura 37), mientras que las historias de usuario asignadas a cada release fueron resultado de la reunión de *story mapping* llevada a cabo también con el rol de PO.

Teniendo todo esto en cuenta, veamos los primeros resultados:
Cantidad de **Features** =10
Cantidad de **Historias de Usuario** =110

Feature	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Historias de Usuario	8	15	11	16	8	12	14	10	6	10

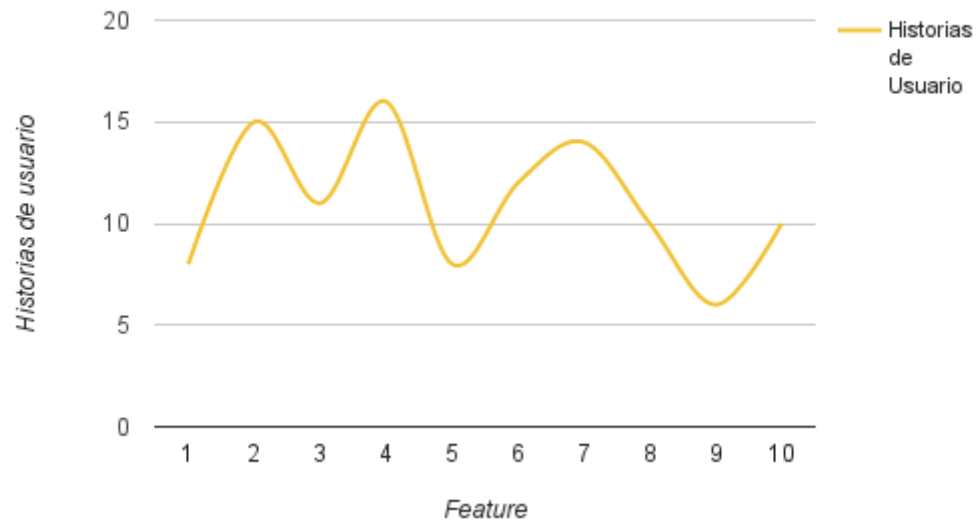


Figura 40. Features y Historias de Usuario.

Si bien la cantidad de historias no está directamente relacionada con los releases, lo que nos indica es el costo de cada feature en término de complejidad, siendo que muchas veces features muy grandes suelen descomponerse en historias de usuario más pequeñas de forma tal que se puedan manejar dentro del alcance de un sprint.

Lo interesante en este punto de las historias de usuario con respecto a la guía es poder ver qué cantidad de reuniones de 3 amigos se llevaron a cabo, ya que es una de las reuniones de vital importancia para BDD, debido a que de ella surgen los criterios de aceptación que se convertirán en especificaciones ejecutables de alto nivel.

En la Figura 41 veremos entonces relacionado con las historias de usuario, la cantidad de reuniones de 3 amigos tuvieron lugar (recordemos que idealmente se debería llevar a cabo una reunión de 3 amigos por cada historia de usuario):

Feature	1	2	3	4	5	6	7	8	9	10
Historias de Usuario	8	15	11	16	8	12	14	10	6	10
Reunión de 3 amigos	8	12	8	10	6	12	12	6	6	7

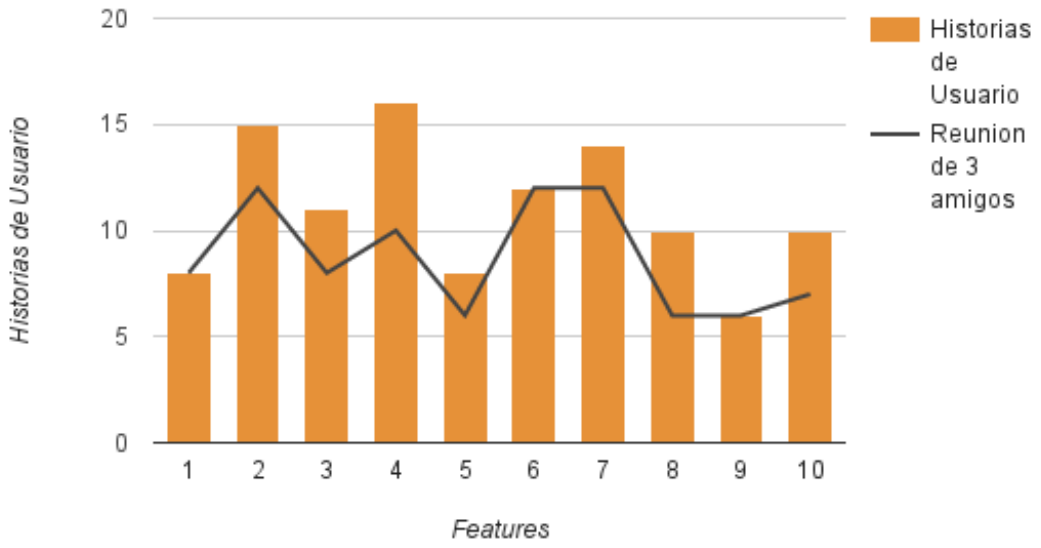


Figura 41. Historias de Usuario y Reunión de 3 amigos.

Como podemos observar en la versión gráfica de la Figura 41, la cantidad de reuniones de 3 amigos que se llevaron a cabo se encuentra muy cercana a la cantidad de historias de usuario, en base a lo cual podemos interpretar:

- Por un lado, la práctica resultó ser efectiva para la obtención de criterios de aceptación, esta afirmación es válida teniendo en cuenta que de no haber resultado beneficiosa se denotaría una fuerte caída de la práctica luego del comienzo del proyecto. Sin embargo, vemos que se mantuvo cercana a lo ideal.
- Por otro lado, también observamos que en los casos en donde la cantidad de reuniones de 3 amigos se redujo fue en aquellas features con más de 6 historias de usuario. En este punto sabemos que es muy posible que las historias de usuario hayan estado íntimamente relacionadas entre sí, con lo cual una misma reunión de 3 amigos puede que haya resultado suficiente para luego a partir de la misma obtener los criterios de las historias relacionadas.

Recordemos que además de la reunión de 3 amigos también se había establecido realizar una reunión luego de finalizado el desarrollo de la historia con el fin de poder validar lo planteado en la reunión de 3 amigos. A esa reunión le pusimos el nombre de *checkpoint meeting*. Veamos la cantidad de estas reuniones que se llevaron a cabo:

Feature	1	2	3	4	5	6	7	8	9	10
Historias de Usuario	8	15	11	16	8	12	14	10	6	10
Checkpoint meeting	6	4	4	2	0	0	0	0	0	0

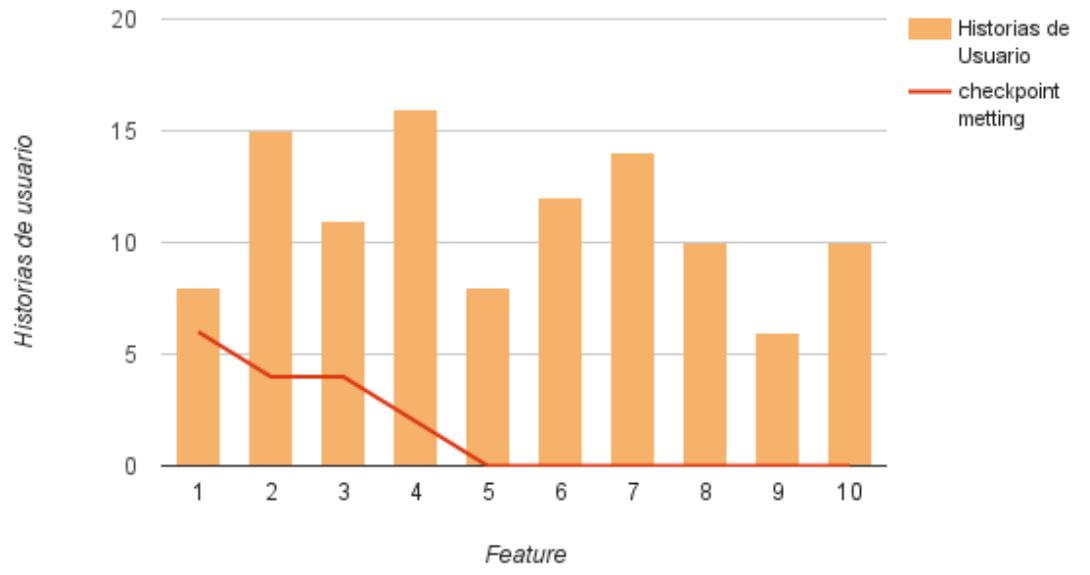


Figura 42. Historias de Usuario y checkpoint meeting.

Fácilmente podemos ver tanto en la tabla como en el gráfico de la figura 42 lo que sucedió con respecto a las reuniones de checkpoint meeting: se desestimaron rápidamente. En las primeras historias de usuario trabajadas se llevó a cabo esta reunión, solo a fin de cumplir con lo pautado, sin embargo, pronto se evidenció que las mismas carecían de sentido. Analizaremos este resultado con más profundidad en el capítulo 5.

Por otro lado, para poder obtener las historias de usuario se llevaron a cabo 2 sesiones de story mapping. Dado que las features ya se encontraban identificadas como resultado de la reunión con la figura de PO y el conjunto era demasiado grande para analizarlo en detalle en una sola sesión, en la primera sesión se trabajó sobre las primeras 5 features, dando como resultado los primeros 4 releases. En la segunda sesión se trabajó sobre las features restantes dando como resultado los 5 releases finales.

Release	R1	R2	R3	R4	R5	R6	R7	R8	R9
Historias de usuario	8	10	11	12	13	24	12	12	8

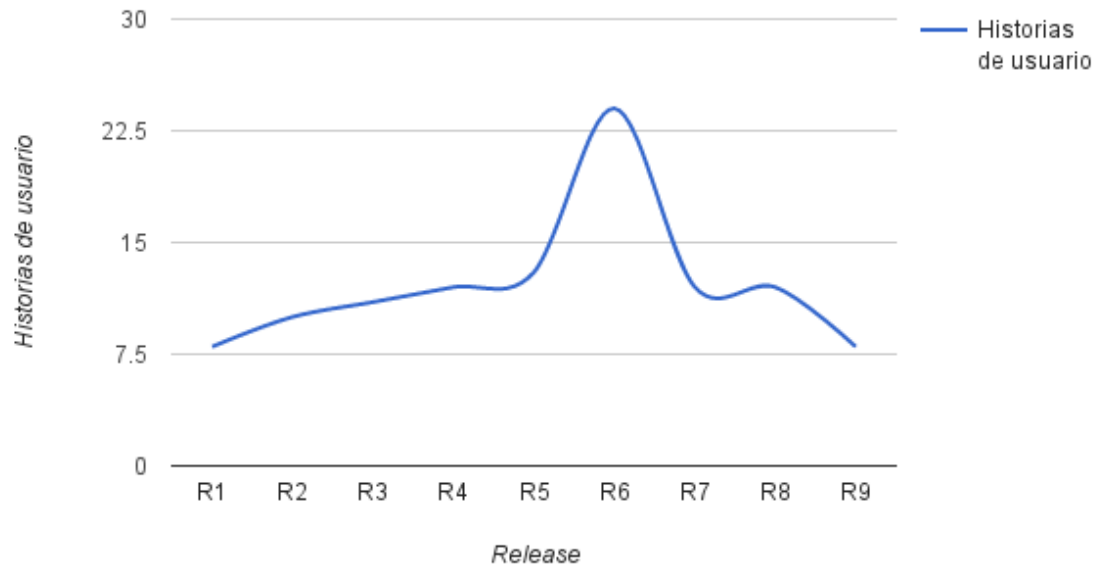


Figura 43. Releases y Historias de usuario.

Como podemos ver en la Figura 43 el promedio de cantidad de historias de usuario por release se dio entre 8-15, siendo de manera excepcional R6 el cual tuvo 24 historias de usuario. Este incremento notorio en la cantidad de historias de usuario se debe principalmente al hecho una feature demasiado grande que fue descompuesta en varias historias de usuario específicas.

Relacionado también con los releases, debemos también mencionar un conjunto de tareas adicionales al desarrollo de las historias que debieron llevarse a cabo sobre todo en los primeros releases. Algunas de ellas fueron:

- Creación de nuevo proyecto en Git.
- Reunión de diseño con el equipo de automatización.
- Incorporación de Cucumber JVM a la LiveSel.
- Reunión de diseño del equipo de desarrollo.
- Integración de VersionOne con Jenkins.
- Taller de Cucumber JVM
- Relevamiento de pruebas de regresión existentes.
- Relevamiento de defectos previos a la nueva implementación.
- Configuración de nuevos pipelines en Jenkins.
- Creación de ambiente "*Sprint test*".
- Migración de datos viejos a la nueva arquitectura.
- Ejecución de pruebas en producción.
- Lanzamiento a producción.
- Armado de scripts para la puesta en producción.

4.3.2 Los defectos

Uno de los principales problemas que habíamos destacado era respecto de los defectos, en particular mencionamos como estos solían encontrarse casi sobre el final del tiempo dedicado a las pruebas. También describimos el caso particular de los DEV que debían seguir implementando historias de usuario nuevas y al mismo tiempo solucionar defectos hallados sobre lo anteriormente desarrollado, haciendo que sea más propenso a nuevos errores (lo que describimos en la sección de problemas actuales como: Falta de tiempo exclusivo de *"bug fixing"*)

Veamos que sucedió en nuestro caso de estudio con respecto a los defectos:

Release	R1	R2	R3	R4	R5	R6	R7	R8	R9
Defectos encontrados	2	5	2	9	2	4	9	4	3
Defectos solucionados	2	3	4	6	3	3	7	6	6
Defectos bloqueantes	1	2	0	2	2	0	0	1	0

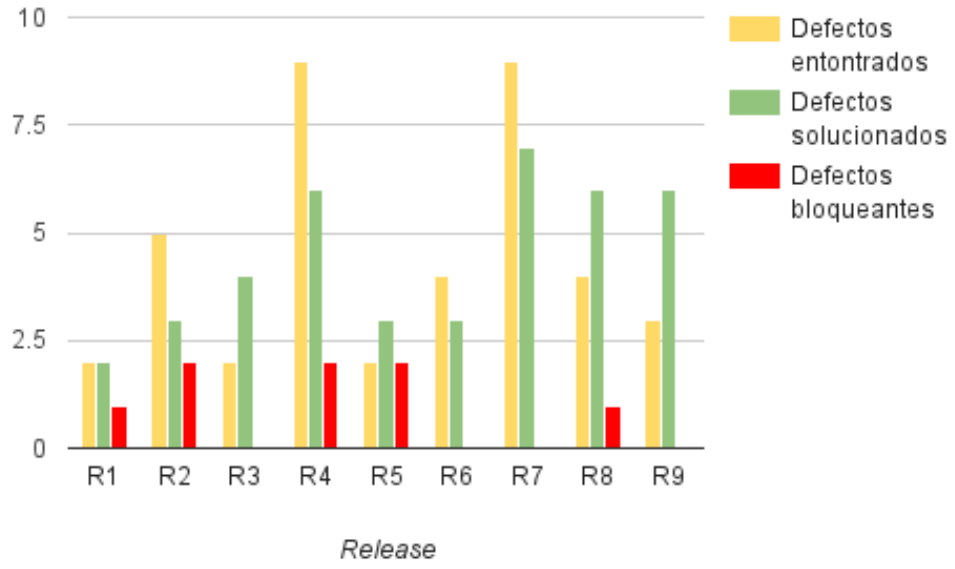


Figura 44. Releases y Defectos.

Si bien en la Figura 44 no estamos discriminando el momento en el cual fueron hallados los defectos con respecto a la finalización del tiempo de pruebas, lo que sí podemos observar tanto en la tabla como en el gráfico son varias cosas:

- Por un lado, el promedio de defectos encontrados en cada release se mantuvo en 4, lo cual es un número consideradamente bajo teniendo en cuenta el volumen de historias de usuario.
- Por otro lado, la cantidad de defectos bloqueantes tiene un total de 8, lo que nos indica de alguna manera que la calidad del código producido era bastante bueno.

4.3.3 El equipo

Antes de aplicar la guía habíamos señalado 3 problemas que se relacionan directamente con el equipo:

- Ausencia de trabajo colaborativo/paralelo entre los distintos roles.
- Mala comunicación entre los roles.
- Falta de feedback con respecto al bienestar del equipo.

Cuando recorrimos la fase 1 de la guía, una de las decisiones que se tomaron fue llevar a cabo la reunión de 3 amigos para las historias de usuario. La introducción de esta reunión provocó un cambio que podemos decir fue significativo y alentó a un trabajo colaborativo. En la figura 45 podemos ver este cambio de manera gráfica, antes y después de implementar la guía. Antes dada una historia de usuario, el BA trabajaba en ella identificando los criterios de aceptación que luego el DEV tomaba para crear el código, el cual era finalmente aprobado por el QA. En cambio, después de implementar la guía, los 3 roles se reúnen para obtener los criterios y dichos criterios son utilizados por el DEV, el QA y el BA para trabajar colaborativamente.

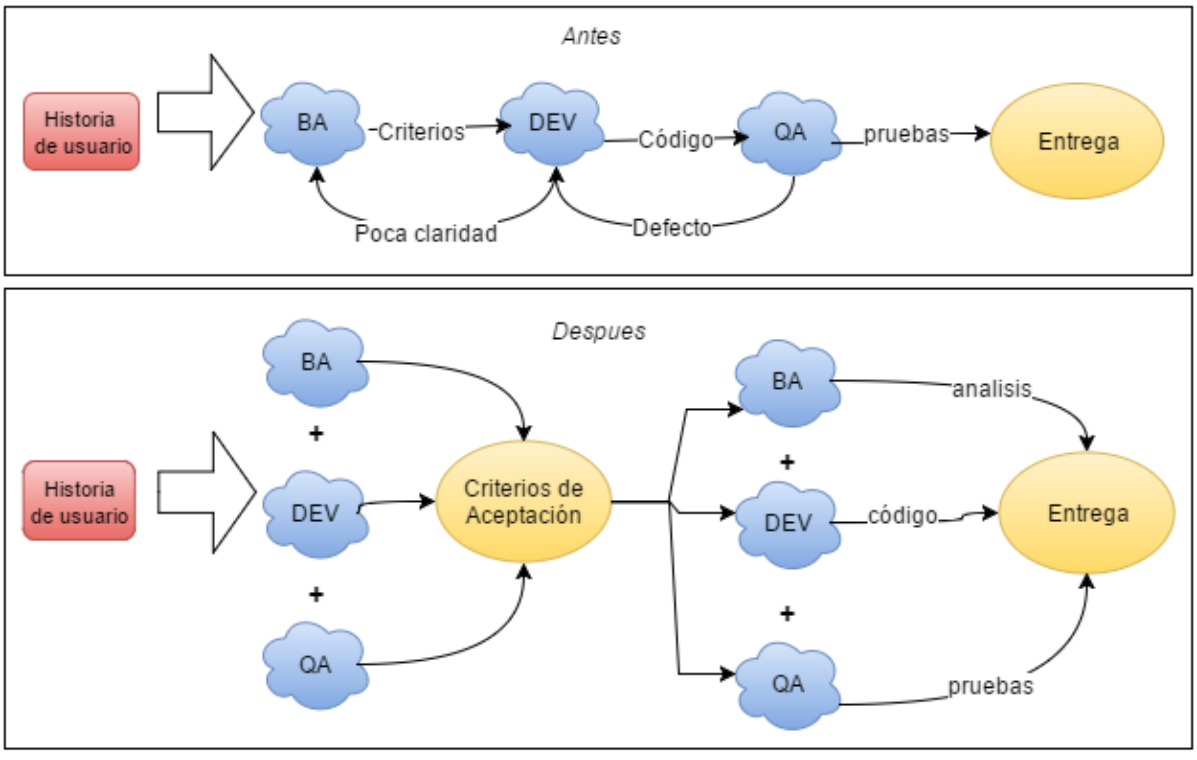


Figura 45. Trabajo en Equipo. Antes y Despues

Otra práctica que también se introdujo a partir de la guía fue la retrospectiva ("*Retrospective meeting"*). Esta reunión se llevó a cabo con una periodicidad de 1 mes, correspondiendo así con cada uno de los releases y también con un bloque de historias de usuario comprometidas.
Para 3 de dichas reuniones se con contó con la ayuda de un facilitador, quien fue una persona perteneciente a otro proyecto dentro de ATLab, sin embargo, para el resto de las repeticiones no se pudo contar con un facilitador externo siendo entonces uno de los QA quien dirigió la reunión. Dichas reuniones tuvieron una duración aproximada de 2 horas, siendo el día y la hora seleccionada por todos los miembros del equipo. Como además el equipo se encontraba distribuido se utilizó la herramienta Webex[72] para la comunicación y VersionOne[73] para registrar los aciertos, problemas y acciones que surgieron en cada una de ellas.

Si bien al principio los miembros del equipo no se encontraban familiarizados con la práctica, dichas reuniones sirvieron para identificar tanto nuevas tareas que no se habían considerado como pequeños inconvenientes que surgieron y las acciones para solucionarlos en los siguientes releases.

El esquema utilizado para estas reuniones puede verse en la Figura 46.

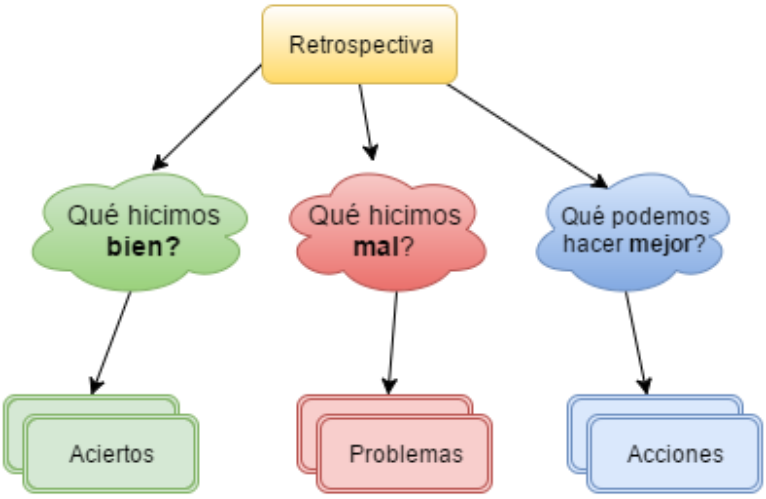


Figura 46 . Retrospectiva Esquema utilizado

[72] https://www.webex.com/
[73] https://community.versionone.com/VersionOneLifecycle/The_Agile_Process/Sprint_or_Iteration_Review/Retrospectives

4.3.4 Las pruebas

Con respecto a las pruebas sucedió algo interesante, ya que si recordamos el problema planteado alrededor de las pruebas era: Las pruebas automatizadas no son confiables. Con la puesta en práctica de la guía comenzamos a hablar de especificaciones ejecutables de alto nivel que fueron creadas en Cucumber JVM e incluidas en un pipeline de Jenkins. Las pruebas que existían al momento de comenzar el proyecto fueron discontinuadas, debido a que se encontraban fuertemente ligadas a la implementación anterior. Podemos ver lo anteriormente mencionado de forma gráfica en la Figura 47.

Sobre las pruebas también nos interesa mencionar que la curva de aprendizaje para utilizar Cucumber JVM e integrar con "*LiveSel*" se vio un poco complicada en sus comienzos, lo cual hizo que parte de los desarrolladores dedicaran tiempo a ayudar a los QA en esta tarea. Esto hizo que los DEV tuvieran una mayor visión de cómo se creaban las especificaciones de alto nivel, lo cual les brindó otra perspectiva respecto del desarrollo y además propició la comunicación tanto con los QA como con los BA.

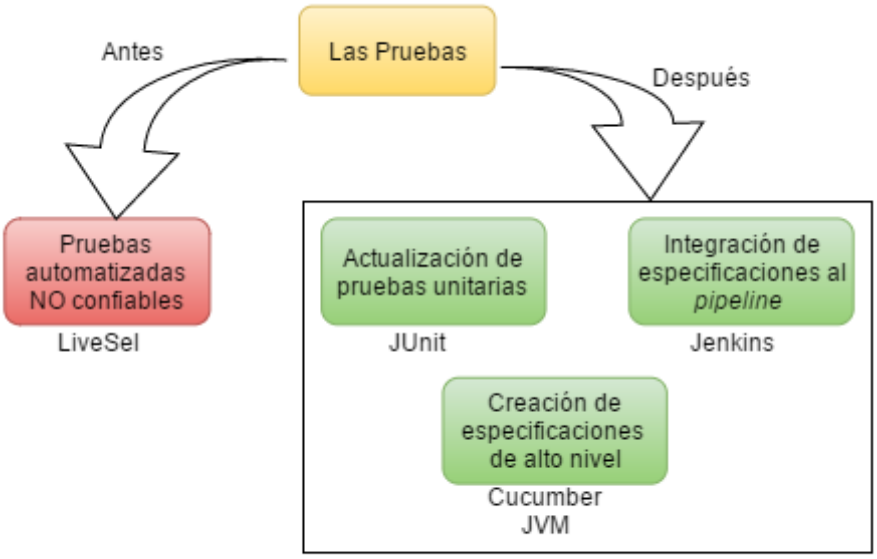


Figura 47. Las pruebas Antes y Después.

Otro punto que es interesante mencionar es que, si bien se crearon especificaciones ejecutables usando Cucumber JVM, también se crearon pruebas manuales y exploratorias, ya que recordemos que no todo es conveniente automatizarlo como bien explicamos en el capítulo 3. Además, se tuvieron que incluir pruebas de regresión de manera tal de corroborar que la nueva arquitectura mantiene y mejora la implementación de la arquitectura anterior, con la posibilidad también de deshabilitar la nueva arquitectura en caso de ser necesario.

Por último, un dato no menos interesante para resaltar, es respecto a los ambientes, recordemos que se poseen 4 ambientes: Desarrollo, Testing, Integración, Producción. Ahora bien, como tanto DEV y QA se encontraban trabajando de forma paralela, la primera opción fue: que los desarrolladores utilicen el ambiente de desarrollo y luego los QA el ambiente de testing, pero esto implicaba tener que pasar por la burocracia establecida a nivel empresarial para el pasaje de un ambiente a otro, el cual lleva tiempo y retrasaría al equipo. Como la idea era que el trabajo fuese más dinámico, se optó por crear un nuevo ambiente similar a desarrollo pero que fuese administrado por los QA y que no tuviese que atravesar la burocracia empresarial, sino que permitiera una mayor agilidad entre los cambios introducidos por los desarrolladores y las especificaciones creadas por los QA. Una vez que en dicho ambiente llamado "*Sprint test*" (haciendo referencia a lo de desarrollado en un sprint y por eso el nombre) las pruebas fueran exitosas es que se volcaban los cambios al ambiente de Testing.

4.3.5 Integración de herramientas

Como vemos en la Figura 48, antes VersionOne se encontraba aislada de los cambios que se producían entre Github (la herramienta encargada de versionamiento y almacenamiento de código) y Jenkins (la herramienta encargada de la integración continua). Después de la integración, cada vez que se genera un nuevo build se realiza una actualización desde Jenkins hacia VersionOne, haciendo posible que tanto las historias de usuario como defectos se encuentren relacionadas con un build y a su vez permite saber para un build dado que historias de usuario y defectos se incluyeron.

Esta integración[74] permitió entre otras cosas:

- Realizar un **seguimiento de las historias de usuario** en todas sus etapas: definición, armado de criterios de aceptación, ejecución de especificaciones de alto nivel, implementación del código.
- Realizar un **seguimiento de los defectos**, pudiendo relacionarlos no solo con la historia a la cual pertenecían, sino también con su especificación ejecutable (en caso de ser un escenario automatizado con Cucumber JVM), relacionar en qué versión de código se produjo el defecto y en cual versión se solucionó.
- Crear una **documentación "*viva*"** ya que la información provista por VersionOne se encuentra nutrida de las ejecuciones provenientes del pipeline de Jenkins. Si bien es cierto que solo la incorporación de modificaciones en el código dispara la generación de nuevo build y ello a su vez ejecuta las especificaciones de alto nivel, es posible también realizar la ejecución solo de estas últimas, las cuales en su gran mayoría forman parte de la suite de regresión diaria mantenida por el sector de automatización de la empresa.

[74]Para más información acerca de cómo se realiza esta integración: https://community.versionone.com/VersionOne_Connect/Supported_Integrations/VersionOne_Integration_for_Jenkins

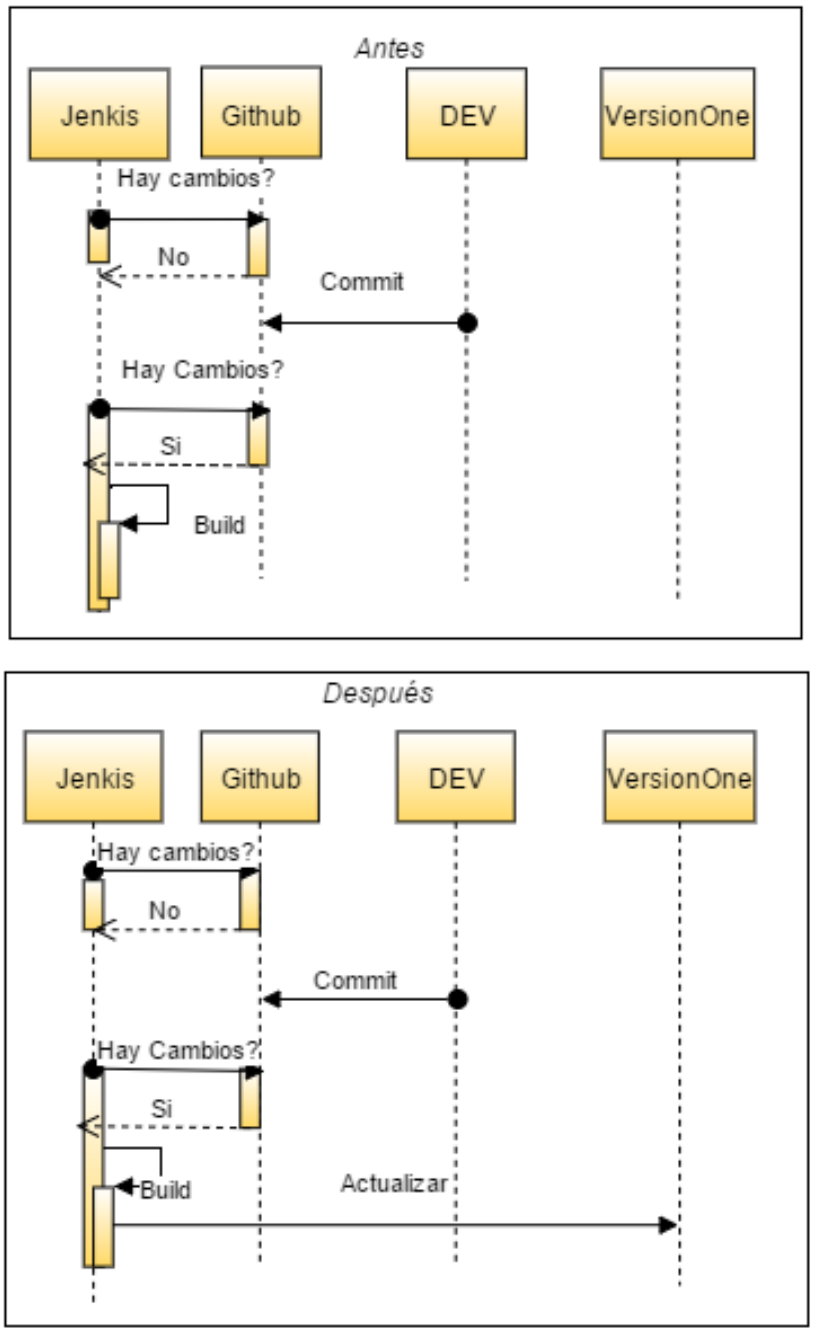


Figura 48. Jenkins y VersionOne. Antes y después
(Basado en Jenkins Integration sequence)

4.3.6 Las métricas

Muy relacionado con la integración de herramientas que mencionamos en la sección anterior es que se encuentran las métricas. Como bien podemos ver en la Figura 49 antes de implementar la guía, uno de los problemas era que: Documentación escasa y no actualizada. Sin embargo, ahora, gracias al trabajo de todos los miembros del equipo en mantener actualizadas las tareas en el tablero y en parte a la integración realizada entre VersionOne y Jenkins, la documentación se encuentra actualizada y cargada correctamente en VersionOne, lo que nos permite obtener métricas. En particular las métricas que nos interesan (ya que VersionOne provee algunas métricas más) son: velocity, burn chart y cumulative flow(CFD) todas ellas fácilmente generadas desde la herramienta y disponibles en todo momento.

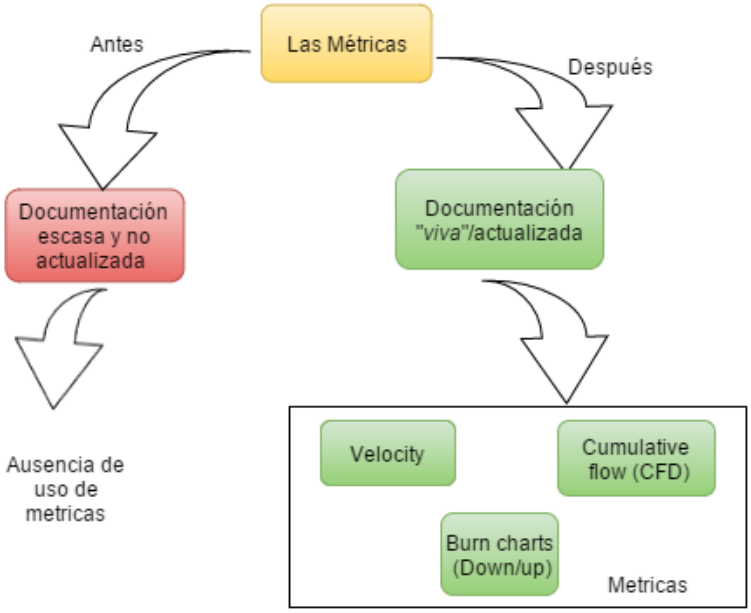


Figura 49. Las métricas Antes y Después.

4.4 Resumen del capítulo

En este capítulo describimos nuestro caso de estudio, explicando en primer lugar porque el mismo se encuadra en un desarrollo semi-ágil, para luego enumerar los problemas con los cuales se lidiaba. Acto seguido y como primera parte de la simulación, respondimos el cuestionario y recorrimos cada una de las fases planteadas por la guía del capítulo 3, decidiendo en cada una de ellas como se haría la implementación de la guía. Por último, clasificamos y describimos brevemente los resultados obtenidos una vez finalizado el proyecto los cuales serán analizados en el próximo capítulo.

Capítulo 5. Análisis resultados

Luego de la simulación sobre el caso de estudio planteado en el capítulo anterior, analizaremos en primer lugar las ventajas y desventajas halladas, luego los aportes a los distintos roles y finalmente enunciaremos una serie de mejoras a la guía.

Este análisis realizado es válido para el caso simulado propuesto, pero brinda una pauta para enunciar ventajas, desventajas, aportes y mejoras a la guía.

5.1 Ventajas

Baja tasa de defectos

Como bien pudimos ver en la sección “Los defectos” del capítulo 4, la cantidad de defectos en ninguno de los releases superó el número 10 y la cantidad de defectos bloqueantes totales fue 8, de lo cual podemos inferir que la tasa de defectos fue baja durante todo el desarrollo.
El hecho de que la tasa de defectos sea reducida nos podría estar indicando por un lado que la calidad de lo desarrollado es alta y, además, que el uso de TDD para las especificaciones de bajo nivel y BDD para las de alto nivel de manera conjunta hacen que cumplir con los criterios de aceptación y evitar los defectos sea más sencillo.

Pruebas automatizadas más confiables

Antes de aplicar la guía se contaba con pruebas automatizadas que no eran confiables, en parte porque no estaban actualizadas, a partir de lo planteado por la guía no solo se crearon nuevas pruebas, sino que se crearon especificaciones de alto nivel en Cucumber JVM que permiten tener una nueva capa de abstracción con respecto a su implementación en “*LiveSel*”. Con este cambio el problema de actualización se erradicó por el momento ya que aún no será necesaria una actualización hasta que existan nuevos cambios que así lo requieran.

Mayor comunicación

Quizás ésta es una de las mayores ventajas tanto buscadas como alcanzadas, ya que recordemos que BDD al igual que el resto de las metodologías ágiles se trata de personas por sobre todo y la comunicación entre ellas es una pieza fundamental. Tanto las reuniones de 3 amigos como las retrospectivas en las cuales los miembros del equipo tuvieron oportunidad de expresarse propiciaron una comunicación más fluida.

Métricas para el seguimiento del proyecto

Gracias al buen uso de VersionOne se pudieron generar métricas tales como Cumulative Flow, Burn up Chart, Burndown Chart Velocity, Defect Trending, Test Status. Todas estas métricas permiten un seguimiento más de cerca y con datos fidedignos acerca del proyecto, además de dejar un precedente. Sobre este punto, lo que principalmente ayudó fue el trabajo de todos los miembros del equipo que contribuyeron en la construcción de la documentación en VersionOne. Tanto la creación de tareas, como la actualización diaria de las mismas, la asignación de personas y de esfuerzo a cada una de las tareas, hizo que fuera posible tener la información necesaria para las métricas.

Trabajo colaborativo

Otro de los grandes inconvenientes de seguir *Waterfall* antes de implementar la guía, era el desfasaje de tiempos que había entre el momento en el cual el desarrollador se dedicaba al código de una historia de usuario y el QA tenía que validarla. Ese desfasaje ya no es posible con lo propuesto por la guía ya que tanto DEV y QA se encuentran en el mismo momento de tiempo trabajado de forma colaborativa sobre la misma historia de usuario y además también se encuentra presente el BA quien participa de las reuniones de 3 amigos para la definición de los criterios de aceptación.

5.2 Desventajas

Tiempo de reuniones

Más allá de haber mencionado las reuniones de 3 amigos y las retrospectivas como una ventaja que facilitó la comunicación, también es por otro lado una desventaja la cantidad de tiempo consumido por dichas reuniones. Recordemos que idealmente se debería realizar una reunión de 3 amigos por cada una de las historias de usuario, teniendo en cuenta que el promedio de duración de dichas reuniones es de 1 o 2 horas (en el mejor de los casos) el tiempo requerido para dichas reuniones se puede ver como una desventaja sobre todo para el rol de BA ya que era una única persona y debía o se esperaba que participe de todas las reuniones de 3 amigos.

Dependencias generadas

A pesar que la integración de herramientas fue un gran logro y marcó una diferencia significativa con respecto a otros proyectos, debemos entender también que involucra una nueva dependencia a VersionOne ya que se mantiene actualizada con información proveniente de Jenkins.

5.3 Aporte a los roles

Si bien existe más roles dentro de un equipo de desarrollo ágil, en nuestro caso de estudio nos focalizamos en 3: QA, DEV, BA. Mencionaremos los aportes para cada uno de ellos a continuación:

Aportes para los QA

Entre los aportes para el rol de QA podemos mencionar: se tiene más tiempo para realizar **pruebas exploratorias**, existe una mejor relación con los desarrolladores, esto es fundamentalmente porque se trabaja de forma paralela. Por otro lado, luego de la puesta en práctica de la guía, los QA ya no son el último eslabón de la cadena para ponerse en conocimiento con una historia de usuario. Además, al participar de las reuniones de 3 amigos tienen un conocimiento más profundo y comprensivo del dominio en el cual se desenvuelve el proyecto.

Aportes para los DEV

Entre los aportes para el rol de DEV podemos mencionar: por un lado, la relación de comunicación con todo el equipo es mejor ya que el contacto con el BA y el QA es más fluida, debido en parte a las reuniones incorporadas y también al trabajo colaborativo. Por otro lado, otro aporte es que se trabaja en la calidad desde el comienzo, esto se debe principalmente al uso de TDD para las especificaciones de bajo nivel y también a la forma de trabajo pautada que les permite crear **código legible** y más sencillo para actualizar.

Aporte para el BA

Entre los aportes para el rol de BA podemos mencionar: se tiene una mejor relación de comunicación con los desarrolladores, ya que, a partir de la especificación de criterios de aceptación en conjunto, **todos se encuentran en la misma página** tanto BA como DEV y utilizan un mismo lenguaje para comunicarse: los ejemplos en los que se basan dichos criterios. Por otro lado, se ahorra tiempo de descripción de casos de uso u otro tipo de especificación que se utilizara antes, ya que la documentación de las historias de usuario se crea a partir del trabajo en común de todos los roles.

5.4 Mejoras a la guía

"If you focus on result, you will never change. If you focus on change, you will get results.", Jack Dixon.

Luego de la simulación de puesta en práctica de la guía sobre nuestro caso de estudio y del breve análisis de sus resultados, se evidenciaron ciertas mejoras que podrían hacerse sobre la guía, las cuales se enumeran a continuación:

1. Crear una "*Encuesta de satisfacción*".
2. Incorporar "*Code Review*"[75].
3. Incorporar "*Taller de Scrum*".
4. Agregar más métricas.

En lo que respecta al punto 1, se cree que sería deseable poder evaluar de manera mínima si la guía tuvo **resultados positivos** o no, del mismo modo que el cuestionario previo cumple la función de filtro para aquellos desarrollos en los cuales es conveniente el uso de la guía; una encuesta luego de la implementación de la misma que compare las respuestas obtenidas del cuestionario y pueda indicar o resaltar los beneficios obtenidos sería sin lugar a dudas una notable mejora.

Por otro lado, el punto 2 podría incorporarse a la sección en donde la guía recomienda el uso de reuniones para fomentar el feedback. Si bien es una práctica referida al código, ya que se encarga de revisar la calidad del mismo teniendo en cuenta varios parámetros (escalabilidad, mantenibilidad, etc.) uno de los focos principales es también el componente social (43) que alienta a un mayor aprendizaje además de una mejor **calidad de código** y por ende también calidad del producto que se está construyendo.

En lo referido al punto 3, también como una recomendación se agregaría la posibilidad de incluir un "*Taller de Scrum*" con un experto en el tema, la ejecución de este taller estará muy relacionado con el alcance que tengan el desarrollo y los costos que el mismo maneje. Sin embargo, sería de mucha utilidad para equipos sin experiencia y con desconocimiento en algunas prácticas tales como las retrospectivas.

Por último, con respecto al punto 4, la idea de agregar más métricas iría muy de la mano con tener una manera de evaluar el progreso y buena evolución del proyecto de forma más amplia y abarcativa. Un ejemplo de métricas que podrían agregarse al conjunto de métricas recomendadas por la guía son las relacionadas con el análisis estático de código ("*Static code-analysis metrics*" (36))

[75] *"(...)es el acto de convocar consciente y sistemática con uno de los compañeros programadores para revisar el código de cada uno en busca de errores (...)"* https://smartbear.com/learn/code-review/what-is-code-review/

5.4 Resumen del capitulo

En este capítulo analizamos los resultados obtenidos en el capítulo 4, donde pusimos en práctica la guía propuesta en el capítulo 3 mediante una simulación con un caso de estudio. A partir de dichos resultados enunciamos las ventajas y desventajas encontradas, así como también resaltamos el aporte para los roles de QA, DEV, BA.

Por último, enumeramos algunas mejoras a la guía que surgieron a partir de la simulación y su análisis.

En la Figura 50 podemos ver el resumen gráfico de lo realizado en este capítulo.

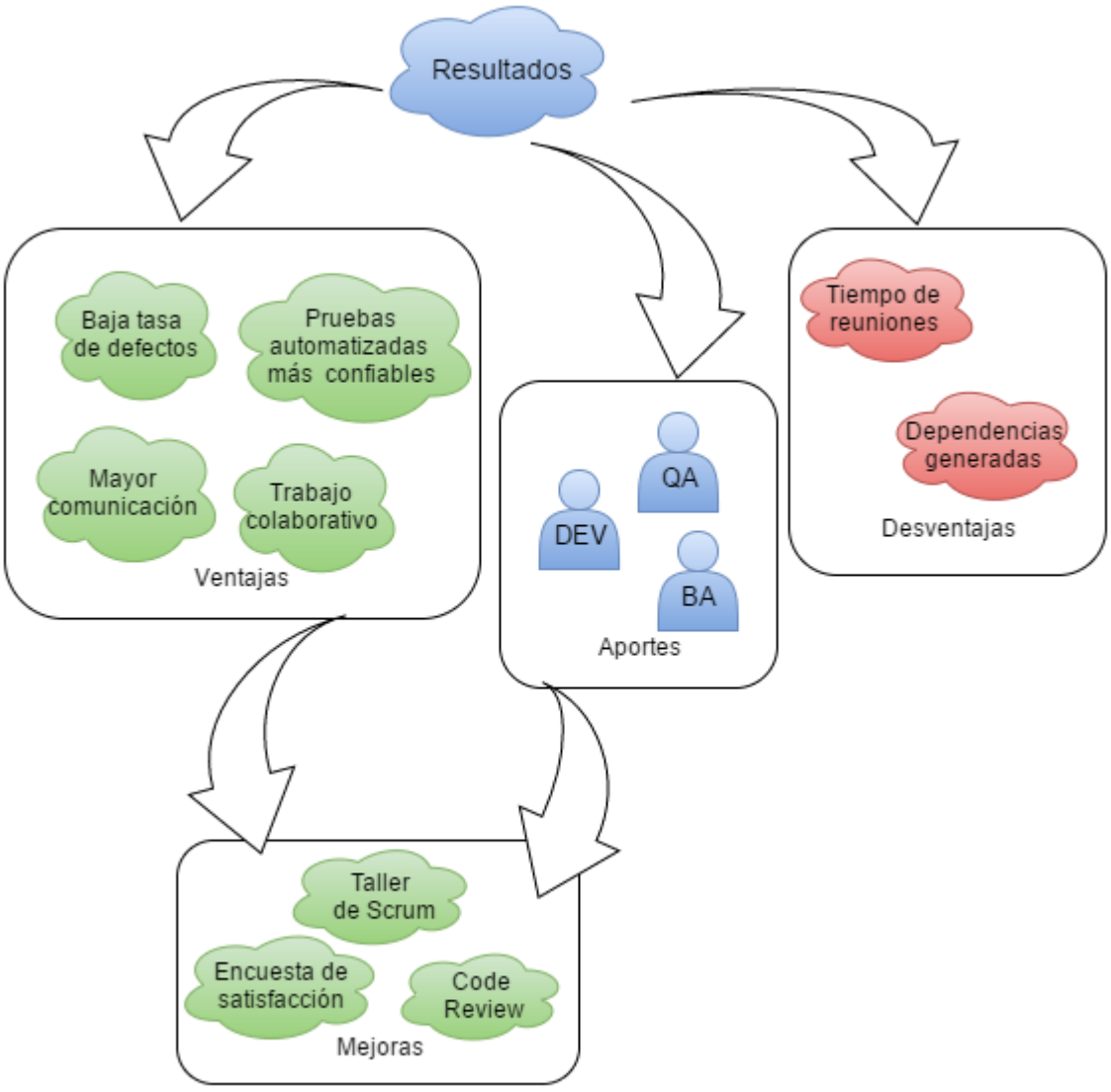


Figura 50. Resumen Gráfico

Capítulo 6. Conclusiones y trabajos futuros

"*One worthwhile task carried to successful conclusion is better than 50 half-finished task*", B.C Forbes

6.1 Conclusiones

Ser capaces de modernizar y enriquecer un desarrollo semi-ágil mediante el uso de una metodología en particular, sin lugar a dudas no es una tarea fácil. Teniendo en cuenta que tanto BDD como sus antecesoras han sido creadas hace ya más de 10 años y durante todo ese tiempo han evolucionado y se han ido adaptando a los días que corren, complejiza aún más la tarea. La pregunta que surge entonces es: "es posible contar con un manual práctico, sencillo que nos indique cómo llevar a cabo esta difícil tarea?". Lo que buscamos en esta tesina fue poder responder: "¡Si, existe! Y acá la presentamos". El fruto de nuestra búsqueda fue la guía que planteamos y los resultados de la simulación corroboran nuestros principales objetivos. Si bien no salimos a la búsqueda de una solución mágica ni tampoco de reinventar la rueda, nos atrevemos a decir (basándonos en Gherkin) que lo que pretendemos es:

Dado *un desarrollo* ***semi-ágil***
Cuando *se aplica la guía práctica propuesta*
Entonces *mejoramos la* ***comunicación*** *del equipo*
Entonces *generamos* ***especificaciones ejecutables***
Entonces *obtenemos un* ***documentación "Viva"***

Podemos ver que resulta un poco ambiciosa la frase anterior y, sin embargo, resume de manera eficaz el trabajo realizado en la presente tesina. Lo que se espera es que la guía resulte beneficiosa para quienes incursionan en el camino de BDD y de las metodologías ágiles, en casos en los cuales se cuente con un desarrollo semi-ágil.

Como conclusión general y a su vez final, podemos decir que nuestro objetivo fue alcanzado ya que, mediante la construcción de la guía y su puesta en práctica con la simulación, expusimos una forma en la cual es posible utilizar BDD para mejorar algunos aspectos principales de un desarrollo semi- ágil y orientarlo hacia las buenas prácticas ágiles.

6.2 Trabajos futuros

La guía presentada fue la versión 1.0, de mano de las mejoras mencionadas en el capítulo anterior se buscará obtener la versión 2.0. Del mismo modo la simulación llevada a cabo fue el primer muestreo de resultados, con lo cual para poder validar la próxima versión de la guía es que se plantea también realizar una nueva puesta en práctica, esta vez en un caso en donde se pueda realizar un seguimiento completo y detallado. Esta nueva fuente de resultados permitirá una análisis más profundo y preciso.
Se asume también que de esa nueva implementación surgirán nuevas mejoras y modificaciones a la guía, que deberá ser adaptada nuevamente.

Podemos resumir los trabajos a futuro en 2 puntos:

- Integración de las mejoras planteadas.
- Nueva puesta en práctica en un caso de estudio más complejo.

Referencias bibliográficas

1. Gojko Adzic, "*Specification by Example*" ,Manning Publications Co, 2011
2. Gojko Adzic, "*Bridging the Communication Gap. Specification by example and agile acceptance testing*", Neuri Limited, 2009.
3. David Chelimsky ,"*The RSpec Book, Behaviour-Driven Development with RSpec, Cucumber, and Friends*" ,Pragmatic Bookshelf, 2010
4. John Ferguson Smart,"*BDD in Action:Behavior-Driven Development for the whole software lifecycle*" Manning Publications Co, 2015
5. Matt Wynne-Aslak Hellesøy, "*The Cucumber book, Behaviour-Driven Development for Testers and Developers*", Pragmatic Bookshelf,2012
6. Lisa Crispin-Janet Gregory, "*Agile testing: a practical guide for testers and agiles teams*", Pearson Education, Inc, 2009
7. Moisés Carlos Fontela, "*Estado del arte y tendencias en Test-Driven Development*" , Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina, 2011
8. Kent Beck, "*Test Driven Development: By Example*", Addison-Wesley Professional, 2002.
9. Manifiesto Ágil (2001).
   Disponible :http://www.agilemanifesto.org/iso/es/
10. Scott Millett- Nick Tune, "*Patterns, Principles, and Practices of Domain-Driven Design*", John Wiley & Sons, Inc, 2015
11. D. North. (2006). *Introducing BDD* [Online].
    Disponible: https://dannorth.net/introducing-bdd/ (Consultado Mayo 2016)
12. Liz Keogh (2005-2015) Artículos relacionados con BDD.
    Disponible https://lizkeogh.com/category/bdd/(Consultado Mayo 2016)
13. DDD Community .
    Disponible:http://dddcommunity.org/ (Consultado Mayo 2016)
14. James Carr . TDD Anti-Patterns,
    Disponible:http://blog.james-carr.org/2006/11/03/tdd-anti-patterns/
15. Scott Ambler. *Introduction to Test Driven Development* (TDD),
    Disponible: http://agiledata.org/essays/tdd.html (Consultado Mayo 2016)
16. Lasse Koskela, "*Test Driven Practical TDD and Acceptance TDD for Java Developers*" Manning Publications Co, 2007(Capitulo 9)
17. Ron Quartel, "*Acceptance Test-Driven Development: Are We Flogging a Dead Horse?*" (2013)
    Disponible:http://www.solutionsiq.com/acceptance-test-driven-development-are-we-flogging-a-dead-horse/.
18. Srini Penchikala, *"Domain Driven Design and Development In Practice"* (2008)
    Disponible:https://www.infoq.com/articles/ddd-in-practice.
19. Dan North. "*How to sell BDD*".(2009)
    Disponible:https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business
20. Liz Keogh,"*ATDD vs. BDD, and a potted history of some related stuff*" (2011)
    Disponible:https://lizkeogh.com/2011/06/27/atdd-vs-bdd-and-a-potted-history-of-some-related-stuff/
21. Konstantin Kudryashov. "*The beginner's guide to BDD* "(Consultado Junio 2016)
    Disponible:http://inviqa.com/insights/bdd-guide
22. Jan Molak. "*The Five Stages of BDD (and Agile) Adoption*" (Mayo 2016)
    Disponible:https://dzone.com/articles/the-five-stages-of-bdd-and-agile-adoption
23. Gabo Esquivel. *Differences Between TDD, ATDD and BDD* (Julio 2014)
    Disponible:http://gaboesquivel.com/blog/2014/differences-between-tdd-atdd-and-bdd/

24. Belatrix Software.Whitepapers: *Successful Project with Behavior-Driven Development* (Consultado Junio 2016)
Disponible:http://www.belatrixsf.com/whitepapers/whitepapers-successful-project-with-behavior-driven-development/
25. Sergey Larionov, *Impact mapping: business success in software development* (2014)
Disponible:http://blog.octo.com/en/impact-mapping-business-success-in-software-development/
26. Dami Buonamico, *Visual Story Mapping Aplicado*
Disponible:http://www.caminoagil.com/2013/02/visual-story-mapping-aplicado.html
27. Martín Alaimo, KLEER. *Análisis, Estimación y Planificación Ágil* (Consultado en Julio 2016)
Disponible:http://media.kleer.la/kleer-scrum-estimation-planning-es.pdf
28. Luane Silvestre, 2015. ¿Cuál es la importancia del feedback y cómo aprovecharlo?
Disponible:https://www.tiendanube.com/blog/cual-es-la-importancia-del-feedback-y-como-aprovecharlo/
29. Jason Yip, "*It's Not Just Standing Up: Patterns for Daily Standup Meetings*" (Consultado en Julio 2016)
Disponible:http://www.martinfowler.com/articles/itsNotJustStandingUp.html
30. Scrum-institute site (Consultado en Julio 2016)
Disponible: http://www.scrum-institute.org/
31. Juan Gabardini , KLEER. *Técnicas para la realización de Retrospectivas* (Consultado en Julio 2016)
Disponible:http://media.kleer.la/kleer-tecnicas-de-retrospectivas-es.pdf
32. Rishi Devendra. *Key Elements of the Sprint Retrospective* (Consultado en Julio 2016)
Disponible:https://www.scrumalliance.org/community/articles/2014/april/key-elements-of-sprint-retrospective
33. SG Buzz, Beneficios de la Automatización de Pruebas (Consultado en Julio 2016)
Disponible: http://sg.com.mx/content/view/683
34. Fowler Martin,"Continuous Integration", 2006.
Disponible: http://www.martinfowler.com/articles/continuousIntegration.html
35. Fowler Martin,"*Continuous Delivery*",2013
Disponible: http://martinfowler.com/bliki/ContinuousDelivery.html
36. Dave Nicolette, "*Software Development Metrics*". Manning Publications Co, 2015.
37. Jaime Carril, Charla: "Documentación viva, cómo y para qué en Agilidad".2015
Disponible en: http://foro.chileagil.cl/t/charla-documentacion-viva-como-y-para-que-en-agilidad/697
38. Cyrille Martraire, "*Living documentation"* Lean Publishing,2014-2016
Version gratuita disponible(Julio 2016): https://leanpub.com/livingdocumentation
39. Raghu Angara, "*Agile Metrics - Running Tested Feature*s", Junio 2013.
Disponible en:
http://www.infosysblogs.com/applicationservices/2013/06/agile_metrics_-_running_tested.html
40. Pawel Brodzinski, "*Cumulative Flow Diagram*", Julio 2013.
Disponible en: http://brodzinski.com/2013/07/cumulative-flow-diagram.html
41. Catia Oliveira, *"How to Calculate and Use Velocity to Help Your Team and Your Projects"* , Febrero 2014.

Disponible en:
https://www.scrumalliance.org/community/articles/2014/february/velocity
42. Seb Rose, Matt Wynne & Aslak Hellesoy, "*The Cucumber for Java Book, Behaviour-Driven Development for Testers and Developers*", Pragmatic Bookshelf,2015.
43. Leonardo S. De Seta, "*Revisiones de código (buenas o malas)*", Agosto 2008. (Consultado octubre 2016)
Disponible en:http://www.dosideas.com/noticias/metodologias/162-revisiones-de-codigo-buenas-o-malas.html

Listado de Figuras