>>> Eugenia Bahit

Scrum & eXtreme Programming

>>> para programadores

Comparte el conocimiento

Eres libre de:

- **Copiar**, **distribuir** y **compartir** este libro

Bajo las siguientes condiciones:

- Reconocer y respetar la autoría de la obra
- No hacer uso comercial de ella
- No alterar el contenido

Índice General

Introducción a la gestión de proyectos de desarrollo de Software

¿Qué es el Desarrollo Ágil?

Así como existen métodos de gestión de proyectos tradicionales, como el propuesto por el **Project Management Institute**[1] más conocido como **PMI®** podemos encontrarnos con una rama diferente en la gestión de proyectos, conocida como Agile. El desarrollo ágil de software, no es más que una metodología de gestión de proyectos **adaptativa**, que permite llevar a cabo, proyectos de desarrollo de software, **adaptándote a los cambios** y evolucionando en forma conjunta con el software.

Un pantallazo general sobre la gestión de proyectos

A lo largo de la historia del Software, muchos proyectos han fracasado y aún continúan haciéndolo.

Implementar una metodología de gestión, básicamente **nos permite organizar mejor un proyecto y obtener mejores resultados del software** entregado a al cliente, evitando los fracasos.

Pero **¿por qué fracasan los proyectos?** Sin dudas, los "porque" podrían llegar a ser casi infinitos, si nos pusierámos demasiado exigentes. Sin embargo, hay tres motivos de los que ningún fracaso está exento:

1 Sitio Web Oficial del PMI: http://www.pmi.org/

1. El proyecto lleva más tiempo del que se había planificado;
2. El proyecto lleva más dinero del que se había pautado invertir;
3. Las funcionalidades del Software no resultan como se esperaba.

Las razones por las cuales, los tres motivos anteriores generan el fracaso, pueden resumirse una simple frase: **no se puede prever con exactitud un proyecto de desarrollo de Software**.

Y ¿en qué se funda esa afirmación? Pues la respuesta está en otra pregunta: **¿en qué se diferencia un Software de otro producto?** La respuesta que primero nos viene a la mente es que **el Software es un producto no tangible que no puede conocerse hasta no estar concluido y poder probarse**.

No obstante ello, el Software tiene grandes ventajas frente a otros productos:

- No puedo construir una silla a medias para probarla ya que la necesito completa. Sin embargo, puedo construir una determinada funcionalidad del Software para poder probarla.
- Tirar abajo un edificio recién terminado, porque no satisface las expectativas de los dueños, sería una gran locura, ya que requeriría invertir tres veces el costo fijado para reconstruir el edificio. Sin embargo, desarrollando un Software con buenas prácticas de programación, rehacer funcionalidades o desarrollar nuevas, es tan factible como simplemente agregar

cortinas a las ventanas de un edificio.

Y estas ventajas, deben ser aprovechadas. De lo contrario, **insistir en gestionar un proyecto de desarrollo de Software de manera tradicional, implicaría caer en un capricho infantil** sin fundamentos válidos. Sino, mira los siguientes números:

> En el año **2009**, el Standish Group (www.standishgroup.com) elaboró un informe llamado **Chaos Report**, el cual arrojó como resultado que solo el **32% de los proyectos de desarrollo de Software han sido exitosos**. El mismo informe, indica que más del **45% de las funcionalidades del Software entregadas al usuario, jamás se utilizan**.
>
> En el año **2004**, PricewaterhouseCoopers[2] elaboró un informe mediante el cual se investagaron a **200 empresas en 35 países**, arrojando como resultado que **mas del 50% de los proyectos fracasan** (sobre la base de 10.640 proyectos, representando 7.000 millones de dólares) .

Por lo tanto, optar por una u otra metodología de gestión, no puede basarse jamás, en un mero capricho, ya que es mucho lo que está en juego.

Diferenciando las metodologías de gestión

Una metodología de gestión de proyectos, consiste en la

2 http://es.wikipedia.org/wiki/PricewaterhouseCoopers

convención de prácticas, métodos, principios, técnicas y herramientas cuya principal utilidad es la de otorgar un mejor rendimiento del equipo de trabajo y por sobre todo, permitir la obtención de mejores resultados en lo que se produce durante el proyecto (en nuestro caso: software).

Si bien las metodologías de gestión, podrían resumirse en solo dos enfoques (o tendencias): el enfoque ágil (en el cual se centra este curso) y el enfoque predictivo (propuesto por el Project Management Institute), cada uno de estos enfoques, presenta una diversa variedad de propuestas que pueden aplicarse. En lo que respecta al enfoque ágil, existen un gran número de propuestas, de las cuales, las más utilizadas y estadísticamente con mejores resultados son Scrum (se pronuncia "scram"), Kanban (se pronuncia tal cual se escribe) y eXtreme Programming, más conocida como XP (se pronuncia por sus siglas en inglés "ex-pi" o en castellano "equis pe").

Los enfoques ágiles se diferencian (y también guardan ciertos aspectos en común) de los predictivos, básicamente por la forma de abordaje de un proyecto.

El enfoque predictivo, es aquel que plantea el abordaje estricto de un proyecto, sobre la base del cumplimiento de tres aspectos predefinidos al comienzo del proyecto: alcance, costo y tiempo; mientras tanto, el enfoque ágil, plantea los proyectos desde el cumplimiento de un objetivo más amplio: entregar software con el mayor valor posible.

Esto, puede explicarse sino, comentando que la principal diferencia es que el enfoque predictivo, propone la definición detallada del alcance del proyecto, y la estipulación precisa de tiempo y costo, mientras que el enfoque ágil, plantea la

definición de un alcanse global al comienzo, para luego ir incrementándolo en las diversas iteraciones (cada una de las cuales, supone la entrega de un software 100% funcional).

Tipos de Proyecto

Si bien cada proyecto es único y tiene sus propias características y restricciones, podemos establecer una diferencia global, entre 7 tipos de proyectos de desarrollo de Software principales, como se define a continuación.

Desarrollo de un nuevo sistema informático

Es el proyecto mediante el cual, se desea automatizar un proceso de negocio. Requiere construir una aplicación desde cero, como por ejemplo, desarrollar un nuevo sistema de finanzas, el sitio Web corporativo de una compañía, etc.

Este tipo de proyectos, suelen ser los más complejos desde el punto de vista del relevamiento funcional necesario que demanda una aplicación hasta ahora no conocida. Sin embargo, iniciar de cero una aplicación, presenta al mismo tiempo, una gran ventaja para el equipo de desarrolladores, ya que no deberán lidiar con códigos ajenos o malas prácticas de programación que arrastran bugs desde largo tiempo atrás.

Desarrollo de nuevas funcionalidades

Al igual que en el caso anterior, son proyectos donde también se requiere automatizar un proceso de negocio, no contemplado en una aplicación existe. En este caso puntual, dicha automatización, se transforma en una funcionalidad que deberá ser incorporada a una aplicación ya existente. Es el caso, por ejemplo, de agregar la emición de códigos de barras a un sistema de control de Stock.

En este tipo de proyectos, la aplicación existente puede representar una ventaja si se ha desarrollado bajo buenas prácticas de programación, tendientes a sostener el mantenimiento y evolución del sistema, o una desventaja, cuando se haya desarrollado sin tener en cuenta su escalabilidad.

Reingeniería de un sistema

Este tipo de proyectos son mucho más puntuales y complejos, que los dos anteriores. En estos casos, se pretende reemplazar un sistema actual, por uno con características similares pero más específicas y posiblemente -en gran parte de los casos- se requiera migrar de tecnología a una más moderna, con mejor soporte o simplemente más robusta. Un ejemplo de ello, sería migrar un sistema desarrollado en los '80 en Cobol, a Python.

Mantenimiento evolutivo

Este tipo de proyectos, son los que más acercan al Desarrollo de Nuevas Funcionalidades, pero con una gran diferencia: se enfocan en la evolución de una funcionalidad de Software existente. Dicha evolución, generalmente se centra en agregar

a una funcionalidad existente (por ejemplo, a la funcionalidad de generación de códigos de barra del sistema de control de stock), una o más características adicionales, tendientes a enriquecer dicha funcionalidad (por ejemplo, que la emisión de códigos de barra, guarde un log de códigos de barra emitidos, que se sume a los reportes de los productos).

Mantenimiento adaptativo

En estos casos, la complejidad del proyecto, estará atada al tipo de adaptación requerida. Son proyectos donde generalmente, se necesita adaptar el Sofwtare existente, a un nuevo entorno de hardware, versión de Sistema Operativo, del lenguaje informático con el que se ha escrito la aplicación o del framework.

Un ejemplo de ello, podría ser adaptar un sistema desarrollado en Python 2.x a Python 3.x, o un complemento de Firefox 6.x a Firefox 10, etc.

Mantenimiento preventivo

Este tipo de proyectos, suelen ser los que mayor resistencia presentan, muchas veces, por parte de los dueños de producto y en otras, por parte de los desarrolladores.

Los mantenimientos preventivos son aquellos que no generan cambios visibles a la aplicación y su objetivo es mejorar cuestiones inherentes al rendimiento interno del sistema, como podrían ser refactorizaciones de código para eliminar redundancia, generación de logs de acceso al servidor o la base de datos, instalación de paquetes de seguridad, desarrollo de

script para prevención de ataques de diversos tipos (XSS, Fuerza bruta, DDoS, etc.).

Mantenimiento correctivo

El mantenimiento correctivo está generalmente ligado de forma directa, a las fallas actuales de un sistema y su objetivo, es corregir bugs (bug-fixing).

La octava clasificación

Otro tipo de proyectos que puede definirse, son los destinados a la investigación cuyos objetivos pueden estar centrados, en el avance de pruebas de concepto, una tecnología o producto.

Abordaje de un proyecto de construcción de Software

Como comentamos anteriormente, un proyecto puede abordarse de manera tradicional o adaptativa, en la cual nos centraremos en este curso. Si bien hemos marcado algunas diferencias entre ambas metodologías de gestión -y continuaremos ampliando el tema-, una de las diferencias fundamentales, es la forma de abordar la construcción de un Software.

La construcción de un Software, tiene un ciclo de vida implícitamente definido, que consiste en:

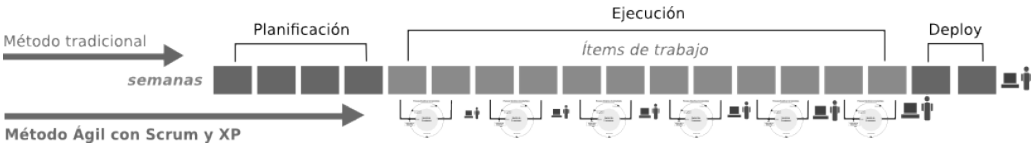
1. El **relevamiento** de necesidades de negocio que requieren ser automatizadas;
2. El **análisis** de los requerimientos técnicos, tecnológicos y funcionales;
3. El **diseño** de los procesos de automatización;
4. La **construcción** de los procesos de automatización;
5. Las **pruebas** de lo construido;
6. La **implementación** de lo construido.

Las **metodologías tradicionales** (o predictivas), **encaran las fases** que componen el ciclo de vida del desarrollo de Software, **de manera sucesiva**. Es decir, que una fase sucede a otra, y cada fase, ocupa un espacio lineal en el.

Relevamiento	*Análisis*	*Diseño*	*Construcción*	*Pruebas*	*Implementación*

En cambio, **las metodologías ágiles, solapan estas etapas,** permitiendo ahorrar tiempo, evitando la dependencia (cada etapa es independiente de la otra) y haciendo del ciclo de vida, un proceso iterativo (se inicia con el relevamiento, se finaliza con la implementación y se vuelve a comenzar para abordar nuevas funcionalidades).

Comparación en tiempo, entre ágil y predictivo:



El agilismo y su Manifiesto

Así como el PMI, propone a través de la PMBOK®[3] Guide una serie de normas, procesos y herramientas para mejorar la gestión de un proyecto, las metodologías ágiles poseen su propio manifiesto: el **Agile Manifesto** (o Manifiesto Ágil en español), el cual, a diferencia de la guía PMBOK ® en muy pocas líneas, se encarga de definir una serie de **principios y valores que rigen al agilismo**.

Los valores del agilismo

En el Manifiesto Ágil, podemos leer los cuatro valores que impulsa el agilismo:

Individuos e interacciones sobre procesos y herramientas

Mientras que las metodologías tradicionales, centran su esfuerzo en definir procesos y herramientas que regulen la gestión de un proyecto, las metodologías ágiles, prefieren valorar la idoneidad de cada individuo, depositando en ésta, la confianza necesaria para lograr una buena comunicación, fluída e interactiva entre todos los participantes del proyecto.

3 Versión en español de la PMBOK Guide en formato PDF: http://gio.uniovi.es/documentos/software/GUIA_PMBok.pdf

Software funcionando sobre documentación extensiva

Mientras que las metodologías predictivas (o tradicionales), invirtien una gran cantidad de tiempo y dedicación al desarrollo de documentos que describan detalladamente el alcance de un sistema así como el diseño de prototipos o bocetos que intenten proveer una imagen visual del futuro Software, las metodologías ágiles proponen destinar ese tiempo al desarrollo de la aplicación, centrando el mismo, en pequeñas partes 100% operativas y funcionales, que por un lado, otorguen mayor valor al negocio y por otro, permitan probar Software real (no prototipos ni bocetos).

Colaboración con el cliente sobre negociación contractual

Las metodologías predictivas, plantean un sistema de gestión de riesgos, mediante el cual, cada cambio sugerido por el cliente, debe superar una serie de procesos administrativos, tales como la ampliación/modificación del alcance, la reelaboración de presupuestos y una nueva estipulación de tiempo de entrega lo que conlleva a la inevitable modificación de un cronograma de entregables. A diferencia de esto, las metodologías ágiles, prefieren integrar al cliente al proyecto, facilitando la comunicación directa de éste con el equipo de desarrollo, a fin de generar un ambiente de colaboración mutua, donde los "cambios" sugeridos por el cliente, no signifiquen un riesgo, sino un "aporte" al valor del Software.

Respuesta ante el cambio sobre seguir un plan

En la gestión tradicional, antes de comenzar el desarrollo del Software (llamado "proceso de ejecución"), se elabora un detallado plan donde se describe un alcance funcional

minucioso y preciso, así como también, se pautan fechas de entregas que se pretende, no sufran variaciones.

Cuando en pleno proceso de ejecución, el cliente solicita algún cambio, al existir un plan previsto que no puede desviar su rumbo, surgen una serie de conflictos derivados del cambio, los cuales conllevan a reelaborar el plan inicial y en consecuencia, extender o prorrograr los plazos de entrega.

Por el contrario, las metodologías ágiles proponen adaptarse a los cambios sugeridos por el cliente, lo que significa, que a la modificación propuesta, no se sumarán escoyos administrativos, sino por el contrario, se acelerarán los procesos, actuando en consecuencia de lo que ha sido pedido.

Vale aclarar, que si bien las metodologías ágiles, valoran los elementos de la derecha (derivados de las propuestas tradicionalistas), otorgan un mayor valor a los de la izquierda. Y ésto, tiene su fundamento en los doce principios, que también se exponen en el Manifiesto Ágil.

Los doce principios del agilismo

En el Manifiesto Ágil, podemos leer los doce principios en los cuáles se argumentan los valores del Manifiesto. Estos son:

Principio #1: Satisfacer al cliente mediante la entrega temprana y continua de software con valor.

El agilismo, propone desarrollar el Software de manera iterativa e incremental. Esto significa, que los requerimientos

funcionales del producto, son fragmentados en lo que se denomina "Historias de Usuario" y, basándose en la prioridad del cliente y el esfuerzo disponible para el desarrollo, se hace una selección de historias de usuario a ser desarrolladas en un período de tiempo fijo (por ejemplo, 2 semanas) y al finalizar dicho período, las Historias de Usuario habrán sido convertidas en Software que puede utilizarse y por lo tanto, se entrega al cliente. Este ciclo, es continuo (al finalizar un ciclo comienza el siguiente) y ésto, genera una entrega rápida y continuada al cliente.

Principio #2: Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

El agilismo respeta la idoneidad del cliente como tal. Acepta que éste, es el único capaz de decir, cuáles funcionalidades requiere el Software, ya que como dueño y/o usuario del mismo, es el único que conoce verdaderamente su negocio. En este sentido, el agilismo recibe los cambios, no como un capricho incómodo, sino con la humildad y el orgullo de saber que dicho cambio, hará del Software un mejor producto.

Principio #3: Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.

En este sentido, extiende el Principio #1, proponiendo como períodos de tiempo fijo para la entrega temprana y continua, ciclos de 2 a 8 semanas. Principio #4: Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto. Este cuarto principio, marca

una diferencia radical e inigualable con las metodologías tradicionales. El Manifiesto Ágil, de forma explícita, propone incluir al cliente en el proceso de desarrollo del Software, sumándolo al proyecto, como parte imprescindible.

Principio #5: Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.

Si uno busca principios claros y de alto contenido humano y social, este principio es el ejemplo de lo que se busca. Por un lado, así como el agilismo respeta la idoneidad del cliente, propone que dicho respeto sea muto y se reconozca la idoneidad del equipo de desarrollo, aceptándolos como los únicos capaces de decidir "el cómo". Y este "cómo" incluye tanto a la autogestión del tiempo como a las técnicas de programación que serán utilizadas.

Por otro lado, abarca un concepto más ampliamente humano cuando se refiere a la motivación de los individuos, su entorno y el dar apoyo. En este sentido, las metodologías ágiles, humanizan el entorno de trabajo, promoviendo actividades de motivación (como los Coding Dojo, por ejemplo) e incentivando al respeto tanto por las libertades individuales como colectivas, de los integrantes del equipo así como también, el respeto por su dignidad profesional, como pilar fundamental de un entorno de trabajo colaborativo, libre de malas intensiones y presiones poco humanas.

Principio #6: Conversación cara a cara.

El agilismo, plantea que la forma más eficaz de comunicarse entre quienes participan del proyecto, es "cara a cara". Y esto se refiere, a la erradicación de intermediarios: todo cambio y

toda planificación, se debe realizar entre el principal interesado en el producto (cliente) y las personas que se encargarán de desarrollarlo e implementarlo (equipo de desarrolladores).

Principio #7: El software funcionando es la medida principal de progreso.

El agilismo, plantea que no puede medirse el éxito de un proyecto, solo en base al cumplimiento efectivo de un plan, puesto que éste, incluso cuando se cumpla al pie de la letra, no garantiza que el Software, satisfaga al 100% las expectativas del cliente. Así es, que como elemento de medición de éxito y progreso, se propone el Software que haya sido entregado y se encuentre en pleno funcionamiento.

Principio #8: Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.

El agilismo prioriza la calidad del Software por sobre las limitaciones de tiempo. En este sentido, el agilismo plantea que el desarrollo de un Software debe ser constante y continuado, permitiendo al cliente a través de las entregas tempranas, poder ir ampliando su campo de alcance funcional, en la medida que a aplicación, va siendo utilizada.

Principio #9: La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.

Un punto clave en el agilismo, es la búsqueda constante, de la perfección tecnológica, proponiendo buenas prácticas de programación que aseguren tanto la mantenibilidad del Software, como su evolución y escalabilidad.

Principio #10: La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.

Al proponer el desarrollo en ciclos con entregas iterativas, el agilismo adquiere una gran ventaja frente a las metodologías tradicionales: facilita el análisis y la revisión retrospectiva de los métodos implementados en ciclos anteriores. Esto, permite ir corrigiendo sobre la marcha, ciertos impedimentos que hacen que el trabajo se estanque o avance lentamente. Al generar Software de manera iterativa, se facilita la mejora de métodos, ampliando así, las posibilidades de maximar la cantidad de trabajo realizado en cada ciclo.

Principio #11: Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.

Nuevamente, el agilismo hace aquí, hincapié en la idoneidad profesional del equipo de desarrollo. Es sabido que en proyectos gestionados de manera tradicional, un Líder de Proyecto (Líder de Equipo o Gerente de Proyecto), es quien elabora "los diseños" indicando "el cómo" debe desarrollarse un software. Y también es cierto, que quien gestiona un proyecto, no siempre cuenta con la idoneidad profesional necesaria, para diseñar la Arquitectura de un Sistema, el modelo de datos o su interfaz gráfica.

Es por ello, que el agilismo plantea que en un equipo auto-gestionado, sus miembros tienen la facultad de decir, que parte del trabajo se encargarán de llevar adelante, basándose en sus propias cualidades profesionales.

Principio #12: A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

Este principio, está estrechamente ligado al décimo principio, pero en el sentido, no de maximar la cantidad de trabajo, sino la calidad del mismo.

Conociendo Scrum

Hemos comentado anteriormente, que **Scrum es una metodología ágil para la gestión de proyectos** relacionados con la construcción (desarrollo) de Software. Veremos ahora en detalle, de que se trata esto de "Scrum". Pete Deemer, Gabrielle Benefield, Craig Larman y Bas Vodde , definen Scrum en el libro The Scrum Primer (2009), con los siguientes párrafos:

> "Scrum es un marco de trabajo iterativo e incremental para el desarrollo de proyectos, productos y aplicaciones. Estructura el desarrollo en ciclos de trabajo llamados Sprints. Son iteraciones de 1 a 4 semanas, y se van sucediendo una detrás de otra. Los Sprints son de duración fija – terminan en una fecha específica aunque no se haya terminado el trabajo, y nunca se alargan. Se limitan en tiempo. Al comienzo de cada Sprint, un equipo multi-funcional selecciona los elementos (requisitos del cliente) de una lista priorizada. Se comprometen a terminar los elementos al final del Sprint. Durante el Sprint no se pueden cambiar los elementos elegidos. [...]" (The Scrum Primer, 2009, pág. 5)

Un **Sprint** tiene dos características fundamentales: a) una duración fija entre 1 a 4 semanas ; b) cada Sprint se ejecuta de forma consecutiva (sin tiempo muerto entre un sprint y otro).

El objetivo principal es transformar un conjunto de ítems requeridos por el cliente en un incremento de funcionalidad 100% operativa para el software.

> "Todos los días el equipo se reúne brevemente para informar del progreso, y actualizan unas gráficas

> sencillas que les orientan sobre el trabajo restante. Al final del Sprint, el equipo revisa el Sprint con los interesados en el proyecto, y les enseña lo que han construido. La gente obtiene comentarios y observaciones que se puede incorporar al siguiente Sprint. Scrum pone el énfasis en productos que funcionen al final del Sprint que realmente estén "hechos"; en el caso del software significa que el código esté integrado, completamente probado y potencialmente para entregar [...]" (The Scrum Primer, 2009, pág. 5)[4]

Según **Jeff Sutherland**[5], uno de los creadores de Scrum y del **método de control empírico de procesos** en el cual se basa dicha metodología, éste se sostiene en la implementación de tres pilares: **transparencia** (visibilidad de los procesos), **inspección** (periódica del proceso) y **adaptación** (de los procesos inspeccionado)[6] [7].

El Marco de Trabajo de Scrum

En un sentido amplio, el marco de trabajo de Scrum se compone de una serie de reglas, que definen roles que integran los equipos, artefactos necesarios para los procesos, bloques de tiempo preestablecidos y ceremonias que deben respetarse.

Los equipos de Scrum definen tres **roles** bien diferenciados:

4 The Scrum Primer – versión en español: http://assets.scrumfoundation.com/downloads/3/scrumprimer_es.pdf?1285932063

5 Sitio Web de Jeff Sutherland: http://jeffsutherland.com/

6 Ken Schwaber, Jeff Sutherland . Scrum Guide (2008)

7 Scrum Guide - versión en español: http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%20ES.pdf

1. El **Scrum Master**, responsable de asegurar los procesos;
2. El **Dueño de Producto**, responsable de maximizar el valor del producto;
3. El **Equipo**, responsable de realizar el trabajo.

La definición de bloques de tiempo iterativos (de 2 a 4 semanas), está destinada a crear continuidad y regularidad en las cuáles, se basarán las seis **ceremonias** (reuniones) que aseguran el cumplimiento de objetivos:

1. Reunión de Planificación de la Entrega
2. Reunión de Planificación del Sprint
3. El Sprint – corazón de Scrum -
4. Reunión diaria
5. Reunión de revisión
6. Reunión de Retrospectiva

Cómo herramientas aplicables a todos los procesos anteriores, Scrum emplea cuatro **artefactos**:

1. **Backlog de Producto** (una lista priorizada de todo lo que requiere el software)
2. **Backlog de Sprint** (una lista de tareas necesarias para convertir parte del Backlog de Producto, en incremento de funcionalidad del Software)
3. **Scrum taskboard** (un tablero físico que permite la transparencia de los ítems de trabajos pendientes, en curso y terminados de un mismo Sprint)
4. **Diagrama de Burndow** (un gráfico que permite medir

visualmente, el progreso de los items del Backlog de Producto)

> "[...] Las reglas sirven de unión para los bloques de tiempo, los roles y los artefactos de Scrum [...]. Por ejemplo, una regla de Scrum es que sólo los miembros del equipo - la gente comprometida a convertir el Product Backlog en un incremento - pueden hablar durante un Scrum Diario [...]" (Scrum Guide, 2008, Pág. 5)

Los Roles en Scrum

Como comentamos anteriormente, Scrum define tres roles bien diferenciados: el Scrum Master, el Dueño de Producto y el Equipo. En este capítulo, veremos en detalle, las responsabilidades y características de cada uno de ellos.

El Dueño de Producto (Product Owner) en Scrum

El Dueño de Producto es la única persona autorizada para decidir sobre cuáles funcionalidades y características funcionales tendrá el producto. Es quien representa al cliente, usuarios del software y todas aquellas partes interesadas en el producto.

> "El propietario del producto puede ser un miembro del equipo, que también participe en las tareas de desarrollo. Esta responsabilidad adicional puede disminuir la capacidad del Propietario del Producto de trabajar con los interesados del proyecto o producto. Sin embargo, el propietario del producto no puede ser nunca el ScrumMaster. "
> (Scrum Guide, 2008, Pág. 7)

Funciones y responsabilidades

- Canalizar las necesidades del del negocio, sabiendo "escuchar" a las partes interesadas en el producto y transmitirlas en "objetivos de valor para el producto", al scrum team.
- Maximizar el valor para el negocio con respecto al Retorno de Inversión (ROI), abogando por los intereses del negocio.
- Revisar el producto e ir adaptándole sus funcionalidades, analizando las mejoras que éstas puedan otorgar un mayor valor para el negocio.

Aptitudes que debe tener un Dueño de Producto

- Excelente facilicidad de comunicación en las relaciones interpersonales
- Excelente conocimiento del negocio
- Facilidad para análisis de relaciones costo/beneficio
- Visión de negocios

El Scrum Master

El Scrum Master es el alma mater de Scrum. Un error frecuente es llamarlo "líder", puesto que el Scrum Master no es un líder típico, sino que es un un auténtico Servidor neutral, que será el encargado de fomentar e instruir sobre los principios ágiles de Scrum.

> Un buen Scrum Master es aquel capaz de trabajar a la par del equipo, sintiéndose parte de él y siendo capaz de colocarse al servicio de éste, y no, al mando del mismo.

Funciones y responsabilidades

- Garantizar la correcta aplicación de Scrum. Esto incluye, desde la correcta trasmición de sus principios a las altas gerencias, hasta la prevención de la inversión roles (es decir, guardar especial cuidado en que el dueño de producto no actúe en nombre del Scrum Team y viceversa, o que la audencia se inmiscuya en tareas que no le son propicias)
- Resolver los conflictos que entorpezcan el progreso del proyecto.
- Incentivar y motivar al Scrum Team, creando un clima de trabajo colaborativo, fomentar la auto-gestión del equipo e impedir la intervensión de terceros en la gestión del equipo.

Aptitudes que debe tener un Scrum Master

- Excelentes conocimientos de Scrum : Debe contar con amplios conocimientos de los valores y principios del agilismo, así como de las pautas organizativas y prácticas de Scrum
- Amplia vocación de servicio : Lisa llanamente, hablamos de tener vocación de servicio evitando la ambición de poder. El Scrum Master DEBE ser servicial al Scrum Team. Si solo se pretende "llevar el mando", liderar y sobresalir, no será una buena idea convertirse en Scrum Master.
- Tendencia altruista
- Amplia capacidad para la resolución de problemas: La facilidad para resolver los impedimentos detectados será una característica clave de la cual, si se carece de ella, no habrá éxito posible.
- Analítico y observador : El poder de observación será fundamental a la hora de detectar los impedimentos que el Scrum Team deba enfrentar.

- Saber incentivar y motivar
- Capacidad docente e instructiva : ser didáctico y contar con una alta capacidad para transmitir los conocimientos de Scrum a toda la organización y principalmente, a los miembros del Scrum Team y Dueño de Producto
- Buen carisma para las negociaciones: Ser carismático y contar con una marcada tendencia negociadora, abrirá las puertas a que los cambios organizacionales de lo tradicional al agilismo, se conviertan en un hecho que no provoque "roces".

> "El ScrumMaster puede ser un miembro del Equipo; por ejemplo, un desarrollador realizando tareas del Sprint. Sin embargo, esto conduce frecuentemente a conflictos cuando el ScrumMaster tiene que elegir entre eliminar obstáculos o realizar las tareas. El ScrumMaster nunca debe ser el Propietario del Producto." (Scrum Guide, 2008, pág.7)

Actitudes que un buen Scrum Master debe evitar indefectiblemente

- Entrometerse en la gestión del equipo sin dejarlos definir su propio proceso de trabajo: recordemos que el Scrum Team debe ser un equipo auto-gestionado. El Scrum Master NO PUEDE involucrarse en las decisiones organizativas del trabajo del equipo, pues no le corresponde.
- Asignar tareas a uno o más miembros del equipo.
- Colocarse en actitud de autoridad frente al equipo: el Scrum Master no tiene "super-poderes". Debe ser servil y no intentar "dominar". Debe ser "parte del equipo". Una actitud autoritaria, podría provocar el quiebre del equipo, obteniendo resultados poco deseables.

- Negociar con el Dueño de Producto la calidad del producto a desarrollar: el Scrum Master debe tener muy en claro, que solo el cliente tiene autoridad para decidir que otorga valor al negocio y que no.
- Delegar la resolución de un conflicto a un miembro del equipo o a un tercero: el Scrum Master tiene la obligación de tomar en sus manos la resolución de un problema. Cuando ésta, no esté a su alcance, deberá "golpear las puertas" que sean necesarias, y ser él, quien realice un seguimiento a fin de asegurarse que el problema ha sido resuelto.

> **El peor error que un Scrum Master puede cometer,** es delegar la resolución de un conflicto a un miembro del equipo, o intentar colocarse en posición de autoridad frente al Scrum Team.

El Equipo de Desarrollo (Scrum Team o Equipo Scrum)

El Scrum Team (o simplemente "equipo"), es el equipo de desarrolladores multidisciplinario, integrado por programadores, diseñadores, arquitectos, testers y demás, que en forma auto-organizada, será los encargados de desarrollar el producto.

Funciones y responsabilidades

- Convertir el Backlog de Producto, en software potencialmente entregable.
- Aptitudes que deben tener los integrantes de un Scrum Team:
- Ser profesionales expertos o avanzados en su disciplina

- Tener "vocación" (la buena predisposición no alcanza) para trabajar en equipo
- Capacidad de auto-gestión

Un buen Scrum Team, actúa como un verdadero equipo

Difícilmente, alguien se anime a "revelar" con exactitud, el perfil personal que un miembro del equipo de desarrollo, debe tener para formar parte de un Scrum Team, pues muchos, podrían aprovechar esta "revelación" para mentir en una entrevista de trabajo.

Sin embargo, cada vez que inicio un entrenamiento con un nuevo Scrum Team, le digo a mi equipo: no olviden que pueden aprovechar esta información, para mejorar aquellos aspectos de su personalidad, que los harán sentirse a gusto con ustedes mismos y como parte de un verdadero equipo.

Las actitudes personales con las cuales, los miembros del Scrum Team deben contar y aquellas, que deben ser evitadas, se describen a continuación:

- Ser solidario y colaborativo: si quieres formar parte de un verdadero equipo, debes ser consciente de que tus compañeros, pueden necesitar tu ayuda profesional. Cuando un compañero de equipo no sepa algo que tú sí sepas, debes ser solidario con tu conocimiento y colaborar con las tareas de tu compañero.
- Ser motivador, en vez de pretender sobresalir: No es lo mismo decir "yo puedo resolver eso" que "me gustaría que intentáramos resolverlo juntos". Que un compañero de trabajo sienta que lo apoyas, es motivarlo. Pero si te pones en el papel de "sabelotodo", solo estarás

adoptando una postura odiosa. En un equipo, no debes querer sobresalir: debes sentirte a gusto, logrando objetivos entre todos.

- Evitar la competencia: ¿cuál es el objetivo de querer ser "mejor" que tu compañero? Intentando ser "mejor que", malgastas tus energías en competencias sin sentido. Puedes aprovechar toda esa energía, en aprender de tus compañeros aquello que ignoras, y enseñarles aquello en lo que tu eres bueno. Notarás la gran diferencia, al finalizar cada sprint, y ver que el resultado de lo entregado, es lo que verdaderamente sobresale.

> Aprende a disfrutar de los logros de tu equipo y verás como el orgullo es mayor, que cuando solo intentas sobresalir en algo.

Artefactos y Herramientas

Scrum, propone tres herramientas o "artefactos" para mantener organizados nuestros proyectos. Estos artefactos, ayudan a planificar y revisar cada uno de los Sprints, aportando medios ineludibles para efectuar cada una de las ceremonias que veremos más adelante. Ahora, nos concentraremos principalmente, en el backlog de producto, el backlog de Sprint y el Scrum Taskboard, para luego hablar brevemente sobre los diagramas de Burndown.

Backlog de Producto

El Backlog de Producto es un listado dinámico y públicamente visible para todos los involucrados en el proyecto.

> "El Propietario del Producto es responsable del Product Backlog, de su contenido, disponibilidad y priorización." (Scrum Guide, 2008, pág.18)

En él, el Dueño de Producto, mantiene una lista actualizada de requerimientos funcionales para el software. Esta lista, representa «qué es lo que se pretende» pero sin mencionar «cómo hacerlo», ya que de esto último, se encargará el equipo.

> El Backlog de Producto, es creado y modificado únicamente por el Dueño de Producto. Durante la ceremonia de planificación, el Scrum Team obtendrá los items del producto, que deberá desarrollar durante el Sprint y de él, partirán para generar el Backlog de Sprint.

Formato del Backlog de Producto

El Backlog de producto, es una lista de items que representan los requerimientos funcionales esperados para el software.

Para cada uno de estos ítem, será necesario especificar:

- El grado de prioridad
- Esfuerzo que demanda
- Granulidad
- Criterios de aceptación

Priorización de los ítems del Backlog de Producto

Los items del backlog de producto, deben guardar un orden de prioridad, cuya base se apoye en:

- Beneficios de implementar una funcionalidad

- Pérdida o costo que demande posponer la implementación de una funcionalidad
- Riesgos de implementarla
- Coherencia con los intereses del negocio
- Valor diferencial con respecto a productos de la competencia

Estimación de esfuerzo

A diferencia de las metodologías tradicionales, Scrum, propone la estimación de esfuerzo y complejidad que demanda el desarrollo de las funcionalidades, solo para aquellas cuyo orden sea prioritario.

Estas estimaciones, no se efectúan sobre items poco prioritarios ni tampoco sobre aquellos donde exista un alto grado de incertidumbre.

De esta manera, se evita la pérdida de tiempo en estimaciones irrelevantes, postergándolas para el momento en el cual realmente sea necesario comenzar a desarrollarlas.

> "El equipo da al Dueño de Producto las estimaciones del esfuerzo requerido para cada elemento de la Pila de Producto. Además, el Dueño de Producto es responsable de asignar una estimación del valor de negocio a cada elemento individual. Esto es normalmente una práctica desconocida para el Dueño de Producto. Por esa razón, a veces el ScrumMaster puede enseñar al Dueño de Producto a hacerlo." (The Scrum Primer, 2009, pág. 8)

Granulidad de los ítems

Los items del Backlog de Producto no necesariamente deben tener una granulidad pareja. Es posible encontrar ítems tales como "es necesario contar con un módulo de control de stock y logística" o uno tan pequeño como "Modificar el color de fondo de los mensajes de error del sistema, de negro a rojo".

Ítems de tan baja granulidad, suelen agruparse en un formato denominado «Historias de Usuario» mientras que los de alta granulidad, suelen llamarse «temas o epics».

> "Cuando los elementos del Product Backlog han sido preparados con este nivel de granularidad, los que están en la parte superior del mismo (los de mayor prioridad, y mayor valor) se descomponen para que quepan en un Sprint." (Scrum Guide, 2008, pág.19)

Una historia de usuario es aquella que puede escribirse con la siguiente frase:

```
Como [un usuario], puedo [acción/funcionalidad] para [beneficio]
```

Por ejemplo: *Como administrador del sistema, puedo agregar productos al catálogo para ser visualizados por los clientes*.

Muchas veces, puede resultar redundante o hasta incluso carecer de sentido, indicar el beneficio. Por ello, es frecuente describir las historias de usuario, sin incorporar este trecer elemento: Como administrador del sistema, puedo agregar productos al catálogo.

Vale aclarar, que es frecuente encontrar términos como "quiero" o "necesito" en reemplazo de "puedo" cuando se

describen las historias de usuario:

Prioridad	Historia de Usuario	Valor	Esfuerzo estimado
1	como administrador del sistema necesito agregar productos al catálogo	10	
2	como usuario del sitio web quiero recorrer el catálogo de productos	10	
3	como cliente de la empresa quiero agregar productos a un pedido	10	
4	como cliente necesito enviar el pedido una vez que haya agregado todos los productos deseados	10	
5	como administrador de pedidos necesito ver la lista de pedidos efectuados por los clientes	7	
6	como administrador de pedidos necesito ver el detalle de los productos solicitados por el cliente en cada uno de los pedidos	10	
7	como administrador de pedidos necesito modificar el estado de cada pedido	3	
8	como cliente quiero poder visualizar el estado de mis pedidos	8	
9	como administrador del sistema necesito clasificar los productos por categorías	3	
10	como administrador del sistema necesito eliminar productos del catálogo	3	
11	como administrador del sistema necesito indicar cuáles productos y cuáles no, son visibles en el catálogo de productos que ven los usuarios	3	
12	como administrador del sistema necesito modificar productos del catálogo	3	
13	como cliente quiero poder modificar la cantidad de ítems de los productos de mi pedido	5	
14	como cliente de la empresa quiero poder cancelar mi pedido que aún no ha sido despachado	5	
15	como empleado de depósito necesito ver el detalle de los pedidos que aún no han sido enviados a empaque	1	
16	como empleado de depósito necesito indicar que un pedido ya ha sido enviado a empaque	5	
17	como cliente de la empresa necesito poder recuperar mi contraseña cuando la olvido	10	

Imagen 1: Backlog de Producto

> "El Product Backlog nunca está completo. La primera versión para el desarrollo, tan sólo establece los requisitos inicialmente conocidos, y que son entendidos mejor. El Product Backlog evoluciona a medida que el producto y el entorno en el que se utilizará evoluciona. El Product Backlog es dinámico, ya que cambia constantemente para identificar qué necesita el producto para ser adecuado, competitivo y útil." (Scrum Guide, 2008, pág.18)

Criterios de Aceptación

Es recomendable que cada ítem del Backlog de Producto,

especifique cuales son los criterios de aceptación (o test de aceptación que debe superar), para considerar cumplido el requisito.

Los criterios de aceptación, entonces, no son más que "pautas" o pequeñas "reglas" que una historia de usuario debe respetar para considerarla cumplida. Por ejemplo, para la historia de usuario **«Como administrador del sistema necesito agregar productos al catálogo»** los **criterios de aceptación**, podrían ser:

- Cada producto debe contener:
    - código de producto (opcional),
    - descripción de hasta 500 caracteres (opcional)
    - precio (obligatorio)
    - stock inicial (opcional)
    - un nombre (obligatorio),
    - una foto (opcional)
- No pueden existir dos productos con el mismo nombre de producto o código de producto
- El nombre del producto jamás puede estar vacío o tener menos de 3 caracteres
- Cuando no se asigne un stock inicial al producto, éste debe asignarse automáticamente en cero (sin stock)

> "Las pruebas de aceptación se utilizan a menudo como un atributo más del Product Backlog. A menudo pueden sustituir a descripciones de texto más detalladas, que contengan una descripción comprobable, de lo que el elemento del Product

> Backlog debe hacer cuando esté completado." (Scrum Guide, 2008, pág.19)

Backlog de Sprint

El Backlog de Sprint es una **lista reducida de ítems del Backlog de Producto**, que han sido negociados entre el Dueño de Producto y el Scrum Team durante la planificación del Sprint.

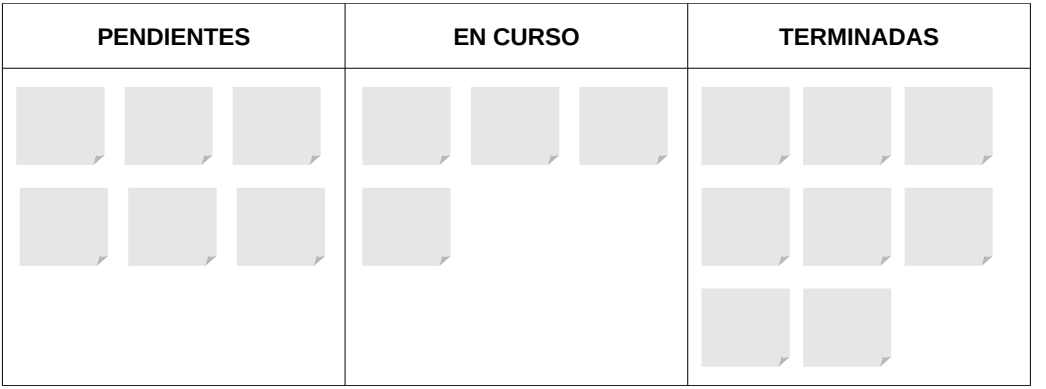
Esta lista, **se genera al comienzo de cada Sprint** y representa aquellas características que el equipo se compromete a desarrollar durante la iteración actual.

Los ítems incluidos en el Backlog de Sprint **se dividen en tareas** las cuales generalmente, **no demanden una duración superior a un día** de trabajo del miembro del equipo que se haya asignado dicha tarea.

Se actualiza diariamente por el equipo y de manera permanente, muestra:

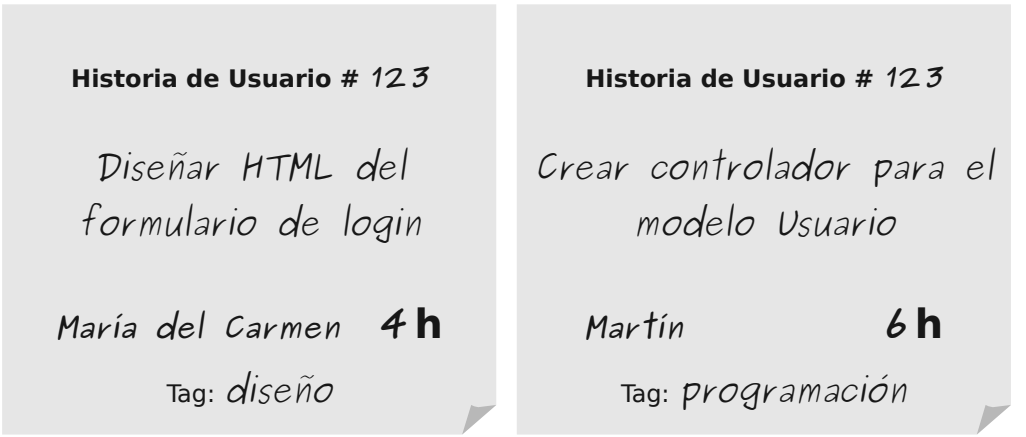
- Las tareas pendientes, en curso y terminadas.
- La estimación del esfuerzo pendiente de cada tarea sin concluir.
- El nombre del miembro del equipo que se ha asignado dicha tarea.

Generalmente, el Backlog de Sprint, se visualiza mediante un tablero físico, montado en alguna de las paredes de la sala de desarrollo.



*Tablero físico (**Scrum Taskboard**)*

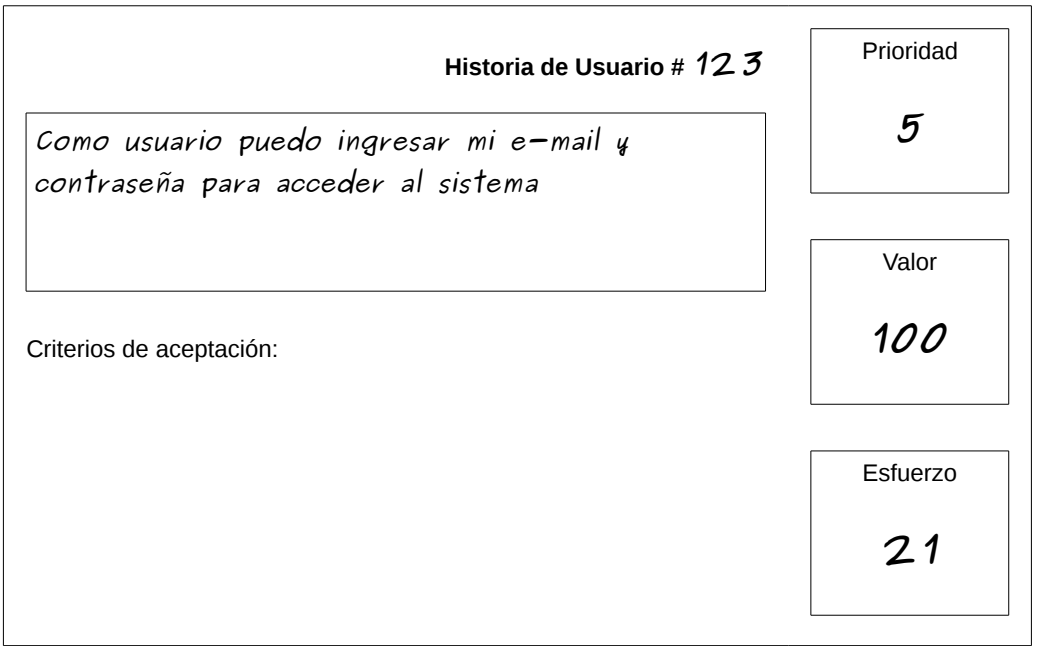
Es muy frecuente, a la vez de ser una práctica recomendada, que cada tarea sea a la vez, "etiquetada", diferenciando, por ejemplo, cuando representa un bug, una tarea de diseño, un test, etc.



Dividiendo Historias de Usuario en Tareas

La estrategia consiste en desmembrar el item a la mínima expresión posible, encuadrada en un mismo tipo de actividad.

El desmembramiento debe hacerse "de lo general a lo particular, y de lo particular al detalle".



Ficha típica de Historia de Usuario

Análisis General:

Es aquel que responde a la pregunta ¿qué es?

- un sistema de validación de usuarios registrados

Análisis Particular:

Es el que responde a la pregunta ¿cómo hacerlo?

- Arquitectura MVC (requiere hacer el modelo, la lógica y la GUI de la vista y el controlador)

Análisis detallado: Es el que responde a la pregunta general ¿qué tareas se necesitan hacer para lograrlo?

> Los detalles, son aquellas restricciones que deberán considerarse para todo lo anterior. Por ejemplo, la creación del modelo, repercutirá en la base de datos. Por lo cual, tras crear los nuevos modelos, habrá que correr el ORM para que modifique las tablas.
> Otro detalle a considerar, es el tiempo que demanda cada tarea. Por ejemplo, correr un ORM lleva solo algunos minutos, pues no puede ser considerado una única tarea. Entonces, puede "sumarse como detalle" a la tarea "crear modelos". De manera contraria, documentar en el manual del usuario, llevará todo un día de trabajo. Por lo cual, debe asignarse a una única tarea.

- Crear el modelo Usuario y correr el ORM para modificar las tablas
    - Tag: programación
    - Esfuerzo: 2 h

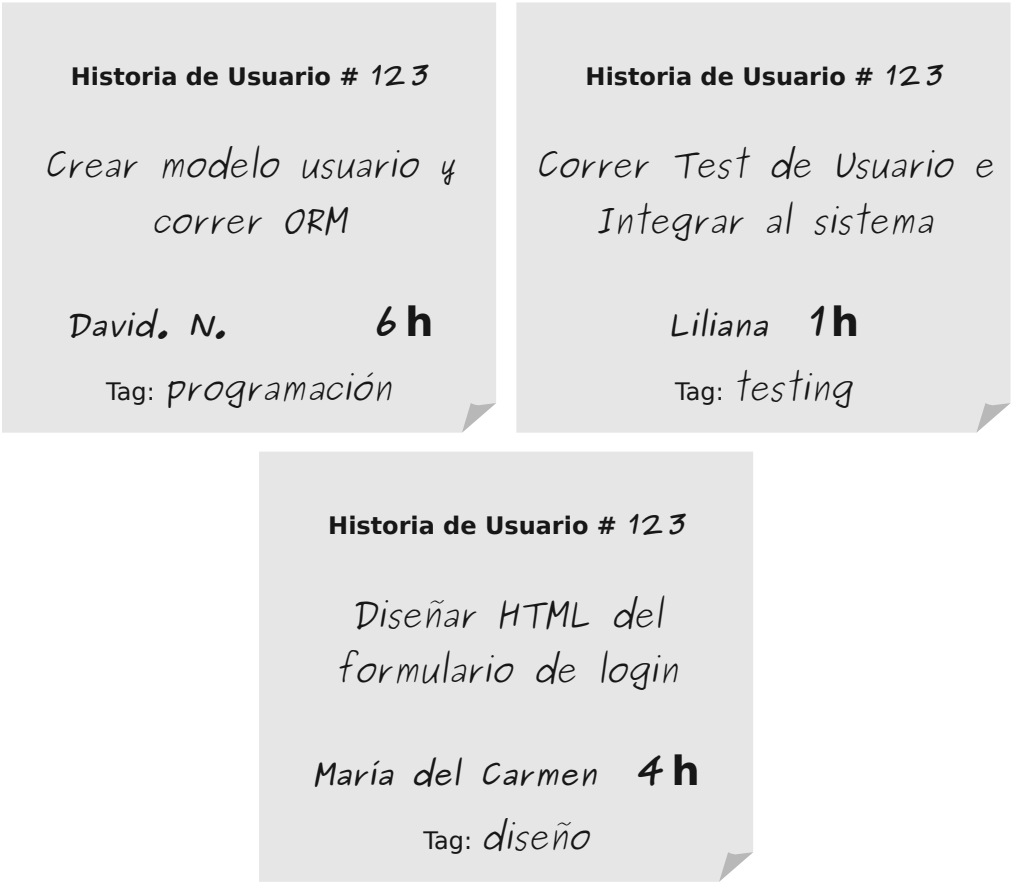
- Diseñar un formulario HTML para insertar usuario y contraseña
    - Tag: diseño
    - Esfuerzo: 4 h

- Desarrollar la lógica de la vista del formulario de logueo
    - Tag: programación
    - Esfuerzo: 4 h

- Crear el controlador para el modelo
    - Tag: programación
    - Esfuerzo: 6 h

- Correr los test e integrar
    - Tag: testing
    - Esfuerzo: 1 h

Finalmente, dichas tareas se plasmarán en diferentes post-it (una tarea en cada uno). Los miembros del equipo decidirán qué tareas se asignará cada uno y se colocarán los post-it en el tablero:



Incremento de Funcionalidad

Al finalizar cada Sprint, el equipo hará entrega de un **incremento de funcionalidad** para el sistema. Este incremento, debe lograr asemejarse a "un software funcionando" pudiendo ser implementado en un ambiente de producción de manera 100% operativa.

Ceremonias en Scrum

En Scrum, es frecuente oir hablar de "ceremonias" cuando nos referimos a las cuatro reuniones que se realizan de forma iterativa en cada Sprint.

Estas reuniones (o ceremonias) son:

1. Planificación del Sprint
2. Reunión diaria
3. Revisión
4. Retrospectiva

Ceremonia de Planificación del Sprint

La planificación es lo primero que debe hacerse al comienzo de cada Sprint. Durante esta ceremonia, participan el Dueño de Producto, el Scrum Master y el Scrum Team.

El objetivo de esta ceremonia, es que el Dueño de Producto pueda presentar al equipo, las historias de usuario prioritarias, comprendidas en el Backlog de producto; que el equipo comprenda el alcance de las mismas mediante preguntas; y que ambos negocien cuáles pueden ser desarrolladas en el Sprint que se está planificando.

Una vez definido el alcance del sprint, es cuando el equipo divide cada historia de usuario, en tareas, las cuales serán necesarias para desarrollar la funcionalidad descrita en la historia.

Estas tareas, tendrán un esfuerzo de desarrollo estimado (en horas que se deducen de la estimación de esfuerzo realizada para la Historia de Usuario, mediante técnicas como Planing Poker, Columnas o T-Shirt Sizing, que veremos más adelnate), tras lo cual, serán pasadas al backlog de Sprint y de allí se visualizarán en el tablero una vez que cada miembro se haya asignado aquellas que considere puede realizar.

La planificación puede demandar solo unas horas, o toda una jornada laboral completa.

Objetivos	• Que el Scrum Team conozca los ítems prioritarios del Backlog de Producto • Que el Dueño de Producto negocio con el equipo los ítems a desarrollar • Que el equipo defina las tareas necesarias para cumplir con la entrega
Participantes	• Dueño de Producto • Equipo • Scrum Master
Momento	• Al comienzo del Sprint
Duración	• Fija, entre 2 y 8 horas
Artefactos involucrados	• Backlog de producto: para negociar los items a desarrollar y comprender su importancia • Backlog de Sprint: se define durante la planificación
Dinámica	• Presentar items El Dueño de Producto presenta los ítems • Hacer preguntas

	El Scrum Team realiza las preguntas que considere necesarias al Dueño de Producto a fin de entender los ítems presentados • Estimar esfuerzo El Scrum Team realiza las estimaciones de esfuerzo y complejidad necesarias para desarrollar los ítems propuestos por el Dueño de Producto. Estas estimaciones pueden realizarse a través de técnicas como Planning Pocker, columnas y/o T-Shirt Sizing. • Definir alcance Basados en la estimación efectuada, el Scrum Team, define el alcance del Sprint • Negociar El Dueño de Producto, acepta, rechaza o solicita cambios al alcance definido por el Scrum Team • Definir tareas El Scrum Team, define las tareas para cada ítem seleccionado, estimando la duración de horas de cada una • Armar tablero Se checkea la lista y se arma el tablero

Reunión diaria

Las reuniones diarias para Scrum, son "conversaciones" de no más de 5-15 minutos, que el Scrum Master tendrá al comienzo de cada día, con cada miembro del equipo.

En esta conversación, el Scrum Master deberá ponerse al día de lo que cada miembro ha desarrollado (en la jornada previa), lo que hará en la fecha actual, pero por sobre todo, conocer cuáles impedimentos estén surgiendo, a fin de resolverlos y

que el Scrum Team pueda continuar sus labores, sin preocupaciones.

Objetivos	• Que los miembros del equipo sincronicen sus tareas • Reportar al Scrum Master los avances e impedimentos
Participantes	• Equipo • Scrum Master
Momento	• Una vez por día en horario fijo (generalmente, al comienzo de la jornada laboral)
Duración	• Fija, no mayor a 15 minutos
Artefactos involucrados	• Backlog de Sprint: actualización del tablero
Dinámica	• Entrevista con el Scrum Master El Scrum Master se reúne con los miembros del equipo y pregunta por lo que se ha heho, lo que queda pendiente para hoy y si existe algún impedimento para resolverlo luego de la reunión • Actualización del tablero Se actualiza y checkea el tablero

Ceremonia de Revisión

Durante la ceremonia de revisión en Scrum, el equipo presentará al Dueño de Producto las funcionalidades desarrolladas. Las explicará y hará una demostración de ellas, a fin de que, tanto Dueño de Producto como la eventual audencia, puedan experimentarlas.

El Dueño de Producto podrá sugerir mejoras a las funcionalidades desarrolladas, aprobarlas por completo o eventualmente, rechazarlas si considera que no se ha cumplido el objetivo.

La ceremonia de revisión se lleva a cabo el último día del Sprint, y no tiene una duración fija. En la práctica, se utiliza el tiempo que sea necesario.

Objetivos	• Que los principales interesados experimenten las funcionalidades construidas y eventualmente sugieran alguna mejora • Que el Dueño de Producto acepte o rechace las funcionalidades construidas
Participantes	• Scrum Master • Dueño de Producto • Scrum Team • Audiencia (opcionalmente)
Momento	• Último día del Sprint
Duración	• No menor a 1 hora y no mayor una jornada completa
Artefactos involucrados	• Incremento de funcionalidad
Dinámica	• Prsentación de los items desarrollados El Scrum Team presenta los items desarrollados del Backlog de Producto y procede a mostrar el software en funcionamiento • Evacuación de consultas El Dueño de Producto y principales interesados, realizan al Scrum Team

	consultas referidas al funcionamiento y eventualmente, el dueño de producto, podrá sugerir mejoras o solicitar la implementación de las mismas en el entorno de producción • Actualización del backlog de producto (opcional) Aquellas funcionalidades aún no resueltas (y planificadas), el Dueño de Producto, las reincorpora al Backlog de Producto • Acuerdo de próximo encuentro El Scrum Master, anuncia fecha, horario y lugar del próximo encuentro

Ceremonia de Retrospectiva: la búsqueda de la perfección

No es en vano la frase "en la búsqueda de la perfección". Como última ceremonia del Sprint, Scrum propone efectuar al equipo, una retrospectiva en forma conjunta con el Scrum Master y opcionalmente, el Dueño de Producto.

El objetivo de esta retrospectiva, como su nombre lo indica, es "mirar hacia atrás (en retrospectiva)", realizar un análisis de lo que se ha hecho y sus resultados correspondientes, y decidir que medidas concretas emplear, a fin de mejorar esos resultados.

La retrospectiva en Scrum suele ser vista como una "terapia de aprendizaje", donde la finalidad es "aprender de los aciertos, de los errores y mejorar todo aquello que sea factible".

Objetivos	• Detectar fortalezas del equipo y

	oportunidades de mejora • Acordar acciones concretas de mejoras para el próximo Sprint
Participantes	• Equipo • Scrum Master • Dueño de producto (opcionalmente)
Momento	• Al finalizar la jornada, del último día del Sprint
Duración	• Fija, no mayor a 2 o 3 horas
Artefactos involucrados	• ninguno
Dinámica	• **Identificación de oportunidades de mejora** El Scrum Master identifica aquellos mecanismo o procedimientos que deban mejorarse • **Definición de acciones concretas** El Scrum Master y equipo, definen las acciones concretas que serán llevadas a cabo en el próximo Sprint, a fin de mejorar lo necesario • **Revisión de acciones pasadas** Se revisan las acciones acordadas en retrospectivas anteriores y se decide si serán continuadas o no

Estimando esfuerzos

A diferencia de las técnicas de estimación tradicionales, centradas en el "tiempo" que demanda la realización de actividades inciertas, las metodologías ágiles en general,

proponen realizar estimaciones de "esfuerzo" ya que seguramente, es mucho más fácil y certero indicar "cuánto esfuerzo te demanda mover 100 ladrillos" que "cuánto tiempo te llevará mover 5 bolsas de arena".

Con certeza, uno podría segurar, que pintar un edificio de 14 pisos, le llevará "mucho esfuerzo". Pero muy difícilmente, pueda decir de forma certera, cuanto tiempo tardará. Muy probablemente, el tiempo que se indique no sea el que finalmente resulte.

Estimar el "esfuerzo" es algo que resulta independiente de la cantidad y calidad de los factores del entorno. Pues para cualquier pintor, pintar el edificio de 14 pisos demandará mucho esfuerzo. Incluso aunque los pintores designados a realizar la tarea, sean seis.

Sin embargo, estimar el tiempo, es un factor que además de poco certero, es inherente a los elementos del entorno. Pues es fácil decir que pintar el edificio demandará mucho esfuerzo, pero calcular el tiempo que demande hacerlo, implicarán factores como la cantidad de pintores involucrados, la calidad de la pintura, las herramientas que se utilicen, el estado climatológico, entre decenas de otros factores.

Por todo esto, es que el agilismo se avoca a estimar lo certero, basándose en un principio de honestidad, mediante el cual, los miembros del equipo se compremeten a ser sinceros, teniendo la humildad necesaria y suficiente, requerida para conocer sus propias limitaciones.

Existen varias técnicas para estimar el esfuerzo. Todas las técnicas de estimación, se realizan sobre las Historias de Usuarios y sin excepciones, involucran un juego de

planificación que hace que estimar, sea algo **divertido**.

Aquí hablaremos de 3 técnicas:

- **T-Shirt Sizing**
- Estimación por **Poker** y
- Estimación por **Columnas**

Hablaremos además, de una nueva técnica que nos proponen **Leonardo de Seta** y **Sergio Gianazza** de **IdeasAgiles.com**, la cual consiste en la combinación de **Columnas y Poker**.

Estimando con T-Shirt Sizing

T-Shirt Sizing es una técnica que se basa en la tabla de medidas americana que utilizan las prendas de vestir:

- **XS**: extra small (muy pequeño)
- **S**: small (pequeño)
- **M**: medium (mediano)
- **L**: large (grande)
- **XL**: extra large (muy grande)

Cuanto más pequeña es la medida, menor esfuerzo requiere una Historia de Usuario. Y cuanto más grande, mayor esfuerzo.

Historia de Usuario # 123

Como usuario puedo ingresar mi e-mail y contraseña para acceder al sistema

Criterios de aceptación:

Prioridad: 5

Valor: 100

Esfuerzo: XL

Una historia de usuario estimada con T-Shirt Sizing que requiere de mucho esfuerzo

Historia de Usuario # 123

Como usuario del sitio Web, puedo ver todos los artículos publicados en el catálogo de productos

Criterios de aceptación:

Prioridad: 2

Valor: 75

Esfuerzo: S

Una historia de usuario estimada con T-Shirt Sizing que requiere de poco esfuerzo

¿Cómo se juega?

Jugar a estimar con T-Shirt Sizing, es sumamente sencillo. El juego consiste en:

- Un miembro del equipo lee una Historia de Usuario
- Cada miembro del equipo anota en un papel, el esfuerzo que cree que demanda realizar dicha HU
- Todos al mismo tiempo, giran el papel, haciéndolo visible
    - Si hay consenso, se indica en la ficha de la HU la "medida" estimada
    - De lo contrario, el miembro del equipo que mayor esfuerzo haya estimado, argumenta su estimación
    - A continuación, argumenta el que menor esfuerzo estimó y se vuelve a votar
    - Cuando no se llega a un consenso, puede optarse por elegir la estimación realizada por la mayoría o intervenir el Scrum Master para solventar dicha falta de consenso, como un impedimento, pudiendo optar por dar una nueva argumentación y permitir votar una vez más, o directamente, elegir la estimación que considera oportuna. De todas formas, lo ideal, siempre es encontrar el consenso.

Estimación por Poker

Hacer estimaciones con **Poker**, no solo es una técnica: **¡Es un juego muy divertido!**

El **Scrum Poker** (Planning Poker o Poker a secas), se juega con una **baraja de cartas** numeradas siguiendo la **serie de Fibonacci**, la cual se compone de una serie lógica donde cada

número es la suma de los dos anteriores:

1, 1, 2, 3, 5, 8, 13, 21, 34, …

Serie de Fibonacci donde cada número es la suma de los dos anteriores

A menor número, menor esfuerzo demanda una Historia de Usuario. Y cuanto más elevado es el valor, mayor esfuerzo.

La baraja de Scrum Poker, cuenta además con dos cartas adicionales:

- Una taza de café, que significa "¡No doy más! Hagamos una pausa"
- Un signo de interrogación, que puede significar "No estoy seguro del esfuerzo que demanda" o también "No entendí muy bien los requerimientos".

En Internet, pueden encontrarse un sinfín de variantes de barajas para Scrum Poker, encontrando algunas muy originales y divertidas como la de www.Autentia.com.

Reglas del Juego

Antes de comenzar, deben establecerse las reglas:

- El Scrum Master podrá ser quien actúa de moderador del juego
- Debe decidirse qué tiempo de exposición será asignado a cada participante para explicar su estimación
- También será necesario definir, el tiempo de descanso

previsto cuando un miembro del equipo, arroje la carta "taza de café"

¿Cómo jugar Scrum Poker?

En la planificación del Sprint, una vez que el Dueño de Producto ha presentado un ítem del Backlog de Producto, y ya todos los miembros del Scrum Team han hecho las preguntas necesarias para entender el alcance del mismo, es hora de determinar el esfuerzo y la complejidad que demanda construir la "historia de usuario". Y aquí es donde comienza el juego:

1. Cada miembro del equipo debe tener en su poder un juego de cartas

2. Tras haber discutido el alcance, cada miembro pensará en el esfuerzo que construir dicha funcionalidad demanda, y colocará la carta correspondiente, boca abajo (es decir, sin que el número sea visible), sobre la mesa.

3. Una vez que todos los miembros hayan bajado una carta sobre la mesa, se dan vuelta, haciéndolas visibles.

4. Si todos los miembros han arrojado la misma carta, el esfuerzo ha sido estimado y debe indicarse el mismo, en la ficha de Historia de Usuario. Una vez hecho, se levanta la carta de la mesa y se prosigue a estimar el próximo ítem del Backlog de Producto (repitiendo desde el paso 2)

5. Si existen diferencias en la estimación, el miembro que

mayor esfuerzo haya estimado, deberá exponer sus motivos (muy probablemente, encontró impedimentos que los otros miembros no). Luego, expondrá aquel que menor esfuerzo haya estimado (seguramente, encontró una forma más sencilla de resolverlo que los otros miembros).

6. Hecha la exposición, se vuelve a votar, repitiendo la secuencia desde el punto "2"

Cuando no se llega al consenso tras la tercera vez de jugar una carta para la misma Historia de Usuario, el Scrum Master deberá decidir la estimación final para dicha HU.

El juego finaliza cuando ya no queden más Historias de Usuario para estimar.

Historia de Usuario # 123

Como usuario puedo ingresar mi e-mail y contraseña para acceder al sistema

Criterios de aceptación:

Prioridad: 5

Valor: 100

Esfuerzo: 144

La misma historia de usuario estimada con T-Shirt Sizing que requiere de mucho esfuerzo, pero esta vez, con valores de Scrum Poker

Historia de Usuario # 123

Como usuario del sitio Web, puedo ver todos los artículos publicados en el catálogo de productos

Criterios de aceptación:

Prioridad: 2

Valor: 75

Esfuerzo: 13

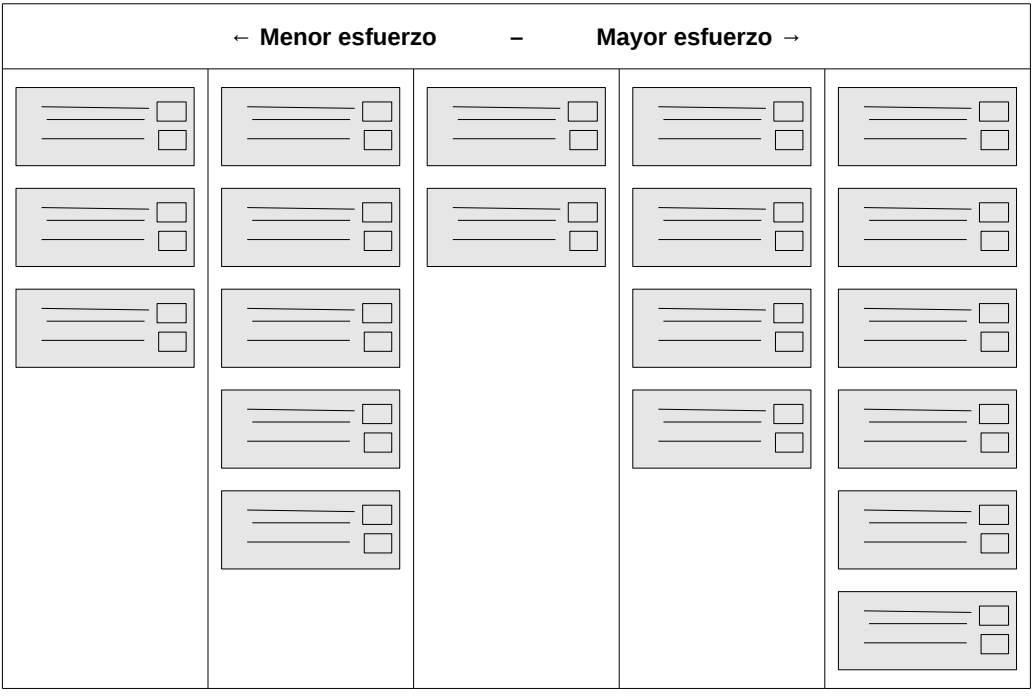
La misma historia de usuario estimada con T-Shirt Sizing que requiere de poco

esfuerzo, pero esta vez, con valores de Scrum Poker

Estimando por Columnas

La estimación por columnas es una técnica pensada para estimar grandes cantidades de Historias de Usuario en bloque. El objetivo es agrupar las Historias de Usuario en columnas con el mismo grado de granulidad.

Cuanto más a la izquierda está la columna, menor esfuerzo demandan las Historias de usuario de esa columna. Y cuanto más a la derecha, mayor esfuerzo:



En la estimación por columnas, los miembros del equipo debn estar sentados alrededor de una mesa. El juego se inicia formando una pila con las Historias de Usuario, a modo de baraja.

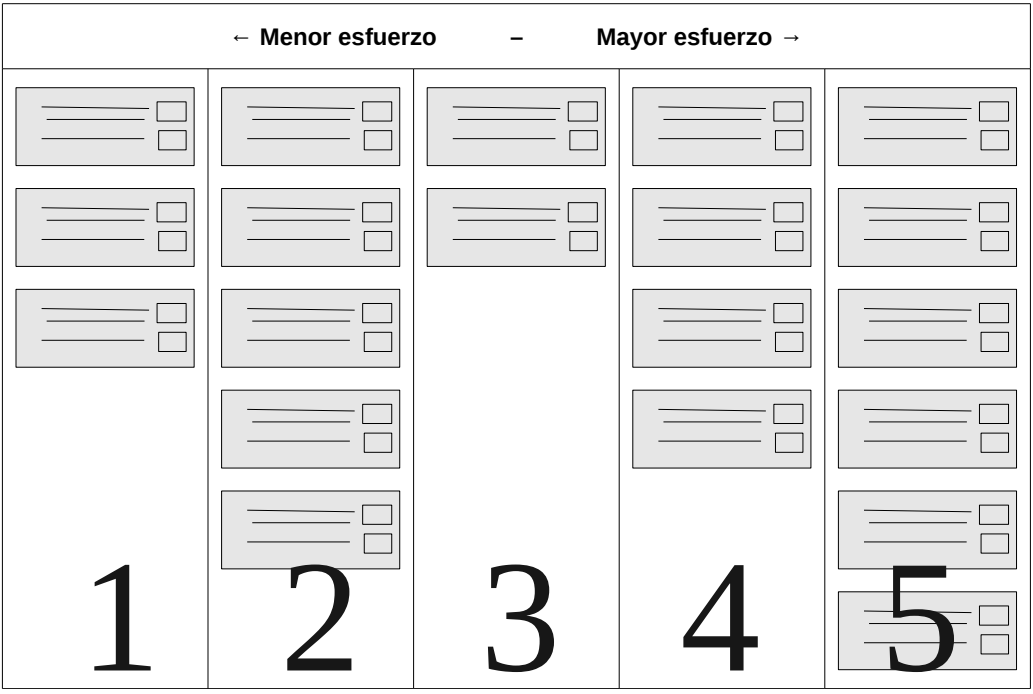
1. Uno de los miembros, toma la baraja de Historias de Usuario y da inicio a la primera ronda del juego, extrayendo una Historia de Usuario del mazo. La lee en voz alta, y la ubica en la mesa, formando una columna imaginaria. A continuación, pasa la baraja al jugador ubicado a su izquierda.

2. El segundo en jugar, vuelve a extraer una Historia de Usuario de la pila, la lee en voz alta y la ubica en la mesa, teniendo en cuenta que:
    - Si la Historia de Usuario demanda menor esfuerzo que la que se encuentra en la mesa, deberá colacarla a la izquierda de ésta;
    - Si demanda el mismo esfuerzo, la ubicará debajo;
    - Y si demanda mayor esfuerzo, la colacará a la derecha;
    - A continuación, pasa la pila al jugador ubicado a su izquierda.

3. A partir del tercer jugador, el juego continúa en ronda de la siguiente forma:
    4. El juegador que posee la pila de Historias de usuario en la mano, debe elegir entre dos opciones:
        5. Mover de columna una Historia de Usuario y pasar la pila al siguiente jugador

        6. Extraer una nueva Historia de Usuario, leerla en voz alta y colocarla en la mesa teniendo en cuenta lo explicado en el punto 2.

7. Cuando ya no quedan más Historias de Usuario en la pila, se asigna a todas las Historias de una misma columna, el mismo grado de esfuerzo (dependiendo de la cantidad de columnas, puede indicarse como esfuerzo, el número de orden de las columnas, de izquierda a derecha) y finaliza el juego



Estimación por Columnas y Poker

Leonardo de Seta y Sergio Gianazza, integrantes del equipo de IdeasAgiles.com, nos proponen una idea muy divertida y sumamente productiva, para estimar grandes cantidades de Historias de Usuario en bloque.

La técnica propuesta por Sergio y Leonardo consiste en realizar la estimación por Columnas utilizando un máximo de 5 columnas y a continuación, realizar una nueva estimación por Poker para cada columna, de la siguiente manera:

1. Se comienza la estimación por Poker desde la fila del medio

2. Como baraja, proponen contar con las siguientes cartas:
    - ½, 1, 2, 3, 5, 8, 13, 21, ∞ (infinito), taza de café, ?, ¡!.
    - La carta infinito, indicará que el esfuerzo es mayor a 21 mientras que la de los signos de admiración, indicará que la Historia de Usuario es demasiado grande para estimar y debe ser dividida.

3. Se elige cualquiera de las Historias de Usuario de esta fila y se estima el esfuerzo mediante la técnica de Scrum Poker.
    - Para estimar la columna del medio, deben exceptuarse las cartas ½ y 1 (ya que al haber dos columnas previas, no podrán estimarse con menos de ½ y 1 cada una) y 13 y 21 (las cuales se reservan para las dos últimas columnas)

4. El valor estimado, se asigna a todas las Historias de Usuario de la columna.

5. A continuación, se repite el procedimiento con las 4 columnas restantes, teniendo en cuenta que si la estimación de la columna central, fue realizada en 3, las Historias de Usuario de las dos primeras columnas, serán estimadas automáticamente, en 1 y 2 respectivamente.

6. El juego finaliza cuando la última columna ha sido estimada.

Scrum Kit

En http://agilecoaching.eugeniabahit.com puedes descargar un **completo Kit para Scrum**, el cual incluye:

1. Mazo de cartas para Scrum Poker;
2. Fichas para Historias de Usuario;
3. Post-it para tareas del Scrum Taskboard;
4. Diagrama de Burndown para Sprints de 2 y 3 semanas.

Introducción a la Programación eXtrema

eXtreme Programming (programación extrema) también llamado **XP**, es una metodología que tiene su origen en 1996, de la mano de **Kent Beck**, **Ward Cunningham** y **Ron Jeffries**.

A diferencia de Scrum, XP propone solo un **conjunto de prácticas técnicas**, que **aplicadas de manera simultánea**, pretenden enfatizar los efectos positivos de en un proyecto de desarrollo de Software.

Bases de la programación eXtrema

eXtreme Programming se apoya en cinco valores, los cuales enfatizan la esencia colaborativa del equipo. Estos **valores** son:

Comunicación

En XP, todo es trabajado en equipo: desde el relevamiento y análisis hasta el código fuente desarrollado. **Todo se conversa cara a cara**, procurando hallar soluciones en conjunto a los problemas que puedan surgir.

Simplicidad

Se pretende **desarrollar solo lo necesario** y no perder tiempo en detalles que no sean requeridos en el momento. En este aspecto, se asemeja a otra metodología ágil, denominada Kanban, en la cual, un proceso "anterior" solo produce lo que el proceso posterior demanda.

Retroalimentación

El objetivo de eXtreme Programming es entregar lo necesario al cliente, en el menor tiempo posible. A cambio, demanda al cliente, un ***feedback* continuo** -retroalimentación-, a fin de conocer sus requerimientos e implementar los cambios tan pronto como sea posible.

Respeto

El equipo respeta la idoneidad del cliente como tal (sólo éste, es quien conoce el valor para el negocio) y **el cliente**, a la vez, **respeta la idoneidad del equipo** (confiando en ellos profesionalmente para definir y decidir el "cómo" se llevará a cabo el desarrollo de lo requerido).

Coraje

Se dice que en XP un equipo **debe tener el valor para decir la verdad** sobre el avance del proyecto y las estimaciones del mismo, planificando el éxito en vez de buscar excusas sobre los errores.

Prácticas técnicas

eXtreme Programming propone **12 prácticas técnicas**, simples, de fácil compresión y que aplicadas en forma conjunta, garantizan un mejor resultado del rpoyecto. Estas doce prácticas, se describen a continuación.

PRÁCTICA #1: CLIENTE IN-SITU (ON-SITE CUSTOMER)

En XP se requiere que el cliente esté dispuesto a participar activamente del proyecto, contando con la disponibilidad suficiente y necesaria, para interactuar con el equipo en todas las fases del proyecto.

> *Siempre es recomendable contar con más de una persona (por parte del cliente), asignada a trabajar de forma permanente con el equipo.*

La comunicación cara a cara con el cliente es fundamental, ya que a partir de ésta, el equipo avanzará más rápidamente en el proyecto, puesto que:

- Evacuará todas sus dudas sobre el proyecto, con el cliente y en el momento que éstas surjan.
- Se establecerán las prioridades a desarrollar, en tiempo real, así como la resolución de conflictos.

Las personas asignadas por el cliente, deben ser:

- Conocedores **expertos** de lo que se pretende producir.
- En lo posible, se pretende que sean **usuarios reales** del software.
- Personas que cuenten con la información suficiente sobre la aplicación y los objetivos del negocio, a fin de poder actuar con **autoridad en la toma de decisiones**.

PRÁCTICA #2: SEMANA DE 40 HORAS (40 HOUR WEEK)

eXtreme Programming asegura la calidad del equipo, considerando que éste, **no debe asumir responsabilidades que le demanden mayor esfuerzo del que humanamente se puede disponer.**

Esto marca una diferencia radical con otras metodologías, sobre todo, con aquellas que siguen una línea tradicional donde las estimaciones deficientes obligan al equipo de desarrollo, a sobre-esfuerzos significativos.

Los docentes del curso **Metodologías Agiles para Gestión de Proyectos de Desarrollo de Software** de la **Universidad Tecnológica Nacional** de Buenos Aires, **Ulises Martins**, **Hernán Ricchio** y **Thomas Wallet**, se refieren a ésto, en los apuntes de la cátedra, diciendo que:

> *"Los proyectos de desarrollo tradicionales suelen necesitar horas extras y esfuerzos heroicos para entregar el software en la fecha prometida. El resultado suele ser un equipo agotado y menos productivo, lo cual genera una fuerte baja en la calidad del software producido, ya sea en el corto, mediano o largo plazo. Horas extras y esfuerzos heroicos son señales de problemas mayores, como por ejemplo compromisos por encima de las posibilidades, estimaciones pobres, falta de recursos[...] Varias organizaciones aceptan convivir con este problema continuamente." (© 2011 Ulises Martins, Hernán Ricchio, Thomas Wallet)*

Un equipo "descansado", sin esfuerzos desmedidos, logra un mejor resultado.

PRÁCTICA #3: METÁFORA (METAPHOR)

A fin de evitar los problemas de comunicación que suelen surgir en la práctica, entre técnicos y usuarios, eXtreme Programming propone el uso de metáforas, intentando hallar un punto de referencia que permita representar un concepto técnico con una situación en común con la vida cotideana y real.

Para que una metáfora, cumpla este objetivo, es fundamental:

- Hacer un **paralelismo entre una funcionalidad del sistema y la vida real**
- Si ese paralelismo (metáfora) no se entiende inmediatamente, se deberá buscar otra metáfora (una metáfora solo sirve cuando es comprendida de forma inmediata)

Muchas veces, puede ser necesario emplear más de una metáfora para explicar una misma característica del sistema. Incluso, en algunas oportunidades, pueden no encontrarse metáforas que describan con precisión toda la característica que se intenta explicar. Son momentos, en los que se requiere de un gran ingenio y creatividad.

> *Una **metáfora** es la forma de ser **didácticos** para explicar a nuestro receptor, un concepto técnico y que éste, lo comprenda con facilidad.*

PRÁCTICA #4: DISEÑO SIMPLE (SIMPLE DESIGN)

Esta práctica, deriva de un famoso principio técnico del desarrollo de software: **KISS**. Cuyas siglas derivan del inglés **"Keep it simple, stupid!"** *-¡Manténlo simple, estúpido!-*.

Dado que frecuentemente puede resultar "chocante" el término "stupid", a lo largo de los años, se han ido produciendo variantes, como "silly" -tonto- e incluso, muchos de nosotros, a veces, preferimos reemplezarlo por un nombre propio, como si

estuviésemes dirigiéndo la frase a una persona en particular: *Keep it Simple, Susan!*

Básicamente, el principio KISS, se trata de mantener un diseño sencillo, estandarizado, de fácil compresión y refactorización. Puede resumirse en **"hacer lo mínimo indispensable, tan legible como sea posible"**.

Puedes ver más sobre esta técnica, leyendo el artículo en Wikipedia.

PRÁCTICA #5: REFACTORIZACIÓN (REFACTORING)

La refactorización (o refactoring) es una técnica que consiste en **modificar el código fuente de un software sin afectar a su comportamiento externo**. Puedes leer más sobre refactorización en la Wikipedia.

> *Más adelante, veremos las principales técnicas de Refactoring, en detalle, utilizando como referencia, el manual de Refactoring[8] (en inglés) del sitio Web SourceMaking.com*

Dado que eXtreme Programming propone "desarrollar lo mínimanete indispensable", en el desarrollo de cada nueva funcionalidad, puede (y debe) ser necesario, realizar un

8 http://sourcemaking.com/refactoring

refactoring del código, a fin de lograr una mayor cohesión de éste, impidiendo redundancias.

Por ello, la refactorización es la principal técnica propuesta por XP, para evolucionar el código de la aplicación con facilidad en cada iteración.

PRÁCTICA #6: PROGRAMACIÓN DE A PARES (PAIR PROGRAMMING)

Otra de las prácticas técnicas fundamentales de eXtreme Programming, es la programación de a pares.

Se estila (aunque no de forma constante) programar en parejas de dos desarrolladores, los cuales podrán ir intercambiando su rol, en las sucesivas *Pair Programming*.

¿En qué consiste la programación de a pares? Sencillo. Dos programadores, sentados frente a una misma computadora, cada uno cumpliendo un rol diferente. Las combinaciones y características de este rol, no solo son variables, sino que además, son inmensas, permitiendo la libertad de "ser originalmente creativos".

Algunas de estas combinaciones (tal vez las más frecuentes), suelen ser:

- Uno escribe el código, mientras que otro lo va revisando;
- Un programador más avanzado, programa, mientras va explicando lo desarrollado a otro menos experto;

- Los dos programadores piensan en como resolver el código, y uno de ellos lo escribe.

De esa forma y según las necesidades de cada equipo, las funciones de cada programador, podrán variar ilimitadamente.

PRÁCTICA #7: ENTREGAS CORTAS (SHORT RELEASES)

Se busca hacer entregas en breves lapsos de tiempo, incrementando pequeñas funcionalidades en cada iteración. Esto conlleva a que el cliente pueda tener una mejor experiencia con el Software, ya que lo que deberá probar como nuevo, será poco, y fácilmente asimilable, pudiendo sugerir mejoras con mayor facilidad de implementación.

> *Las entregas cortas, reducen la curva de aprendizaje sobre la aplicación.*

PRÁCTICA #8: TESTING

En esta práctica, podemos encontrar tres tipos de test -pruebas- propuestos por eXtreme Programming:

1. Test Unitarios
2. Test de aceptación
3. Test de integración

Los primeros, consisten en testear el código de manera unitaria (individual) mientras se va programando. Técnica conocida como **Unit Testing**, la cual forma parte de la técnica **TDD (Test-driven development** *-desarrollo guiado por pruebas-*). Veremos esta técnica detalladamente, con ejemplos reales en Python y PHP y hablaremos de los Frameworks para Unit Testing, en los próximos capítulos.

Los **Test de Aceptación** están más avocados a las pruebas de funcionalidad, es decir al comportamiento funcional del código. Estas pruebas, a diferencia de los Test Unitarios, son definidas por el cliente y llevadas a cabo mediante herramientas como *Selenium*, por ejemplo, y basadas en Casos de Uso reales que definirán si la funcionalidad desarrollada, cumple con los objetivos esperados (criterios de aceptación). Puedes leer más sobre las pruebas de aceptación en Wikipedia.

Los **Test de Integración**, como su nombre lo indica, tienen por objeto, integrar todos los test que comforman la aplicación, a fin de validar el correcto funcionamiento de la misma, evitando que nuevos desarrollos, dañen a los anteriores.

PRÁCTICA #9: CÓDIGO ESTÁNDAR (CODING STANDARDS)

Los estándares de escritura del código fuente, son esenciales a la hora de programar. Permiten hacer más legible el código y más limpio a la vez de proveer a otros programadores, una rápida visualización y entendimiento del mismo.

Algunos lenguajes de programación como Python, poseen sus propias reglas de estilo, estandarizando la escritura del código fuente, por medio de las PEP.

Otros lenguajes "menos prolijos" como PHP, no tienen reglas de estilo o estándares definidos (aunque Zend Framework, está realizando un esfuerzo por proponerlos), y es aquí donde el equipo deberá ponerse de acuerdo, definir y documentar sus propias reglas. Más adelante, también veremos como hacerlo.

PRÁCTICA #10: PROPIEDAD COLECTIVA (COLLECTIVE OWNERSHIP)

Para eXtreme Programming no existe un programador "dueño" de un determinado código o funcionalidad. La propiedad colectiva del código tiene por objetivo que **todos los miembros del equipo conozcan "qué" y "cómo" se está desarrollando el sistema**, evitando así, lo que sucede en muchas empresas, que existen "programadores dueños de un código" y cuando surge un problema, nadie más que él puede resolverlo, puesto que el resto del equipo, desconoce cómo fue hecho y cuáles han sido sus requerimientos a lo largo del desarrollo.

PRÁCTICA #11: INTEGRACIÓN CONTINUA (CONTINUOUS INTEGRACIÓN)

La integración continua de XP propone que todo el código (desarrollado por los miembros del equipo) encuentren un punto de alojamiento común en el cuál deban enviarse los nuevos desarrollos, diariamente, previo correr los test de integración, a fin de verificar que lo nuevo, no "rompa" lo

anterior.

Estos puntos de alojamiento en común, suelen ser repositorios, los cuales pueden manejarse ampliando las ventajas de la integración, mediante software de control de versiones como Bazaar, Git o SVN (entre otros).

> *En este punto, los **Test de Integración**, juegan un papel fundamental, puesto que de ellos depende la integración de lo nuevo con lo ya desarrollado. Independientemente de los repositorios, **jamás se logrará una integración continua real, si no existen Test de Integración o si éstos fallan**.*

Más adelante, veremos como integrar un desarrollo, utilizando Bazaar, la **herramienta libre** creada y mantenida por **Canonical** -creadores del Sistema Operativo **Ubuntu** GNU/Linux-.

PRÁCTICA #12: JUEGO DE PLANIFICACIÓN (PLANNING GAME)

Con diferencias pero grandes similitudes con respecto a la planificación del Sprint en Scrum, la dinámica de planificación de eXtreme Programming llevada a cabo al inicio de la iteración, suele ser la siguiente:

1. **El cliente presenta la lista de las funcionalidades** deseadas para el sistema, escrita con formato de "Historia de Usuario", en la cual se encuentra definido el

comportamiento de la misma con sus respectivos criterios de aceptación.

2. **El equipo de desarrollo estima el esfuerzo** que demanda desarrollarlas, así como el esfuerzo disponible para el desarrollo (las iteraciones en XP, suelen durar 1 a 4 semanas) . Estas estimaciones pueden hacerse mediante cualquiera de las técnicas de estimación, vista anteriormente.

3. **El cliente decide que Historias de Usuario desarrollar** y en qué orden.

Programación de a pares y Coding Dojo: ¿quién dijo que el trabajo es aburrido?

> *El Coding Dojo es una reunión de programadores formada con el objetivo de resolver un determinado desafío, la cual utiliza la programación de a pares como punto de partida para enfrentar los retos propuestos.*

¿Por qué "Dojo"?

Dojo es un término de origen japonés, mediante el cual se

designa a los espacios destinados al aprendizaje, la meditación y la sabiduría. Quién ocupa el lugar de "guía" en un Dojo, es llamado Sensei.

En Japón, el término se emplea para designar también, a los lugares donde se practica e instruye, sobre las artes marciales. De igual manera, en el resto del mundo, Dojo se emplea para referirse a todo lugar en el cuál se practiquen dichas artes marciales.

Por ello, el significado semántico de Dojo, se refiere a la **búsqueda de la perfección** y de allí, es que se adopta el término de "Coding Dojo": **el lugar donde los programadores se reúnen en búsqueda de perfeccionarse profesionalmente.**

¿Para qué hacer en un Coding Dojo? ¿Cuál es la finalidad?

Para un programador, participar de un Coding Dojo es una experiencia vital para su carrera. La finalidad de un Coding Dojo, es **aprender de otros programadores y adquirir nuevas técnicas**.

Un Coding Dojo, es un lugar exento de competitividad que se sostiene sobre la base de un **espíritu de colaboración mutua** entre todos los participantes. Esto, es un concepto fundamental, ya que marca una de las mayores diferencias, con cualquier otro tipo de eventos para programadores.

Duración de un Coding Dojo

En la práctica, un Coding Dojo suele durar entre 4 y 9 horas.

¿Qué se hace en un Coding Dojo?

Generalmente, en un Coding Dojo, se emplea una de las siguientes modalidades:

1. Codekata
2. Randori

Codekata en el Coding Dojo

En el **codekata** lo que se busca es **perfeccionar una técnica de programación**. El kata (palabra de origen japonés [型] que significa "forma") se utiliza a menudo en el ámbito de las Artes Marciales japonesas (de forma muy intensiva en el Karate Do y menos intensiva en el Judo, por ejemplo), para referirse al refinamiento de los diversos "movimientos" y "técnicas" esenciales de cada arte marcial.

El Kata en el Coding Dojo

Llevado metafóricamente a la práctica informática, un Kata (o codekata), consiste en presentar una solución (técnica) ya probada y refinada a un problema informático concreto. Se presenta la solución, se explica en detalle, se prueba y finalmente, la técnica presentada, se perfecciona mediante ensayos.

Un Randori Coding Dojo

En el randori lo que se busca es **encontrar una solución a un determinado problema planteado**. El Randori (también de origen japonés) es la práctica de entrenamiento opuesta al Kata, la cual consiste en un "entrenamiento libre".

El Randori en el Coding Dojo

Llevado metafóricamente a la práctica informática, el Randori plantea en principio, un problema al cual habrá que encontrarle una solución (técnica que en un futuro Coding Dojo, podrá ser utilizada como Kata).

En el randori, **se utiliza la programación de a pares** (pair programming, una técnica propuesta por la Programación Extrema o eXtreme Programming).

En la programación de a pares, se van turnando parejas de programadores que a la vez rotan su rol entre ambos (comúnmente, llamados "piloto" y "copiloto").

La pareja de programadores irá analizando y "codeando" alternativas, mientras las explica a la audiencia. Al mismo tiempo, la audiencia puede "colaborar" en la resolución del problema planteado.

Tras un determinado tiempo, el programador que estaba en el teclado (piloto) intercambia su rol con el copiloto. Finalmente, la pareja de programadores, sede el lugar a una nueva pareja de programadores y esto se repite, hasta que se agoten las parejas en la sala.

Ventajas de implementar un Coding Dojo en el lugar de trabajo de forma periódica

Si bien son decenas las ventajas que aquí podrían enumerarse, me centraré en aquellas que considero más relevantes:

1. **Se mantiene activo y motivado al equipo de programadores**, ya que por las características psicológicas de un programador, existe una probada necesidad de satisfacer desafíos profesionales en el ámbito laboral.

2. Es una forma de **fomentar el trabajo colaborativo en equipo y reducir la competencia entre miembros de un mismo equipo** (la cual generalmente, es la principal causa de los ambientes de tensión).
3. Representa un medio libre de inversión, para **capacitar a los desarrolladores**, puesto que, sumando los conocimientos y habilidades de todo el grupo, se perfeccionan las técnicas individuales a la vez de las colectivas.

4. Frente a problemas reales surgidos en un proyecto actual, es una **posibilidad certera, efectiva y rápida, de encontrar una solución eficiente al conflicto**, resolverlo y evitar retrasos en el proyecto.

5. **La empresa pasa a ser algo más que el lugar de trabajo**, ya que se convierte en un Dojo: el lugar ideal para perfeccionarse profesionalmente y adquirir sabiduría.

Cuándo y cómo implementar el Coding Dojo en la empresa

Sin dudas, el mejor momento para implementar un Coding Dojo, es el último día de una iteración mensual.

Hacerlo al finalizar una iteración, otorga un beneficio adicional a las ventajas listadas anteriormente: **se comienza la nueva iteración, con conocimientos y técnicas reforzados.**

Sin dudas, el último viernes de cada mes, es el momento ideal para implementar un Coding Dojo en la empresa, puesto que de esta forma, los programadores que hayan quedado motivados por la nueva técnica adquirida, seguramente utilizarán su fin de semana para reforzarla y llegar el lunes "con más fuerza".

TDD – Test-Driven Development

Entre las prácticas técnicas sugeridas por XP, nos encontramos con la técnica de programación TDD, del inglés *Test-Driven Developmen* (Desarrollo Guiado por Pruebas).

A muchos asusta esta práctica por el simple hecho de ser "desconocida" y resultar su descripción, algo confusa: ¿Qué es a caso, aquello de hacer un test antes de programar?

Pero no dejes que el miedo a lo desconocido te gane, que aprender a programar guiándote por test, es algo realmente simple, divertido y sobre todo, muy productivo.

¿Qué es el desarrollo -o programación- guiado por pruebas?

TDD es una técnica de programación que consiste en guiar el desarrollo de una aplicación, por medio de Test Unitarios.

Los Test Unitarios (Unit Testing) no son más que algoritmos que emulan lo que la aplicación se supone debería hacer, convirtiéndose así, en un modo simple de probar que lo que "piensas" programar, realmente funciona.

Para verlo de forma sencilla, imagina que estás desarrollando una aplicación, que necesita un algoritmo que sume dos números. Sabes perfectamente como hacerlo:

```
function sumar_dos_numeros($a, $b) {
    return $a + $b;
}

$a = 10;
$b = 25;

$suma = sumar_dos_numeros($a, $b);

echo $suma; // salida: 35
```

¿Sencillo verdad? ¿Para qué necesitarías complicarte la vida con eso de los test?

```
function sumar_dos_numeros($a, $b) {
    return $a + $b;
}

$a = 'Zanahorias: 10';
$b = '25 hinojos;

$suma = sumar_dos_numeros($a, $b);

echo $suma; // salida: 25
```

¿a caso 10 zanahorias más 25 hinojos no debería dar como resultado, 35 vegetales? Hasta que no lo pruebas, tal vez no sepas que suceda. Y hasta no saber que sucede, no sabrás como solucionarlo.

Y para esto, sirven los Test, ya que **te guiarán para saber qué, cómo, cuáles y cuántos algoritmos necesitarás desarrollar para que tu aplicación haga lo que se supone debe hacer.**

Es entonces, que en el caso anterior, los test que se desarrollen previamente, irán guiando tu desarrollo hasta encontrar la solución a todos los problemas. Y siguiendo el ejemplo, sabrás

que evidentemente, necesitarás de algún algoritmo extra: validas los parámetros antes de sumarlos y, o los conviertes a enteros o retornas un mensaje de error. En definitiva, **los Test Unitarios serán una guía para entender como funciona el código, ayudándote a organizarlo de manera clara, legible y simple**.

Carlos Blé Jurado[9] en su libro **Diseño Ágil con TDD**[10] nos define al TDD como:

> "[...] la respuesta a las grandes preguntas: ¿Cómo lo hago? ¿Por dónde empiezo? ¿Cómo se qué es lo que hay que implementar y lo que no? ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente? [...]"

Según **Kent Beck** -uno de los principales fundadores de la metodología XP-, **implementar TDD nos otorga grandes ventajas:**

1. **La calidad del software aumenta** disminuyendo prácticamente a cero, la cantidad de *bugs* en la aplicación;
2. **Conseguimos código altamente reutilizable** puesto que los test nos obligan a desarrollar algoritmos genéricos;
3. **El trabajo en equipo se hace más fácil, une a las personas,** ya que al desarrollar con test, nos aseguramos no romper funcionalidades existentes de la aplicación;

9 Carlos Blé es un desarrollador de Software y emprendedor español, con una gran trayectoria en la ingeniería de sistemas ágiles. Visitar su blog: www.carlosble.com

10 Diseño Ágil con TDD. ISBN 978-1-4452-6471-4. Creative Commons-ND

4. **Nos permite confiar en nuestros compañeros aunque tengan menos experiencia.** Esto es, debido a que el hecho de tener que desarrollar test antes de programar el algoritmo definitivo, nos asegura -independientemente del grado de conocimiento y experiencia del desarrollador- que el algoritmo efectivamente, hará lo que se supone, debe hacer y sin fallos;
5. **Escribir el ejemplo (test) antes que el código nos obliga a escribir el mínimo de funcionalidad necesaria, evitando sobrediseñar**, puesto que desarrollando lo mínimamente indespensable, se obtiene un panorama más certero de lo que la aplicación hace y cuál y cómo es su comportamiento interno;
6. **Los tests son la mejor documentación técnica que podemos consultar a la hora de entender qué misión cumple cada pieza del rompecabezas**, ya que cada test, no es más que un "caso de uso" traducido en idioma informático.

Test Unitarios

Los Test Unitarios (o Unit Testing), representan el alma de la programación dirigida por pruebas. Son test que se encargan de verificar -de manera simple y rápida- el comportamiento de una parte mínima de código, de forma independiente y sin alterar el funcionamiento de otras partes de la aplicación.

Características de los Test Unitarios

Un Test Unitario posee cuatro características particulares que debe guardar para considerarse "unitario". Estas cualidades son:

1. Atómico:

Prueba una parte mínima de código.

Dicho de manera simple, cada test unitario debe probar una -y solo una- "acción" realizada por un método.

Por ejemplo, para un método que retorna el neto de un monto bruto más el IVA correspondiente, deberá haber un test que verifique recibir en forma correcta el importe bruto, otro que verifique el cálculo del IVA sobre un importe bruto y finalmente, un tercer test unitario que verifique el cálculo de un importe bruto más su IVA correspondiente.

2. Independiente:

Cada Test Unitario DEBE ser independiente de otro.

Por ejemplo, siguiendo el caso anterior, el test que verifique la suma de un importe bruto más su IVA correspondiente, no debe depender del test que verifica el cálculo del IVA.

3. Inocuo:

Podría decirse que cada test unitario debe ser inofensivo para el Sistema. Un test unitario DEBE poder correrse sin alterar ningún elemento del sistema, es decir, que no debe, por ejemplo, agregar, editar o eliminar registros de una base de datos.

4. Rápido:

La velocidad de ejecución de un test unitario cumple un papel fundamental e ineludible en el desarrollo guiado por pruebas, ya que de la velocidad de ejecución de un

test, dependerá de manera proporcional, la velocidad con la que una funcionalidad se desarrolle.

Anatomía

Los Test Unitarios se realizan, en cualquier lenguaje de programación, mediante herramientas -Frameworks- con un **formato** determinado, conocido como **xUnit**.

De allí, que los frameworks para Unit Testing que cumplen con dicho formato, suelen tener nombres compuestos por una abreviatura del lenguaje de programación, seguida del término "unit": **PyUnit** (Python), **PHPUnit** (PHP), **ShUnit** (Shell Scripting), **CppUnit** (C++), etc.

Exceptuando el caso de Shell Scripting, **los frameworks xUnit, utilizan el paradigma de programación orientada a objetos** (OOP) tanto en su anatomía de desarrollo como para su implementación (creación de test unitarios).

Por lo tanto, los Test Unitarios se agrupan en **clases**, denominadas *Test Case*, que heredan de una clase del framework xUnit, llamada **xTestCase**:

```
import unittest

class BalanceContableTestCase(unittest.TestCase):
    # Métodos
```

*Creación de una clase Test Case con **PyUnit** (Python). La clase **hereda de** **unittest.TestCase***

```
ini_set('include_path', '.:/usr/share/php:/usr/share/pear');

class BalanceContableTest extends PHPUnit_Framework_TestCase {
   # Métodos
}
```

*Creación de una clase Test Case con **PHPUnit** (PHP). La clase **hereda de** **PHPUnit_Framework_TestCase***

Los métodos contenidos en una clase Test Case, pueden o no, ser Test Unitarios. Los Test Unitarios contenidos en una clase Test Case, deben contener el prefijo **test_** en el nombre del método a fin de que el framework los identifique como tales.

```
import unittest

class BalanceContableTestCase(unittest.TestCase):

    def test_calcular_iva(self):
        # Algoritmo
```

Definición de un método "test" con PyUnit en Python

```
ini_set('include_path', '.:/usr/share/php:/usr/share/pear');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva() {
        # Algoritmo
    }

}
```

Definición de un método "test" con PHPUnit. Nótese que en PHP, los métodos de una clase Test Case DEBEN ser métodos públicos.

Otra ventaja que los frameworks xUnit nos proveen, es la facilidad de poder crear dos métodos especiales dentro de una clase Test Case -que no son test-, los cuales están destinados a preparar el escenario necesario para correr los test de esa clase y eliminar aquello que se desee liberar,

una vez que el test finalice. Estos métodos, son los denominados **setUp()** y **tearDown()** respectivamente:

```
import unittest

class BalanceContableTestCase(unittest.TestCase):

    def setUp(self):
        self.importe_bruto = 100
        self.alicuota_iva = 21

    def tearDown(self):
        self.importe_neto = 0
        self.alicuota_iva = 0

    def test_calcular_iva():
        # Algoritmo
```

Los métodos setUp() y tearDown() en PyUnit

```
ini_set('include_path', '.:/usr/share/php:/usr/share/pear');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->importe_bruto = 100;
        $this->alicuota_iva = 21;
    }

    public function tearDown() {
        $this->importe_bruto = 0;
        $this->alicuota_iva = 0;
    }

    public function test_calcular_iva() {
        # Algoritmo
    }

}
```

Los métodos setUp() y tearDown() en PHPUnit

Esta anatomía dual -por un lado, la del Framework y por otro, la de utilización o implementación de éste-, se logrará finalmente, dividiendo a cada Test Case -imaginariamente- en tres partes, identificadas por las siglas AAA las cuáles representan a las tres "acciones" que son necesarias llevar a cabo, para dar forma a los Tests:

Arrange, Act and Assert

(preparar, actuar y afirmar)

Preparar los test consiste en establecer -setear o configurar- todo aquello que sea necesario para que cada uno de los métodos "test" pueda ejecutarse. Esto puede ser, la declaración de propiedades comunes a todos los test, la instancia a objetos, etc.

Cuando estas "preparaciones previas" sean comunes a todos los test, deberán valerse del método setUp() para ser creadas. De lo contrario, se crearán métodos -a modo de helpers- dentro de la clase Test Case, a los cuáles cada uno de los test, recurra cuando los necesite. Vale aclarar que estos "helpers" NO podrán contener el prefijo "test" en su nombre.

Actuar, se refiere a hacer la llamada al código del Sistema cubierto por Test (SUT) que se desea probar. Esto se conoce como "cobertura de código" o Code Coverage:

```
class BalanceContableTest extends PHPUnit_Framework_TestCase {

    # Arrange (preparar)
    public function setUp() {
        // importar la clase a ser testeada
        require_once('/contabilidad/models/balance_contable.php');

        // setear propiedades comunes
        $this->importe_bruto = 100;
        $this->alicuota_iva = 21;
    }

    public function tearDown() {
        $this->importe_bruto = 0;
        $this->alicuota_iva = 0;
    }
```

```
    public function test_calcular_iva() {
        # Act (actuar)
        // Instanciar al objeto que será probado
        $this->coverage = new BalanceContable();

        // modificar las propiedades del objeto
        $this->coverage->importe_bruto = $this->importe_bruto;
        $this->coverage->alicuota_iva = $this->alicuota_iva;

        // invocar al método que se está testeando
        $result = $this->coverage->calcular_iva();

        # Assert (afirmar)
        // sentencias

    }

}
```

Finalmente, **afirmar el resultado de un test**, se refiere a **invocar a los métodos assert del Framework xUnit**, que sean necesarios para afirmar que el resultado obtenido durante la actuación, es el esperado (más adelante, veremos cuáles son estos métodos `assert` de los cuales disponemos):

```
class BalanceContableTest extends PHPUnit_Framework_TestCase {

    # Arrange (preparar)
    public function setUp() {
        // importar la clase a ser testeada
        require_once('/contabilidad/models/balance_contable.php');

        // setear propiedades comunes
        $this->importe_bruto = 100;
        $this->alicuota_iva = 21;
    }

    public function tearDown() {
        $this->importe_bruto = 0;
        $this->alicuota_iva = 0;
    }

    public function test_calcular_iva() {
        # Act (actuar)
        // Instanciar al objeto que será probado
        $this->coverage = new BalanceContable();

        // modificar las propiedades del objeto
        $this->coverage->importe_bruto = $this->importe_bruto;
```

```
        $this->coverage->alicuota_iva = $this->alicuota_iva;

        // invocar al método que se está testeando
        $result = $this->coverage->calcular_iva();

        # Assert (afirmar)
        $this->assertEquals(121, $result);

    }

}
```

Algoritmo para escribir pruebas unitarias

Existe un algoritmo (o por qué no, "una fórmula"), para escribir Test Unitarios bajo TDD, el cual consiste en:

PRIMER PASO: Escribir el Test y hacer que falle

Para ello, lo primero que haremos será crear nuestra clase Test Case (nótese que ningún otro código ha sido escrito al momento, para esta aplicación).

Nos baseremos en el mismo ejemplo que venimos siguiendo, donde lo que se necesita es calcular el IVA a un monto bruto dado.

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva() {
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 1500;
        $this->coverage->alicuota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

}

?>
```

archivo: */MyApp/Tests/BalanceContableTest.php*

Nuestro test, nos está guiando el desarrollo, diciéndonos que:

1. Debemos crear una clase llamada `BalanceContable` que la guardaremos en el archivo `BalanceContable.php`
2. Esta clase debe tener dos propiedades: `importe_bruto` y `alicuota_iva`
3. La clase debe contener un método llamado c`alcular_iva()`

¿Qué hace nuestro test? Verificará que el método `calcular_iva()` retorne el valor `315`. Este valor, debe corresponder al 21% (`alicuota_iva`) de 1500 (`importe_bruto`).

Entonces, el siguiente paso, será crear aquello que nuestro Test, nos guió a hacer:

```
<?php
class BalanceContable {

    public $importe_bruto;
    public $alicuota_iva;

    public function calcular_iva() {
    }

}

?>
```

archivo: */MyApp/contabilidad/models/BalanceContable.php*

Finalmente, correremos nuestro Test, y éste, deberá

fallar:

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests
PHPUnit 3.4.5 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) BalanceContableTest::test_calcular_iva
Failed asserting that <null> matches expected <integer:315>.

/home/eugenia/borrador/MyApp/Tests/BalanceContableTest.php:11

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

En la muestra de consola anterior, corrimos el Test creado mediante el comando `phpunit` al cual le pasamos como parámetro la carpeta `Tests` que contiene (y contendrá) todos nuestros test. Se resaltan en color naranja, las explicaciones arrojadas por PHPUnit, donde nos dice que el método `test_calcular_iva` del Test Case `BalanceContableTest` ha fallado, en afirmar que `NULL` coincidió con el valor 315 esperado de tipo entero.

Con lo anterior, nos aseguramos que el Test falle, cuando se espera que así sea.

SEGUNDO PASO: Escribir la mínima cantidad de código para que el test pase.

De la misma forma que nos aseguramos que el test fallará cuando deba hacerlo, ahora debemos asegurarnos de que el test NO falle cuando no lo esperamos.

Para ello, debemos evitar escribir -de buenas a primeras-, el algoritmo de nuestro método. Por el contrario, debemos encontrar la expresión mínima que nos retorne el resultado que esperamos. Es decir, que nos retorne el entero 315. Entonces, editamos nuestra clase `BalanceContable` y agregamos la mínima expresión necesaria al método `calcular_iva()`:

```
<?php
class BalanceContable {

    public $importe_bruto;
    public $alicuota_iva;

    public function calcular_iva() {
        return 315;
    }

}

?>
```

Volvemos a correr el test para asegurarnos de que esta vez, NO falle:

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests
PHPUnit 3.4.5 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 4.00Mb

OK (1 test, 1 assertion)
```

TERCER PASO: Escribir un nuevo test y hacer que falle

Hemos escrito la cantidad mínima de código para que nuestro primer test, pase. Lo hemos hecho fallar, cuando aún no existía algoritmo o instrucción en el método probado. Es hora de crear un nuevo test que falle, cuando ya existe una mínima porción de código escrita (o modificar el existe, cambiando los parámetros de éste). Agregaremos entonces, un nuevo test y modificaremos el nombre del anterior, de manera tal que nuestro Test Case se vea como sigue:

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva_con_1500_esperando_315() {
```

```
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 1500;
        $this->coverage->alicuota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

    public function test_calcular_iva_con_2800_esperando_588() {
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 2800;
        $this->coverage->alicuota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(588, $result);
    }
}

?>
```

Ahora nuestro segundo test, debe fallar:

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests
PHPUnit 3.4.5 by Sebastian Bergmann.

.F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) BalanceContableTest::test_calcular_iva_con_2800_esperando_588
Failed asserting that <integer:315> matches expected <integer:588>.

/home/eugenia/borrador/MyApp/Tests/BalanceContableTest.php:19

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

CUARTO PASO: Escribir el algoritmo necesario para hacer pasar el test

Es hora de programar lo que nuestro método, realmente necesita y sin *hardcodear* el retorno de resultados. Modificaremos nuestro método `calcular_iva()` de la clase `BalanceContable` a fin de escribir el algoritmo que efectivamente se encargue de calcular el valor de retorno:

```
<?php
class BalanceContable {
```

```
    public $importe_bruto;
    public $alicuota_iva;

    public function calcular_iva() {
        $iva = $this->alicuota_iva / 100;
        $neto = $this->importe_bruto * $iva;
        return $neto;
    }
}

?>
```

Nuestros test, ahora pasarán:

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests
PHPUnit 3.4.5 by Sebastian Bergmann.

..

Time: 0 seconds, Memory: 4.00Mb

OK (2 tests, 2 assertions)
```

Es válido hacer notar, que a medida que se van escribiendo los test, éstos, no solo guiarán nuestro desarrollo en cuanto a "lo nuevo que se debe escribir", sino que además, **nos irán obligando a refactorizar el código constantemente**, en principio, mostrándonos el camino para eliminar redundancias y **crear un diseño más simple**:

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva_con_1500_esperando_315() {
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 1500;
        $this->coverage->alicuota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

    public function test_calcular_iva_con_2800_esperando_588() {
```

```
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 2800;
        $this->coverage->alicuota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(588, $result);
    }
}

?>
```

La redundancia anterior, nos está diciendo que existen elementos de preparación (arrange) comunes para nuestra Test Case:

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->coverage = new BalanceContable();
        $this->coverage->alicuota_iva = 21;
    }

    public function test_calcular_iva_con_1500_esperando_315() {
        $this->coverage->importe_bruto = 1500;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

    public function test_calcular_iva_con_2800_esperando_588() {
        $this->coverage->importe_bruto = 2800;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(588, $result);
    }
}

?>
```

> ***RECUERDA:*** *cada cambio que se haga al código tanto del test como del SUT (código de la aplicación cubierta por test), requiere que se vuelva a correr el test.*

Unit Testing con PHPUnit

Existen varios frameworks xUnit para Unit Testing en PHP, pero sin dudas, el único que ha demostrado contar con una gran cobertura de código, estabilidad y buena documentación, es PHPUnit.

El **manual oficial de PHPUnit (en inglés)** puede encontrarse en: http://www.phpunit.de/manual/3.6/en/.

Se puede **instalar PHPUnit** en sistemas operativos **GNU/Linux**, ejecutando los siguientes comandos:

```
sudo pear upgrade PEAR
pear config-set auto_discover 1
pear install pear.phpunit.de/PHPUnit
```

Aunque en **distribuciones basadas en Debian**, puede hacerse directamente mediante la instalación del paquete **`phpunit`** con **`apt-get`**:

```
sudo apt-get install phpunit
```

Métodos Assert de PHPUnit

PHPUnit provee una gran cantidad de métodos **`assert`** cuyas referencias, podemos encontrar en el **Capítulo 4 del manual oficial**:

http://www.phpunit.de/manual/3.6/en/writing-tests-for-phpunit.html

Algunas características comunes de los métodos assert, son:

- Generalmente, por cada método **assert** existe su opuesto: `assertContains()` y `assert`**`Not`**`Contains()`.
- A la vez, cada método `assert` deberá recibir mínimamente un parámetro que será el resultado de ejecutar el código del SUT.
- Adicionalmente, a cada método `assert`, se le puede pasar como parámetro opcional, un mensaje personalizado para ser arrojado en caso de error (generalmente, será el último parámetro).
- Los métodos `assert` que requieren el paso de dos parámetros obligatorios (valores que deben compararse entre sí), generalmente guardan el siguiente orden:

  ```
  metodoAssert($valor_esperado, $valor_recibido)
  ```

  Es decir, que en esos casos, siempre el primer parámetro será el valor esperado y el segundo parámetro, el valor recibido por la ejecución del código SUT.

Veamos algunos ejemplos puntuales:

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->coverage = new BalanceContable();
        $this->coverage->alicuota_iva = 21;
    }

    // AssertEquals($valor_esperado, $valor_recibido)
    public function test_calcular_iva() {
        $this->coverage->importe_bruto = 1500;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }
```

```
    // AssertTrue($valor_recibido)
    public function test_alcanzado_por_impuesto_de_importacion_con_160() {
        $this->coverage->importe_bruto = 160;
        $result = $this->coverage->alcanzado_por_impuesto_de_importacion();
        $this->assertTrue($result);
    }

     // AssertNull($valor_recibido)
    public function test_alcanzado_por_impuesto_de_importacion_con_143() {
        $this->coverage->importe_bruto = 143;
        $result = $this->coverage->alcanzado_por_impuesto_de_importacion();
        $this->assertNull($result);
    }
}

?>
```

Código fuente de la clase Test Case

```
<?php

class BalanceContable {

    public $importe_bruto;
    public $alicuota_iva;

    # Calcular IVA sobre un importe bruto
    public function calcular_iva() {
        $iva = $this->alicuota_iva / 100;
        $neto = $this->importe_bruto * $iva;
        return $neto;
    }

    # Determinar si un importe paga impuesto de importación
    public function alcanzado_por_impuesto_de_importacion() {
        // importes mayores a 150 USD pagan impuesto
        if($this->importe_bruto > 150) {
            return True;
        }
    }

}

?>
```

Código fuente del SUT

Ejercicio

Escribir el código SUT del siguiente Test Case:

```
<?php
require_once('Usuario.php');

class UsuarioTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->coverage = new Usuario();
        $this->coverage->username = "juanperez75";
        $this->coverage->password = md5("pablito: clavo_1_palito");
    }

    public function test_set_usuario_esperando_key() {
        $result = $this->coverage->set_usuario();
        $this->assertArrayHasKey('Usuario', $result);
    }

    public function test_set_usuario_esperando_juanperez75() {
        $result = $this->coverage->set_usuario();
        $this->assertEquals('juanperez75', $result['Usuario']);
    }

    public function test_set_usuario_esperando_pass() {
        $result = $this->coverage->set_usuario();
        $this->assertEquals(md5('pablito: clavo_1_palito'),
$result['Clave']);
    }

}

?>
```

Unit Testing con PyUnit

PyUnit es el framework xUnit **elegido Oficilamente por Python** desde su versión 1.5.2. Si bien existen muchos otros, generalmente están destinados a ampliar los beneficios de PyUnit para la realización de test más complejos, como el caso de PyDoubles -creado por Carlos Blé-.

Toda la referencia sobre **PyUnit** -a diferencia de PHP- se encuentra en el **manual oficial de Python (en inglés)**: http://docs.python.org/library/unittest.html

PyUnit no necesita ser instalado, ya que desde la versión 2.1, forma parte de la librería estándar de Python, a través del módulo **unittest**.

Métodos Assert de PyUnit

Una lista completa de los métodos **assert** de **PyUnit** puede encontrarse en la **documentación sobre unittest de Python** en la siguiente URL:

http://docs.python.org/library/unittest.html#assert-methods

Una diferencia particular que existe entre PyUnit y frameworks como PHPUnit, es que nos permite efectuar afirmaciones, con una sintaxis bastante simple, sin necesidad de recurrir a métodos assert específicos:

```
assert resultado == valor_esperado
```

En un ejemplo más concreto, podríamos verlo así (donde **coverage** será la instancia al objeto del SUT):

```
assert self.coverage.sumar_dos_numeros(5, 15) == 20
```

Otra **diferencia fundamental con PHPUnit**, es que el método assertIgualdad, posee su nombre en singular:

```
PHPUnit:
      assertEquals($a, $b);

PyUnit:
      asserEqual(a, b)
```

Algunas **características comunes** de los métodos assert, son:

- Al igual que con PHPUnit, generalmente, por **cada método `assert` existe su opuesto**: `assertEqual()` y `assertNotEqual()`.
- También siguiendo nuevamente la línea de PHPUnit, cada método `assert` deberá recibir mínimamente **un parámetro** que será el **resultado** de ejecutar el código del SUT y opcionalmente, como último parámetro, puede recibir **un mensaje personalizado** para ser arrojado en caso de error.
- **A diferencia de PHPUnit,** los métodos `assert` que requieren el paso de dos parámetros obligatorios (valores que deben compararse entre sí), generalmente guardan el siguiente orden:

  `metodoAssert(valor_recibido, valor_esperado)`

  Es decir, que en esos casos, siempre **el primer parámetro será el valor recibido** por la ejecución del código SUT y el **segundo parámetro, el valor esperado**.

Veamos el ejemplo realizado anteriormente en PHP, pero esta vez, en Python con PyUnit:

```
# -*- coding: utf-8 *-*
import sys

import unittest

sys.path.append('/path/a/la/app')

from myapp.modules.balance_contable import BalanceContable


class BalanceContableTestCase(unittest.TestCase):

    # setUp()
    def setUp(self):
        self.coverage = BalanceContable()
        self.coverage.alicuota_iva = 21

    # assertEqual(valor_recibido, valor_esperado)
    def test_calcular_iva(self):
          self.coverage.importe_bruto = 2500
          result = self.coverage.calcular_iva()
          self.assertEqual(result, 525)

      # AssertTrue(valor_recibido)
    def test_alcanzado_por_impuesto_de_importacion_con_160(self):
        self.coverage.importe_bruto = 160
        result = self.coverage.alcanzado_por_impuesto_de_importacion()
        self.assertTrue(result)

    # AssertIsNone(valor_recibido)
    def test_alcanzado_por_impuesto_de_importacion_con_143(self):
        self.coverage.importe_bruto = 143
        result = self.coverage.alcanzado_por_impuesto_de_importacion()
        self.assertIsNone(result)

# Necesario para correr los test si es llamado por línea de comandos
if __name__ == "__main__":
    unittest.main()
```

Código fuente de la clase Test Case

Nótese que el método `assertEquals` de PHPUnit, se denomina `assertEqual` (en singular) en PyUnit y que en reemplazo del método `assertNull`, PyUnit propone `assertIsNone` (esto es debido a que Python no retorna valores nulos como tales, sino como "None").

```
# -*- coding: utf-8 *-*


class BalanceContable(object):

    def __init__(self):
        self.importe_bruto = 0
        self.alicuota_iva = 0

    # Calcular IVA sobre un importe bruto
    def calcular_iva(self):
        iva = self.importe_bruto * self.alicuota_iva / 100
        return iva

    # Determinar si un importe paga impuesto de importación
    def alcanzado_por_impuesto_de_importacion(self):
        # importes mayores a 150 USD pagan impuesto
        if self.importe_bruto > 150:
            return True
```

Código fuente del SUT

Corriendo test por línea de comandos

Desde la versión 2.7 de Python, ya no es necesario realizar "maniobras" o crear Test Suites, con el único fin de correr todos los Test de nuestra aplicación, de un solo paso.

Desde Python 2.7 todos los test de una aplicación, pueden correrse mediante el comando `discover`:

```
eugenia@cocochito:~/proyectos$ python -m unittest discover
```

`discover`, "descubrirá" todos los test, identificándolos por el nombre del archivo: debe comenzar por el prefijo "`test`" (discover utiliza la expresión regular **`test*.py`** para identificar Test Cases).

Además, debe tenerse en cuenta que el nombre de los métodos de prueba, también deben comenzar por el prefijo

`test`. Nótese que también es posible correr una única Test Case:

```
eugenia@cocochito:~/proyectos$ python myapp/Test/test_balance_contable.py
```

O un test en particular:

```
eugenia@cocochito:~/proyectos$ python myapp/Test/test_balance_contable.p
BalanceContableTestCase.test_calcular_iva
```

También es posible, pasar el parámetro **`-v`** a fin de obtener un reporte más detallado:

```
eugenia@cocochito:~/proyectos$ python -m unittest discover -v
test_alcanzado_por_impuesto_de_importacion_con_143
(Test.test_balance_contable.BalanceContableTestCase) ... ok
test_alcanzado_por_impuesto_de_importacion_con_160
(Test.test_balance_contable.BalanceContableTestCase) ... ok
test_calcular_iva (Test.test_balance_contable.BalanceContableTestCase) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Integración continua

La integración continua es una técnica que se encuentra estrechamente relacionada con la de TDD y veremos cómo llegamos a esta conclusión.

Para entender de qué se habla exactamente, cuando nos referimos a integración continua, deberíamos centrarnos primero en el objetivo de ésta (el "para qué") y luego, tratar de obtener el "cómo" lograrlo.

El **objetivo** de la integración continua, consiste en **integrar las nuevas funcionalidades desarrolladas a las existentes, asegurando que lo nuevo, no arruine lo viejo.** La respuesta a ¿cómo lograr cumplir con este objetivo? Es la que nos definirá de forma exacta, de qué se trata esta técnica. Veamos cuáles son los pasos a seguir para lograr el objetivo.

La técnica de integración contínua, se logra cuando **diariamente, los miembros del equipo se comprometen a:**

1. Escribir Test Unitarios
2. Correr los Test Unitarios asegurando que todos pasen
3. Generar los Test de Integración (es decir, aquellos conformados por un conjunto de Test Unitarios, Test Funcionales, Test de Aceptación y Test de Sistema)
4. Correr los Test de Integración asegurando que todos pasen
5. Unificar todos los desarrollos que hayan pasado los test, en un repositorio local

6. Llevar un control histórico de cambios en el repositorio

Estos seis pasos, son los que finalmente, nos especifican qué es exactamente la integración continua y veremos aquí en detalle, cómo lograr cada uno de estos pasos.

Generación de Test de Integración

Los pasos 1 y 2, los hemos visto en capítulos anteriores. Nos resta ahora, continuar con el paso 3.

El paso 3, está compuesto por cuatro tipo de test:

1. Test Unitarios (que ya hemos visto)
2. Test de Aceptación: son Test Unitarios que prueban los criterios de aceptación definidos por el Dueño de Producto (cliente)
3. Test Funcionales: son test que prueban el conjunto de criterios evaluados mediante los Test Unitarios, los cuales inetgran la funcionalidad completa (es decir, una Historia de Usuario)
4. Test de Sistema: aquellos que prueban los casos de uso, y están más relacionados con la experiencia de usuario que con cuestiones funcionales o de programación.

Test de Aceptación

Los Test de Aceptación, no dejan de ser Test Unitarios, que

corren bajo la responsabilidad del programador.

Estos test, simplemente se encargan de verificar que un criterio de aceptación especificado por el cliente se cumple, lo que conlleva a pensar que **"si un test de aceptación pasa, entonces ese criterio de aceptación es aceptado por el cliente"**.

Los Test de Aceptación son especificados por el cliente, generando un grado de detalle mayor, a cada criterio de aceptación.

Veamos el siguiente ejemplo:

Una Historia de Usuario típica: **Como usuario del sistema puedo calcular el costo final de mi pedido con los gastos de envío incluidos**, posee los siguientes **criterios de aceptación**:

- `Pedidos inferiores a USD 20 tienen un costo de envío de USD 75`
- `Pedidos entre USD 21 y USD 100, tienen un costo de envío de USD 50`
- `Pedidos superiores a USD 100, tienen un costo de envío equivalente al 15% del importe del pedido`

Sobre esa base, el cliente definirá (en su idioma), los Test de Aceptación (los cuales serán criterios de aceptación más detallados, basados en casos de uso reducidos a resultados -es decir, no incluyen aspectos del sistema, como modelos gráficos, etc.-):

- `Pedidos inferiores a USD 20 tienen un costo de envío de USD 75`
    - `Test de Aceptación:`
        - `Pedido de USD 1 retorna USD 76`
        - `Pedido de USD 15 retorna USD 90`

- Pedidos superiores a USD 100, tienen un costo de envío equivalente al 15% del importe del pedido
    - Test de Aceptación:
        - Pedido de USD 125 retorna USD 143.75
        - Pedido de USD 300 retorna USD 345

El Programador, deberá desarrollar estos test, de la misma forma que lo ha hecho con los Test Unitarios, pero ¿cuál es la diferencia? Simplemente, deberá afirmar el resultado de cada test, con los valores sugeridos por el cliente.

Pero para diferenciar los Test de Aceptación de los Unitarios, se debe especificar en el nombre de los métodos de prueba:

```
class CalculadoraDeCostosTestCase(unittest.TestCase):

    # setUp()
    def setUp(self):
        self.coverage = CalculadoraDeCostos()

    def test_calcular_precio_final_con_1_esperando_76(self):
          self.coverage.importe_base = 1
          result = self.coverage.calcular_precio_final()
          self.assertEqual(result, 76)

    def test_calcular_precio_final_con_300_esperando_345(self):
          self.coverage.importe_base = 300
          result = self.coverage.calcular_precio_final()
          self.assertEqual(result, 345)
```

Otra forma menos sutil (y más difícil de implementar cuando el criterio de aceptación es demasiado largo), es la siguiente:

```
    def test_calcular_precio_final_criterio_300_aceptacion_345(self):
          self.coverage.importe_base = 300
          result = self.coverage.calcular_precio_final()
          self.assertEqual(result, 345)
```

Pero lógicamente, el nombre de los test, va en gustos.

Test Funcionales

Al Igual que los Test de Aceptación, los Test Funcionales también corren por cuenta del programador. Los Test Funciones no dejan de ser Test Unitarios y se asemejan a los Test de Aceptación, en el sentido de que los Test de Aceptación, indirectamente, también prueban la funcionalidad completa ya que los criterios de aceptación, corresponden a una Historia de Usuario.

> *Historia de Usuario = Funcionalidad completa*

Es decir, siguiendo con el ejemplo anterior, el método `calcular_precio_final` del SUT, seguramente estará recurriendo a otros métodos -incluso, a métodos de otras clases-, ya que `calcular_precio_final` ES la Historia De Usuario. Ergo, los Test Unitarios, se encargarán de probar métodos más específicos y puntuales:

```
    def test_obtener_diccionario_costos_de_envio(self):
         dummy = {75: [0, 20], 50: [21, 100], 101: 15}
          result = self.coverage.obtener_diccionario_costos_de_envio()
          self.assertDictEqual(dummy, result)
```

Mientras que los Test de Aceptación, como hemos visto, probarán el método final, que englobe a todos los demás. Entonces, el Test Funcional, hará lo mismo, con una diferencia fundamental: es el objetivo el que lo diferencia.

El Test de Aceptación, estará basado en "criterios de negocio", mientras que el Test Funcional -si bien puede ser exactamente idéntico a uno de Aceptación-, se basará en criterios de programación, que muchas veces, no pueden ser contemplados por el cliente y por lo general, tienen a probar aquello que

podría suceder si se provocase, por ejemplo, un fallo en el sistema.

Es decir, al cliente se le puede ocurrir un test de aceptación que diga pedido de USD 1 retorna USD 76, pero muy difícilmente, se le pueda ocurrir "0 retorna un error" o más difícil aún, le surja la idea de probar si el objeto CalculadoraDeEnvio se está conectando de forma correcta con el objeto Pedido.

Test de Sistema

A diferencia de los anteriores, los Test de Sistema, suelen correr por cuenta de *Testers*, o personas que no necesariamente necesitan conta con conocimientos de programación. Como este curso está dirigido exclusivamente a programadores, no nos explayeremos en este tema. No obstante, comentaré muy brevemente, en qué consisten.

Los Test de Sistema, a pesar que el nombre confunde muchísimo -yo preferiría llamarlos Test de Usuario- tiene por objetivo **probar la respuesta del Software frente al usuario**. Estos test, son los únicos que deben realizarse DESPUÉS de haber desarrollado la Historia de Usuario.

En muchos casos, se realizan "utilizando el Software manualmente" y en otros, se generan los test con herramientas de automatización -como el caso de Selenium, para aplicaciones Web based-, que permiten utilizar el software manualmente por única vez, e ir capturando los resultados en tiempo real, para finalmente, generar la automatización de lo que se realizó manualmente. Y en eso consisten: en usar el Software -como usuarios finales- y ver si el Software responde

de la manera que se espera.

Unificación del código en Repositorios

Un Repositorio es un espacio destinado a almacenar información digital. En nuestro caso, lo que se almacenará en ese repositorio, serán los archivos -código fuente, *tarballs*, binarios, etc- de nuestra aplicación.

Sus principales características son:

- Espacio de **almacenamiento centralizado** de, principalmente, el código fuente de la aplicación así como scripts de construcción -en el caso de aplicaciones que requieran ser compiladas o simplemente, necesiten realizar configuraciones especiales, ya sea tanto para continuar desarrollándolas como para ejecutarlas-.
- Para ser efectivos, deben llevar un **control histórico de cambios** que se vayan efectuando en los archivos -preferentemente automático-, permitir el establecimiento de *tags* -etiquetas- que ayuden a identificar diferentes *releases* -versiones-.

Y todas esta característica, son aquellas que nos brindan los programas de control de versiones[11].

11 http://es.wikipedia.org/wiki/Programas_para_control_de_versiones

Sobre los Sistemas de Control de Versiones

Los Sistemas de Control de Versiones (SCV) pueden agruparse en dos tipos:

- Centralizados:

  un único repositorio centralizado administrado por un solo responsable.

- Distribuidos (recomendados):

  donde existe un repositorio central que cada usuario podrá clonar para obtener su propio repositorio -local- e interactuar con con otros repositorios locales.

Entre los **SCV distribuidos** podemos destacar excelentes alternativas **GPL** (Software Libre), como es el caso de -entre otros-, Git (de Linus Torvalds, creador del Kernel Linux en el que se basa el Sistema Operativo GNU/Linux), Mercurial (desarrollado en Python y C) o el magnífico Bazaar, nacido a partir de GNUArch y desarrollado íntegramente en Python por Martin Pool, con el patrocinio de Canonical y **elegido en este curso**.

> *Una gran **ventaja de los SCV** es que **permiten a varios programadores trabajar simultáneamente sobre los mismos archivos**, impidiendo que el trabajo de uno, pise al trabajo de otro.*

Los SCV pueden utilizarse tanto a través de línea de comandos,

como de aplicaciones gráficas. En este curso, nos centraremos en el uso por medio de línea de comandos.

Los SCV, en su mayoría -y a rasgos generales- cuentan con un conjunto de funcionalidades, las cuales, para cada una, existe un determinado comando (generalmente, similar en la mayoría de los SCV).

Integración continua con Bazaar

Bazaar, cuenta con una estructura que puede dividirse en:

1. Repositorio:

   conjunto de revisiones

2. Árbol de trabajo:

   un directorio que contiene las revisiones con sus correspondientes ramas

3. Ramas:

   un conjunto ordenado de las diferentes revisiones con sus archivos correspondientes

4. Revisiones: es una vista espontánea del estado de cada uno de los archivos, en un momento determinado


REVISIÓN 1.1
(Revision)
REVISIÓN 1.2
(Revision)
RAMA 1
(Branch)
REVISIÓN 2.1
(Revision)
REVISIÓN 2.2
(Revision)
RAMA 2
(Branch)
ÁRBOL DE TRABAJO
(Work tree)
REPOSITORIO
(Repository)

Un repositorio, entonces, contiene un árbol de trabajo, quien a la vez, puede estar integrado por varias ramas, las cuales almacenarán las distintas versiones (revisiones) por las que ha ido transitando el Software.

Instalación de Bazaar

Nótese que Bazaar deberá instalarse en cada una de las máquinas donde se desee usar. Esto es, en los ordenadores que contarán con repositorios locales y en el ordenador destinado a actuar como repositorio central.

Para instalar Bazaar, por favor, visita http://wiki.bazaar.canonical.com/Download para obtener las instrucciones necesarias.

Bazaar por línea de comandos

Más adelante veremos todo lo que puede hacerse con Bazaar. Aquí, nos limitaremos a conocer la sintaxis básica.

```
bzr [-h|-v|-q] comando-interno-de-bazaar [argumentos]
```

Las opciones globales (opcionales) **-h, -v y -q**, significan ayuda (sobre el comando interno), modo verboso (despliega mayor información sobre lo que sucede mientras se ejecuta el comando en cuestión) y silenciar (solo desplegará errores y advertencias), respectivamente.

Veremos los comandos internos de bazaar, en el siguiente título. No obstante, citaré algunos comandos útiles a la propia aplicación, que pueden servirnos para entender mejor de que se trata:

```
bzr help commands
```

Despliega una lista completa de los comandos internos y su descripción.

```
bzr version
```

Despliega información sobre la versión de Bazaar.

```
bzr help comando-interno
```

Despliega información de ayuda sobre el comando indicado.

Presentarse ante Bazaar

Cada vez que enviemos cambios al repositorio, Bazaar tendrá que identificarnos a fin de poder ofrecer la información correcta sobre quien ha efectuado alguna revisión. Para ello, el primer paso, es presentarnos:

```
bzr whoami "Juan Perez <juanperez@dominio.ext>"
```

Iniciar un nuevo proyecto

El siguiente paso, será crear un nuevo proyecto. Para ello, haremos lo siguiente:

Primero inicializamos el repositorio central en el ordenador destinado a tal fin:

```
__eugenia_1978_esAR__@mydream:/srv/repos$ bzr init-repo app-curso-xp
Shared repository with trees (format: 2a)
Location:
  shared repository: app-curso-xp
```

Luego, allí mismo, creamos nuestro `branch` (al que llamaremos trunk):

```
__eugenia_1978_esAR__@mydream:/srv/repos$ bzr init app-curso-xp/trunk
Created a repository tree (format: 2a)
Using shared repository: /srv/repos/app-curso-xp/
```

Clonar el repositorio central: crear los repositorios locales

Es hora de que cada miembro del equipo, se traiga el branch desde el repositorio central. Esta actividad deberá realizarse en cada una de las máquinas de cada uno de los miembros del equipo (previamente, deberán presentarse ante Bazaar mediante bzr whoami).

Para clonar el repositorio, cada uno de los miembros del equipo, hará los siguiente:

```
eugenia@cocochito:~/example$ bzr branch
bzr+ssh://user@x.x.x.x/srv/repos/app-curso-xp/trunk
user@x.x.x.x's password:
Branched 0 revision(s).
```

Nótese que la ruta hacia el branch debe formarse por:

```
protocolo://usuario@host/ruta/al/branch/central
```

> *Nótese que **en el caso del protocolo SSH**, debe anteponerse el prefijo **`bzr+`***

Si listamos el directorio donde clonamos el repo, podremos ver que ya tenemos nuestro branch local:

```
eugenia@cocochito:~/example$ ls -lha
total 12K
drwxr-xr-x  3 eugenia eugenia 4,0K 2012-04-21 19:59 .
drwxr-xr-x 65 eugenia eugenia 4,0K 2012-04-21 20:04 ..
drwxr-xr-x  3 eugenia eugenia 4,0K 2012-04-21 19:59 trunk
```

Podremos comprobar también, que efectivamente estamos "conectando" nuestro repo local con el central:

```
eugenia@cocochito:~/example$ cd trunk/
eugenia@cocochito:~/example/trunk$ bzr info
Standalone tree (format: 2a)
Location:
  branch root: .

Related branches:
  parent branch: bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-
curso-xp/trunk/
```

Nociones básicas para integrar código de forma continua

Crearemos ahora nuestro primer archivo (de prueba):

```
eugenia@cocochito:~/example/trunk$ echo "Hola Mundo" > prueba.txt
```

Y verificaremos cual es el estado actual de nuestro repo:

```
eugenia@cocochito:~/example/trunk$ bzr st
unknown:
  prueba.txt
```

puede utilizarse tambien, `bzr status`

Nos indica que hay un archivo desconocido. ¡Es el archivo que acabamos de crear! No te preocupes. El siguiente paso tras crear nuevos archivos y/o directorios, es avisarle a Bazaar, para lo cual, hay que agregar el archivo o directorio:

```
eugenia@cocochito:~/example/trunk$ bzr add prueba.txt
adding prueba.txt
```

Volvemos a comprobar el estado:

```
eugenia@cocochito:~/example/trunk$ bzr st
added:
  prueba.txt
```

Bazaar no está informando que se ha realizado un cambio en el repositorio. Y **todo cambio, debe ser informado Bazaar** con un mensaje que lo describa:

```
eugenia@cocochito:~/example/trunk$ bzr ci -m "Agregado archivo de prueba"
Committing to: /home/eugenia/example/trunk/
added prueba.txt
Committed revision 1.
```

¡Ya tenemos la primera revisión!!! Pero el *commit*, fue realizado localmente. Nuestro repo central, aún no se ha enterado y tenemos que **enviar los cambios al repositorio central**:

```
eugenia@cocochito:~/example/trunk$ bzr push
bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/
__eugenia_1978_esAR__@66.228.52.93's password:
This transport does not update the working tree of:
bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/.
See 'bzr help working-trees' for more information.
Pushed up to revision 1.
```

Pero ¿por qué nos dice que el árbol de trabajo no se ha actualizado? Debemos actualizarlo en el repositorio central:

```
__eugenia_1978_esAR__@mydream:/srv/repos/app-curso-xp/trunk$ bzr st
working tree is out of date, run 'bzr update'
__eugenia_1978_esAR__@mydream:/srv/repos/app-curso-xp/trunk$ bzr update
+N  prueba.txt
All changes applied successfully.
Updated to revision 1 of branch /srv/repos/app-curso-xp/trunk
```

Diariamente, todos los miembros del equipo, deberán **traer los cambios** desde el repo central:

```
eugenia@cocochito:~/example/trunk$ bzr pull
bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/
__eugenia_1978_esAR__@66.228.52.93's password:
No revisions to pull.
```

> ***TIPS:***
> ***Avisa de los cambios a Bazaar:*** *Cada vez que realices cambios, haz un commit local (bzr ci -m "mensaje").*
>
> ***Envía los cambios:*** *Envía los cambios al repositorio central diariamente, solo después de haber corrido todos los test verificando que pasaran (bzr push)*
>
> ***Trae los cambios:*** *Diariamente, antes de comenzar a trabajar en el repo local, trae del repositorio central, todos los cambios que tus compañeros de equipo hallan enviado el día anterior (bzr pull)*

Guardando el path del repo central

Puede ser extremadamente molesto, tener que estar indicando la dirección del repo central con cada pull y push que hagamos. Podemos evitar esto, editando el archivo de configuración de nuestro repo local:

```
eugenia@cocochito:~/example/trunk$ vim .bzr/branch/branch.conf
```

y configuramos las variables pull_location y push_location (modificar la ruta por la que corresponda):

```
push_location = bzr+ssh://user@host/srv/repos/app-curso-xp/trunk/
pull_location = bzr+ssh://user@host/srv/repos/app-curso-xp/trunk/
```

Guardamos los cambios y ya podremos hacer pull y push, solo con el comando respectivo:

```
eugenia@cocochito:~/example/trunk$ bzr pull
Using saved parent location:
bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/
__eugenia_1978_esAR__@66.228.52.93's password:
No revisions to pull.
eugenia@cocochito:~/example/trunk$ bzr push
Using saved push location:
bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/
__eugenia_1978_esAR__@66.228.52.93's password:
This transport does not update the working tree of:
bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/.
See 'bzr help working-trees' for more information.
No new revisions to push.
```

Integración continua avanzada con Bazaar

A la hora de trabajar con un SCV, aparece una larga lista de cuestiones, que ameritan especial cuidado. Por ejemplo ¿qué sucede si por accidente, elimino un archivo en mi repositorio local? ¿Cómo lo recupero? O ¿qué sucede si modifiqué un archivo en mi repo local, sobre una versión distinta a la que está en el repo central? Y estas, son solo dos de las decenas de preguntas que nos pueden surgir.

Veremos aquí, como solucionar cada una de las problemáticas más frecuentes que se presentan en la vida diaria de la integración continua.

Problema	Descripción	solución
Ignorar archivo	Se desea evitar que un archivo o directorio sea enviado al repo central	**`bzr ignore archivo`** `bzr ignore archivo.txt`
Recuperar archivo	Recuperar una versión anterior de un archivo o directorio	**`bzr revert nro_revision`** `bzr revert 2`
Dejar de lado un archivo temporalmente	Se desea evitar temporalmente, enviar los cambios de un archivo, al repo central	**`bzr shelve archivo`** `bzr shelve archivo.txt`
Recuperar cambios	Se desea recuperar los cambios de un archivo, previamente salvado mediante shelve	**`bzr unshelve archivo`** `bzr unshelve archivo.txt`
Encontradas versiones diferentes de un mismo archivo	Al traer los cambios desde el repo central, las modificaciones hechas sobre un archivo, interfieren con las efectuadas localmente al mismo archivo	**`bzr merge -i archivo`** `la opción -i es opcional y permite seleccionar los cambios de forma interactiva`
Merge no resolvió el conflicto	Tras combinar diferentes versiones, Bazaar informa de conflictos existentes	**`Bzr conflict-diff archivo`** `(generalmente requerirá solucionar el conflicto de forma manual. Conflict-diff mostrará las diferencias encontradas que no pudieron resolverse con el merge)`

Nótese que toda vez que se indica la palabra “archivo”, se hace referencia no solo a archivos sino también a directorios.

Resumen de comandos de uso frecuente

Comando	Descripción
`add`	Agregar archivo o directorio
`check`	Verifica la consistencia del árbol de trabajo
`ci`	Envía mensajes al repo informando cambios (debe ejecutarse siempre después de realizar cambios y antes de enviar cambios al repo central con push)
`conflict-diff`	Muestra las diferencias que generan conflictos en un archivo
`deleted`	Muestra la lista de archivos eliminados
`ignore`	Ignora un archivo
`ignored`	Muestra la lista de archivos ignorados

info	Muestra información sobre el árbol de trabajo
log	Muestra los cambios históricos (revisiones) de un branch
merge	Combina los cambios centrales con los locales en un archivo (tras el merge, siempre se debe hacer commit – generalmente: bzr ci -m “merge”)
mv	Mueve o renombra un archivo o directorio
pull	Enviar cambios al repo central
push	Traer cambios desde el repo central
remerge	Elimina un merge anterior
remove	Elimina archivos o directorios
renames	Muestra la lista de archivos que han sido renombrados
resolve	Marca resuleto un conflicto
revert	Recupera un archivo a una revisión anterior
revno	Muestra el número de la revisión actual
shelve	Deja temporalmente a un lado, un determinado archivo que no se desee enviar aún al repo central
st	Muestra el estado del repo (localmente) – cuando no muestra nada, significa que todo está en orden y se pueden enviar cambios al repo central sin conflictos. Cualquier otro mensaje, requerirá intervenir para resolverlo, antes de enviar cualquier cambio al repo central.
tag	Taguea (etiqueta) una revisión (la remueve o modifica según las opciones pasadas como argumentos)
tags	Retorna la lista completa de tags
uncommit	Revierte un commit efectuado anteriormente
unshelve	Recupera los cambios de un archivo salvado mediante shelve
update	Actualiza el árbol de trabajo trayendo el último commit enviado
whoami	Presentarse ante Bazaar (u obtener los datos de presentación aportados con anterioridad)

Resumen para uso diario de Bazaar

Esta, es una breve síntesis, de **los pasos diarios a seguir**, cuando se trabaja con repositorios.

> *Al comienzo, podría ser muy útil, imprimir esta planilla e ir marcando los casilleros vacíos a medida que se completa cada paso.*

¿Cuándo?	¿Qué?	¿Cómo?	
Al comenzar el día	Actualizar el repo local	**`bzr pull`**	
	Mergear archivos con diferencias (si aplica)	**`bzr merge`** **`bzr ci -m "merge"`**	
	Resolver conflictos en archivos, no resueltos con un merge: `nombre_archivo` (archivo conflictivo mergeado localmente) `nombre_archivo.`**`BASE`** (original del repo central) `nombre_archivo.`**`OTHER`** (archivo con las modificaciones hechas por otro miembro del equipo) `nombre_archivo.`**`THIS`** (archivo modificado por uno mismo, localmente)	`visualizar los archivos en conflicto y corregirlos.`	
	Eliminar el conflicto	`bzr resolve`	
Luego de actualizar el repo	Trabajar libremente sobre tus archivos		

local			
Al concluir un task (tarea)	Verificar el estado de los cambios	**`bzr st`**	
	Agregar archivos nuevos (reportados como unknow)	`bzr add nombre_archivo`	
	Eliminar archivos obsoletos	`bzr remove nombre_archivo`	
	Ignorar archivos que no deban actualizarse en el repo central	`bzr ignore nombre_archivo`	
	Comitear los cambios	**`bzr ci -m`** `"breve descripción de la tarea terminada"`	

	Correr todos los test y verificar que pasen	`php -f test`
		
		`python -m unittest discover -v`	

	Enviar cambios al repo central	**`bzr push`**	
Al finalizar el día	Actualizar el repo CENTRAL	**`bzr update`**	

Los pasos que se encuentran sombreados en tono más claro, son opcionales, y dependerán de cada caso en particular.

Ver el historial de revisiones: **`bzr log`** | Obtener el nro. de revisión actual: **`bzr revno`**

Refactoring

Veremos aquí, la quinta práctica sugerida por eXtreme Programming, con mayores detalles y algunos ejemplos.

Refactoring, como comentamos anteriormente, es una técnica que consiste en mejorar el código fuente de una aplicación (limpiarlo), sin que dichas modificaciones, afecten el comportamiento externo del sistema.

Existen diferentes tipos de refactorizaciones que pueden ser necesarias implementar al código de nuestra aplicación. Cada tipo, representa una ténica diferente de refactorización. Por ejemplo, eliminar código redundante, requiere de una técnica diferente a dividir los algoritmos de un método para crear métodos derivados.

Sin embargo, hablar de técnicas de refactorización puede resultar confuso, ya que **la refactorización en sí misma es una técnica, que ofrece diferentes soluciones a cada tipo de problema**. Por lo tanto, es preferible pensar la refactorización como una única técnica que propone diferentes soluciones a cada tipo de problema.

El problema

En principio, habría que diferenciar el término "problema" de la palabra "error", para no generar confusiones. El error en sí, es una falla en el código fuente, que impide el correcto comportamiento del sistema. Mientras que el problema, puede

definirse como "algo que huele mal en el código fuente"[12] pero sin embargo, no impide el correcto funcionamiento de la aplicación.

Los problemas que se pueden presentar en el código fuente de una aplicación, dependen de muchísimos factores, que en gran parte de los casos, encuentran una relación directa con el paradigma de programación empleado así como en el lenguaje que se utilice.

Si se intentara abarcar todos los problemas posibles, la lista podría tornarse infinita, tediosa y hasta inutil o muy confusa. Es por ello, que solo abarcaremos los problemas más frecuentes, que puedan considerarse generales, independientes al lenguaje pero más cercanos al paradigma de la programación orientada a objetos.

La solución

Indefectiblemente, la solución a cada problema será la refactorización y auqnue resulte redundante, la solución, dependerá de cada problema. Sin embargo, como regla general, la solución deberá comenzar por identificar el momento en el cual llevarla a cabo.

12 Kent Beck, uno de los creadores de eXtreme Programming, es quien introdujo el término "bad smells" (malos olores) para referirse de manera global, a aquellas expresiones y algoritmos poco claros que generan confusión en el código fuente de un sistema, tornándolo más complejo de lo que debería ser.

Cuándo y cómo tomar la desición de refactorizar

> ***Tres strikes y ¡Refactoriza!***
> *En el mundo de la refactorización, haciendo una anología con el béisbol, suele utilizarse la regla "Tres Strike[13] y ¡refactoriza!". Esto puede descrbirse análogamente como:* ***"la primera vez que hagas algo, solo hazlo. La segunda vez que hagas algo similar, notarás que estás duplicando código, pero lo harás de todas formas. La tercera vez que te enfrentes al mismo caso, refactoriza".***

Cuando se está programando una aplicación con TDD, como hemos visto anteriormente, el proceso de desarrollo se está dividiendo en dos acciones concretas: programar y refactorizar. Esto es, a medida que vamos creando nuevos métodos, vamos refactorizando el código para eliminar redundancias y en definitiva, hacer el código -del test- más legible y así obtener un mejor rendimiento. Pero no estamos refactorizando el SUT constantemente, puesto que éste, tiene un momento y lugar para ser refactorizado.

La refactorización del SUT, implica que lo primero que debemos hacer, es cumplir el objetivo (programar aquello que se necesita) y luego refactorizar el código del SUT, cada vez que:

- Se agregue un nuevo método

13 En el béisbol, un strike es una anotación negativa para el bateador ofensivo, cuando la pelota no es lanzada hacia el diamante. Al tercer strike anotado, termina el turno del bateador.

- Se corrija un bug
- Se haga una revisión de código

Pero siempre, respetando la regla de "los tres strikes". Una vez identificado el momento, solo será cuestión de identificar el problema a fin de poder elegir la solución indicada.

Una solución a cada problema

Como comentamos anteriormente, no haremos una extensa lista de problemas, sino que nos centraremos en problemas generales. Muchas de las soluciones sugeridas en este capítulo, han sido extraídas de *SourceMaking.com*[14], sitio donde se puede encontrar una completa clasificación de problemas[15] y sus respectivas soluciones[16]. Como hemos hecho a lo largo del curso, iremos de lo general a lo particular y de lo particular al detalle.

Variables de uso temporal mal implementadas

En principio, definiremos a las variables de uso temporal, como aquellas variables que son asignadas en el ámbito local de un método de clase y son necesarias temporalmente, solo en ese método, sin ser llamadas o requeridas por otros métodos.

14 http://sourcemaking.com/refactoring. Nótese que algunas de las técnicas expuestas en el sitio Web referido, no se mencionan en este curso, por considerarlas poco apropiadas. Esto es debido a que algunas prácticas son más específicas de lenguajes como Java, mientras que a otras, las considero contrarias a las buenas prácticas de la programación orientada a objetos y por lo tanto, contraproducentes.

15 "Bad Smells in Code" http://sourcemaking.com/refactoring/bad-smells-in-code

16 Diferentes técnicas de refactorización: http://sourcemaking.com/refactoring

Generalmente representan un problema en los siguientes casos:

1) Variables de uso temporal que definen una acción concreta:

```
$var = ($a * $b ) / (int)$c;
```

En el ejemplo anterior, vemos una variable de uso temporal, que define una acción concreta: dividir el producto de dos factores. Esto representa un problema, ya que las acciones son responsabilidad de los métodos y no de las variables. En estos casos, la solución, es **transferir la responsabilidad de la acción a un método**:

```
$var = dividir_producto($a, $b, $c);

function dividir_producto($a, $b, $c) {
    return ($a * $b ) / (int)$c;
}
```

> *Nótese que variables de uso temporal que definen un valor directo:* `$var = 15;` *o por el retorno de la llamada a una función:* `$var = strlen($variable);` *no necesitan transferir su responsabilidad a otro método.*

2) Variables de uso temporal son requeridas por más de un método:

```
function metodo_a() {
```

```
    $a = 15;
    $b = 100;
    $c = 2;
    $var = self::dividir_producto($a, $b, $c);
    // continuar...
}

private static function dividir_producto($a, $b, $c) {
    return ($a * $b ) / $c;
}
```

En el ejemplo, anterior, las variables temporales $a, $b y $c, son requeridas por dos métodos y se están definiendo como tales en un método, requiriendo ser pasadas como parámetros. Aquí, la solución, será **convertir las variables temporales, en propiedades de clase**:

```
function metodo_a() {
    self::$a = 15;
    self::$b = 100;
    self::$c = 2;
    $var = self::dividir_producto();
    // continuar...
}

private static function dividir_producto() {
    return (self::$a * self::$b ) / self::$c;
}
```

3) Variables de uso temporal que reasignan parámetros:

```
function foo($a) {
    $a = htmlentities($a);
    // continuar ...
}
```

En casos como éste, la confusión puede ser grande: un parámetro es un parámetro y una variable temporal, una variable temporal. Es entonces, cuando **variables temporales no deben tener el mismo nombre que los parámetros**:

```
function foo($a) {
    $b = htmlentities($a);
    // continuar ...
}
```

Métodos que reciben parámetros

Aquí debe hacerse una notable distinción entre parámetros, variables de uso temporal y propiedades de clase. Y esta distinción, está dada por la finalidad de cada una:

- Las **variables de uso temporal**, como hemos visto antes, están destinadas a definir un valor concreto al cual se hará referencia solo en el ámbito donde se haya definido.
- Las **propiedades de clase**, son características inherentes al objeto a las cuales se hará referencia desde diversos ámbitos.
- Y finalmente, los **parámetros**, serán valores adicionales, que no pueden ser considerados propiedades del objeto pero que sin embargo, son requeridos para que una acción, modifique las propiedades de un objeto.

```
Class Usuario {

    function validar_usuario($username, $pass) {
        if($username == 'pepe' && $pass == '123') {
            return True;
        }
    }

}
```

En el ejemplo anterior, claramente los parámetros $username y $pass, deberían ser propiedades del objeto Usuario puesto que

son características intrínsecas al objeto. Como regla general, los parámetros deben ser evitados toda vez que sea posible, reemplazándolos por propiedades de clase:

```
Class Usuario {

    function validar_usuario() {
        if($this->username == 'pepe' && $this->pass == '123') {
            return True;
        }
    }

}
```

Expresiones extensas

Muchas veces, podremos encontrarnos con expresiones que debido a su extensión, se hacen difíciles de leer y cuando no, confusas:

```
return ((in_array('abc', $array) || in_array('bcd', $array)) &&
(in_array('cde', $array) || in_array('def', $array))) ? 'OK' : 'ERROR';
```

Cuando estamos en presencia de expresiones tan extensas, lo mejor es -aquí sí- utilizar variables de uso temporal para simplificar dichas expresiones:

```
$a = in_array('abc', $array);
$b = in_array('bcd', $array);
$c = in_array('cde', $array);
$d = in_array('def', $array);

return (($a || $b) && ($c || $d)) ? 'OK' : 'ERROR';
```

Métodos extensos

No solo una expresión puede ser extensa. Muchas veces, nos encontraremos con métodos con extensos algoritmos que realizan varias acciones:

```
function renderizar_plantilla($data=array(), $pattern, $template) {
    $ini_pattern = "[[INI-PATTERN-{$pattern}]]";
    $end_pattern = "[[END-PATTERN-{$pattern}]]";
    $plantilla = file_get_contents($template);
    $pos_ini = strpos($ini_pattern);
    $pos_fin = strpos($end_pattern);
    $longitud_cadena = $pos_fin - $pos_ini;
    $cadena = substr($plantilla, $pos_ini, $longitud_cadena);
    $reemplazos = '';
    foreach($data as $identificador=>$valor) {
        $reemplazos .= str_replace("[{$identificador}]", $valor, $cadena);
    }
    $resultado = str_replace($cadena, '[[NUEVO-CONTENIDO]]', $plantilla);
    $resultado = str_replace('[[NUEVO-CONTENIDO]]', $reemplazos, $plantilla);
    return $resultado;
}
```

Cuando existen métodos tan extensos, probablemente, la solución consista en la combinación de diversas técnicas, que van desde **agrupar expresiones en una misma línea** hasta evitar la asignación de variables temporales (como vimos al comienzo) y **extraer código llevándolo a diferentes métodos**:

```
function renderizar_plantilla($data=array()) {
    self::set_patterns();
    self::set_contenido_plantilla();
    self::get_pattern();
    self::reemplazar_datos($data);
    return str_replace('[[NEW]]', self::$reemplazos,
self::set_new_pattern());
}

// extracción de código para crear nuevo método
// y sustitución de parámetros por propiedades de clase
static function set_patterns() {
    self::$ini_pattern = "[[INI-PATTERN-{self::$pattern}]]";
    self::$end_pattern = "[[END-PATTERN-{self::$pattern}]]";
}

// extracción de código para crear nuevo método
// y sustitución de parámetros por propiedades de clase
static function set_contenido_plantilla() {
    self::$contenido = file_get_contents(self::$template);
}

// extracción de código para crear nuevo método
// sustitución de parámetros por propiedades de clase
// y sustitución de expresiones en línea
static function get_pattern() {
    self::$cadena = substr(self::$contenido, strpos(self::$ini_pattern),
```

```
                            (self::$pos_fin – self::$pos_ini));
}

// extracción de código para crear nuevo método
// y sustitución de parámetros por propiedades de clase
static function reemplazar_datos($data=array()) {
    self::$reemplazos = '';
    foreach($data as $identificador=>$valor) {
        self::$reemplazos .= str_replace("[{$identificador}]", $valor, self::
$cadena);
    }
}

// extracción de código para crear nuevo método
static function set_new_pattern() {
    return str_replace(self::$cadena, '[[NEW]]', self::$contenido);
}
```

Código duplicado en una misma clase

Es frecuente -y de lo más común-, que las mismas expresiones, comiencen a duplicarse en diferentes métodos de una misma clase:

```
function metodo_1() {
    $a = strip_tags(self::$propiedad);
    $a = htmlentities(self::propiedad);
    return self::metodo_a() . self::$propiedad;
}

function metodo_2() {
    $a = strip_tags(self::$propiedad);
    $a = htmlentities(self::propiedad);
    return self::$propiedad . self::metodo_b() . self::metodo_c();
}
```

Las expresiones duplicadas en el código de los diferentes métodos de una misma clase, se solucionan **extrayendo el código duplicado de los métodos, y colocándolo en un nuevo método de clase**:

```
function metodo_1() {
    self::metodo_3();
    return self::metodo_a() . self::$propiedad;
}
```

```
function metodo_2() {
    self::metodo_3();
    return self::$propiedad . self::metodo_b() . self::metodo_c();
}

static function metodo_3() {
    self::$propiedad = strip_tags(self::$propiedad);
    self::$propiedad = htmlentities(self::propiedad);
}
```

Código duplicado en varias clases con la misma herencia

El caso anterior puede darse también, cuando el código se encuentra duplicado en diferentes métodos de clases con la misma herencia:

```
class B extends A {

    function metodo_1() {
        $a = strip_tags(self::$propiedad);
        $a = htmlentities(self::propiedad);
        return self::metodo_a() . self::$propiedad;
    }
}


class C extends A {

    function metodo_2() {
        $a = strip_tags(self::$propiedad);
        $a = htmlentities(self::propiedad);
        return self::$propiedad . self::metodo_b() . self::metodo_c();
    }
}
```

En estos casos, en los cuáles existen dos o más clases que heredan de la misma clase, se extrae el código duplicado en los métodos de las clases hijas, y con éste, se crea un nuevo método de en la clase madre:

```
class A {

    static function metodo_3() {
        self::$propiedad = strip_tags(self::$propiedad);
```

```
        self::$propiedad = htmlentities(self::propiedad);
    }

}


class B extends A {

    function metodo_1() {
        self::metodo_3();
        return self::metodo_a() . self::$propiedad;
    }
}


class C extends A {

    function metodo_2() {
        self::metodo_3();
        return self::$propiedad . self::metodo_b() . self::metodo_c();
    }
}
```

Código duplicado en varias clases sin la misma herencia

Como era de esperarse, el código también podrá aparecer duplicado en diferentes clases pero que no tienen la misma herencia:

```
class B {

    function metodo_1() {
        $a = strip_tags(self::$propiedad);
        $a = htmlentities(self::propiedad);
        return self::metodo_a() . self::$propiedad;
    }
}


class C {

    function metodo_2() {
        $a = strip_tags(self::$propiedad);
        $a = htmlentities(self::propiedad);
        return self::$propiedad . self::metodo_b() . self::metodo_c();
    }
}
```

En estos casos, la solución es extraer el código duplicado, crear una nueva clase y con el código extraído, crear un método para esta nueva clase que podrá ser heredada por las anteriores o simplemente, instanciada:

```
class A {

    static function metodo_3($parametro) {
        return htmlentities(strip_tags($parametro));
    }

}


class B {

    function metodo_1() {
        return self::metodo_a() . A::metodo_3(self::$propiedad);
    }
}


class C {

    function metodo_2() {
        return A::metodo_3(self::$propiedad) . self::metodo_b() .
self::metodo_c();
    }
}
```

Combinando Scrum y eXtreme Programming

¿Te imaginas convertirte en un(a) programador(a) eXtremo/a que organiza su trabajo con Scrum o viceversa? Sin dudas, la combinación Scrum + XP (o XP + Scrum), es perfecta y compatibilizar ambas metodologías, es sumamente simple. Veremos aquí, algunas sugerencias, para obtener el máximo provecho de ambas.

Compatibilizando ambas metodologías

Para compatibilizar ambas metodologías, es necesario hacer un análisis comparativo de cada propuesta para obtener así, el grado de compatibilidad entre ambas. Haremos un repaso por cada uno de los valores y prácticas técnicas de eXtreme Programming, analizándolos de forma separada a fin de deducir cuán compatibles (o no) resultan con Scrum.

Compatibilidad de Scrum con los valores de XP

La siguiente tabla muestra una comparación entre los valores de XP y su correlación con Scrum, obteniendo el grado de compatibilidad en una escala de 1 a 3, siendo 1, poco compatible y requierirá decidir si se debe implementar o no; 2 algo compatible pero debe adaptarse y 3, absolutamente compatible.

VALOR	XP	SCRUM	COMPATIBILIDAD
Comunicación	Los impedimentos son resueltos entre los miembros del equipo y cliente	Los impedimentos son relevados por el equipo al Scrum Master y éste se encarga de resolverlos (incluyendo o no, al dueño de producto)	1/3
Simplicidad	Se pretende desarrollar solo aquellos aspectos mínimos necesarios para cumplir con los objetivos	Se priorizan las Historias de Usuario, desarrollando solo aquellas más prioritarias	3/3
Retroalimentación	Pretende lograr un feedback continuo con el cliente, a través de las entregas tempranas	Se planifica y revisa en forma conjunta entre el equipo y dueño de producto (cliente) en períodos regulares. Si surgen impedimentos, son relevados al Scrum Master y éste se encarga de resolverlos	2/3
Respeto	Se respeta la idoneidad del cliente con respecto a las decisiones que aportan valor al negocio y la del equipo con respecto al cómo esas decisiones deben ser desarrolladas	Solo el dueño de producto (cliente) puede decidir que Historias de Usuario serán desarrolladas en el Sprint y únicamente el equipo puede definir la forma en la cuál serán desarrolladas	3/3
Coraje	El equipo se compromete a estimar con sinceridad y	Las estimaciones se realizan frente al dueño de producto y el equipo se	3/3

	comentar con honestidad el avance del proyecto	compromete a relevar los impedimentos al Scrum Master y mostrar el avance del proyecto en el tablero	

Visualizando la tabla anterior, nos encontramos con que la **comunicación**, es el valor que menor compatibilidad posee. Esto significa, que al momento de unificar metodologías, **se deberá decidir la forma en la cual, los impedimentos serán afrontados:**

- Estilo XP → serán resueltos entre los miembros del equipo
- Estilo Scrum → serán resueltos por el Scrum Master

Luego, la **retroalimentación**, requerirá ser adaptada ya que mayormente, es compatible entre ambas metodologías. La adaptación de este valor, **dependerá de forma directa de lo que se decida con respecto a la comunicación**.

Compatibilidad con las prácticas técnicas de XP

Aquí no será necesario hacer una análisis exhaustivo de cada práctica, ya que Scrum, no propone ninguna de forma directa. Solo nos concentraremos en aquellas prácticas que puedan llegar a generar ciertas dudas. Utilizaremos el mismo método de análisis de compatibilidad, empleado en el punto anterior.

PRÁCTICA	XP	SCRUM	COMPATIBILIDAD
Cliente in-situ	Lo integra al desarrollo de forma directa (el cliente está físicamente presente durante todo el proceso de desarrollo)	Lo integra de forma indirecta, en las ceremonias de planificación y revisión y opcionalmente, en las de revisión y retrospectiva.	2/3
Entregas cortas	Cada 1 o 2 semanas	Cada 2 o 4 semanas	2/3
Testing	El cliente define los test de aceptación	El dueño de producto define los criterios de aceptación	2/3
Juego de Planificación	No propone una técnica de estimación de forma explicíta	Indefectiblemente propone el uso de al menos una técnica de estimación (Planning Pocker, T-Shirt Sizing, Columnas o una combinación de varias)	3/3

Como podemos ver, son 3 las prácticas técnicas que deberán ser adaptadas. En el caso de cliente in-situ, se deberá adaptar esta práctica, analizando si:

- Sumar al cliente al Scrum diario como en XP
- Contar con el cliente solo en las ceremonias de planificación y revisión como en Scrum

No obstante, existe una alternativa intermedia, que es incorporar al Dueño de Producto (cliente) en las ceremonias de revisión y retrospectiva, lo cual es permitido por Scrum y se asemeja más a lo propuesto por XP.

Con respecto a las **entregas cortas**, son perfectamente compatibles entre ambas metodologías. Establecer Sprints de 2 semanas, sería el equilibrio ideal.

En cuanto al **testing**, la única diferencia encontrada, es que en XP, el cliente define los Test de Aceptación y en Scrum, solo indica los criterios de aceptación. La forma de compatibilizar ambas metodologías, es simplemente, sumar al cliente (dueño de producto), la facultad de definir Test de Aceptación para cada uno de los Criterios de Aceptación definidos en una Historia de Usuario.

Combinando ambas metodologías

¿Scrum con XP o XP con Scrum? Parece un simple juego de palabras, pero no lo es. En este caso, "el orden de los factores, altera el producto". Por ello, la forma en la cual se combinen ambas metodologías, **dependerá de cuál de las dos, se utilice como base**.

Combinar Scrum con eXtreme Programming

Sin lugar a duda, considero que utilizar Scrum con XP, **es la combinación que mejores resultados genera**.

En este caso, se utiliza Scrum como base de toda la organización y sin modificaciones de ningún tipo:

- Se respetan sus ceremonias, roles y artefactos

- Se organiza el trabajo por Sprints, priorizando Historias de Usuario, estimando esfuerzos, planificándolas por Sprint y dividiéndolas en tareas
- Se respeta la transparencia de todo el proceso, mediante el uso de tableros físicos y diagramas de burndown que permitan conocer el esfuerzo restante por Sprint

Es decir, **se respeta Scrum en su totalidad sin alteraraciones de ningún tipo**. En cada Sprint, **el trabajo es complementado por las prácticas técnicas de eXtreme Programming que no requieren ser adaptadas** y hacen referencia exclusiva a cuestiones de índole tecnológica:

- Práctica #4: KISS (Keep it Simple, Silly!)
- Práctica #5: Refactoring
- Práctica #6: Pair Programming
- Práctica #8: Testing / TDD
- Práctica #9: Código estándar
- Práctica #10: Propiedad colectiva
- Práctica #11: Integración contínua

Combinar Scrum con eXtreme Programming

En este caso, la base del proyecto será XP en su totalidad, respetando sus cinco valores y las 12 prácticas técnicas sin alteraciones de ningún tipo.

Esta base, será complementada, por los artefactos y Ceremonias de Scrum, adaptados a los valores de XP:

- Organización por Sprints
- Planificación, revisión y retrospectiva (queda excluido el Scrum diario)
- Tablero físico, diagrama de burndown
- Estimación por Planning Poker, T-Shirt Sizing, Columnas o combinaciones de éstas

Los roles, solo quedarán repartidos entre Cliente (Dueño de Producto para Scrum) y Equipo. La figura del Scrum Master queda exceptuada en esta combinación (marca una diferencia radical con la combinación Scrum + XP).

Material de lectura complementario

Puedes encontrar más información sobre cómo compatibilizar estas dos metodologías, en el libro **Scrum y XP desde las Trincheras** de **Henrik Kniberg**, el cual se encuentra disponible (en español) en **InfoQ**:

http://www.infoq.com/minibooks/scrum-xp-from-the-trenches

Kanban: la metodología ágil que menor resistencia ofrece

A esta altura, ya estamos sumamente familiarizados con el agilismo. Sabemos qué es y cómo llevarlo a la práctica, con dos de las metodologías ágiles por excelencia: Scrum, la elegida por la mayoría y XP, la que sugiere prácticas técnicas que incluso, pueden implementarse en metodologías más tradicionales.

Sin embargo, existen muchísimas otras metodologías ágiles y si bien, este curso solo se enfoca en Scrum y eXtreme Programming, existe una metodología, que por una particularidad, no podemos dejar de conocer. Esta metodología, es **Kanaban**, y su particularidad, es que debido a su simplicidad -y facilidad para camuflarse con las metodologías más tradicionales- **estadísticamente, es la que menor resistencia al cambio ofrece a la hora de pasar de una metodología predictiva a una ágil**.

De TOYOTA™ al Desarrollo de Software

Kanban es un término Japonés el cual puede traducirse como "insignia visual" (Kan: visual, ban: sello o insignia).

De todas las metodologías ágiles,

Kanban es la más nueva, ya que comenzó a implementarse, prácticamente una década posterior a otros enfoques adaptativos.

Descubierta por el Ing. Taiichi Ohno de la empresa automotriz Toyota™, Kanban ha llegado a la industria del Software recientemente en el año 2004, de la mano de David Anderson en la Unidad de Negocio XIT de Microsoft™, arrojando resultados muy alentadores.

> *Según David Anderson, Kanban permitió en Microsoft, producir cambios incrementales en la forma de trabajo* ***con una mínima resistencia al cambio****, generando un* ***incremento de productividad superior al 300%*** *y promedió una* ***reducción del ciclo de desarrollo en un 90%.***

Básicamente, Kanban se apoya en una producción a demanda: se produce solo lo necesario, de manera tal que el ritmo de la demanda sea quien controle al ritmo de la producción (se necesitan N partes, entonces solo se producen N partes), limitando el trabajo en curso a una cantidad predefinida.

> [...] Un sistema Kanban es un sistema o proceso diseñado para disparar trabajo cuando hay capacidad para procesarlo [...] (2005, David Anderson, Microsoft™)

En el sistema Kanban, este disparador es representado por tarjetas, que se distribuyen en cantidades limitadas (esto es, lo que limitará el trabajo en curso). Cada item de trabajo, se

acompaña de una de estas tarjetas, por lo cual, un nuevo ítem solo podrá iniciarse si se dispone de una tarjeta Kanban libre. Cuando no hay más tarjetas libres, no se pueden iniciar nuevos trabajos. Y cuando un item es concluido, una nueva tarjeta se libera, permitiendo el comienzo de un nuevo item de trabajo.

Las tres reglas de Kanban

Partiendo de las bases anteriores, en el desarrollo de Software, se "virtualiza" esta esencia, llevándola a la práctica mediante la implementación de tres reglas:

1. Mostrar el proceso
2. Limitar el trabajo en curso (WIP)
3. Optimizar el flujo

Mostrar el proceso

Esta regla busca hacer visibles los items de trabajo permitiendo conocer de manera explícita el proceso trabajo actual, así como los impedimentos que vayan surgiendo. Dicha visualización, se realiza a través de tableros físicos, al igual que en Scrum, solo que con diferentes columnas (que veremos más adelante).

En el caso de Kanban, la implementación de tableros físicos, es fundamental a la hora de comprender la capacidad de proceso del equipo de desarrollo.

Para cumplir con la regla de mostrar el proceso, será necesario definir con precisión:

- Punto de inicio y finalización de la visibilidad del proceso

  Limitar la visualización a las actividades del proceso en las cuales el equipo de desarrollo tiene mayor influencia para efectuar cambios. A partir de estos límites, se negocia la forma de interacción entre procesos anteriores y posteriores.

- Tipos de ítems de trabajo

  Esto hace referencia a los tipos de items solicitados en el proceso de trabajo (por ejemplo, bugs, refactoring, actualizaciones, etc.)

- Diseño de las tarjetas que acompañarán a cada ítem

  La información contenida en cada tarjeta, debe ser lo suficientemente precisa, como para permitir su evaluación y la toma de decisiones a cualquier parte involucrada en el proyecto.

  Generalmente, la información mínima que debe contener cada item es: titulo, ID, fecha de entrada, tipo de ítem, persona asignada, prioridad y deadline.

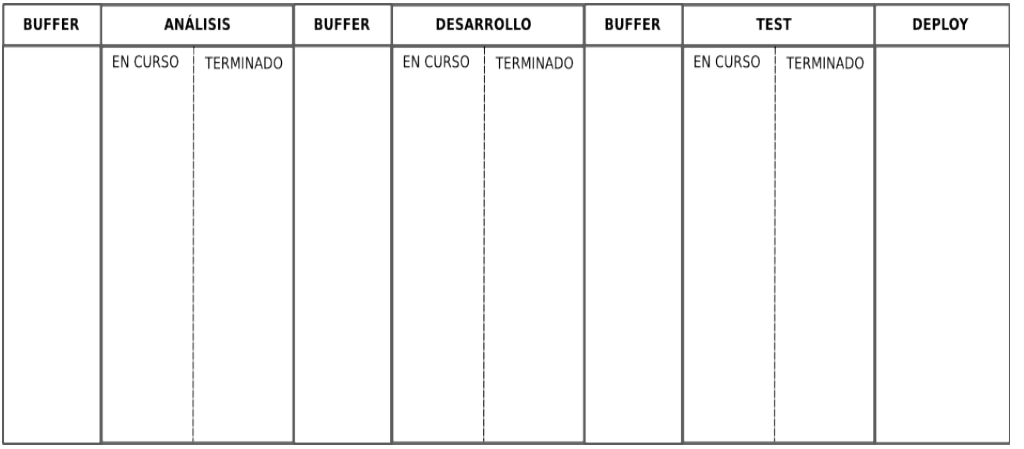
Los tableros Kanban

Al igual que en Scrum, en Kanban se utilizan tableros físicos (o herramientas digitales equivalentes) para representar el flujo del trabajo.

En los tableros Kanban, el flujo del proceso de trabajo, se verá reflejado de izquierda a derecha en el tablero, mediante columnas que representen cada etapa (análisis, diseño, desarrollo, test, etc.) Y cada una de estas columnas podrá dividirse a su vez, en dos: en curso y terminado.

Frecuentemente, en medio de cada columna representativa de un proceso, suele colocarse una columna intermedia para los ítems en espera.

Un ejemplo de tablero Kanban puede verse en la siguiente imagen:

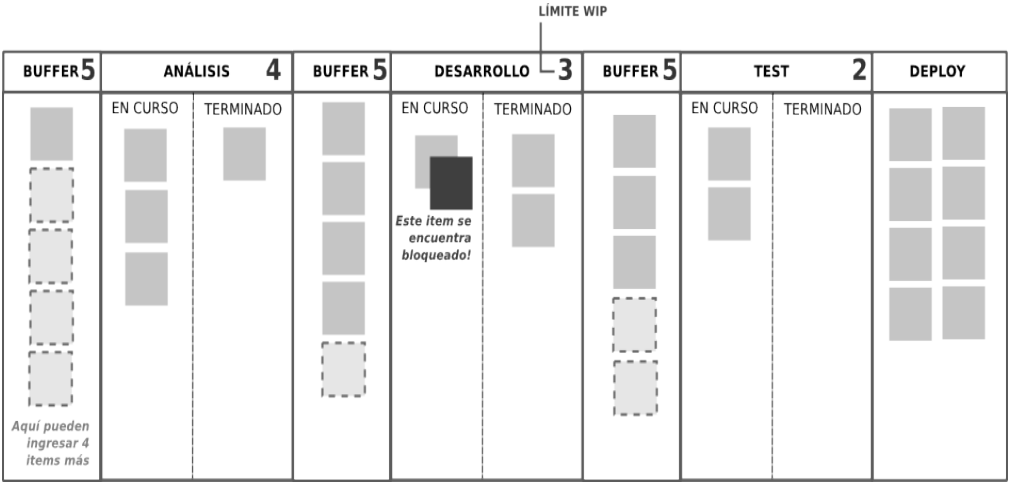


Limitar el trabajo en curso: WIP

En Kanban, el límite de trabajo en curso, está dado por el **WIP**: *Work in Progress*.

El límite WIP, consiste en definir la **cantidad de ítems simultáneos que pueden ejecutarse en un mismo proceso**.

Cada proceso, puede tener diferentes límites WIP, como se muestra a continuación:



Los límites WIP, nos permitirán detectar los **cuellos de botella** rápidamente. Con solo observar el tablero prestando atención en las columnas bloqueadas y su procesos anterior y posterior inmediatos, podremos deducir donde se encuentran los puntos de conflicto, y trabajar para remediarlos.

Existen diversas formas de **manejar los cuellos de botella**, pero sin dudas, la que mejore resultados ha demostrado dar, es la de colocar una cola de entrada (buffer) previa al proceso donde el cuello de botella se genera.

Ésto, se realiza a fin de **fomentar la colaboración del equipo**:

> *Cuando un cuello de botella es detectado, habrá procesos anteriores y posteriores que se encuentren bloqueados, generando en consecuencia, que los miembros del equipo cuyo proceso esté a cargo, no tengan tareas*

> *que realizar. La detección de los cuellos de botella, provocará que aquellos miembros del equipo "estancados", colaboren con sus compñaeros, para acelerar los procesos y liberar espacio para el ingreso de nuevos items.*

Optimizar el flujo de trabajo

El flujo de trabajo es la progresión visible de los ítems de los sucesivos procesos en un sistema Kanban. El objetivo de optimizar dicho flujo de trabajo, es alcanzar un proceso de desarrollo estable, previsible y acorde a las necesidades del proyecto, en el cual se distinga un ritmo de trabajo parejo.

Cuando un ítem de trabajo se encuentra estancado, el equipo deberá colaborar a fin de lograr recuperar un flujo parejo y tomar medidas para prevenir futuros estancamientos.

> "Un flujo regular, establece la capacidad de un equipo para entregar software funcionando con una velocidad fiable. Una organización que entrega con un flujo regular logra establecer claramente las posibilidades de su proceso y puede medir fácilmente su capacidad." (Mary y Tom Poppendieck, Lean software development)

Si bien no existen reglas preestablecidas para optimizar el flujo de trabajo en Kanban, las principales actividades sobre las cuales suele enfocarse esta optimización, en la práctica, son:

- El trabajo sobre cuellos de botella
- Análisis sobre las colas de entrada (buffer de ítems de trabajo)
- Mejoras que impliquen modificaciones en el proceso de creación de valor