





# Desarrollo de Software Ágil en 10Pines



# Desarrollo de Software Ágil en 10Pines

Federico Zuppa

## Colaboraciones:

Jorge Silva

Juan Manuel Carraro

Germán Gaitán

Gisela Decuzzi

Hernán Wilkinson

Nicolás Paez





Zuppa, Federico

Desarrollo de software ágil en 10pines / Federico Zuppa; contribuciones de Hernán Wilkinson ... [et al.]; editado por Cristian Boccia; ilustrado por Emmanuel Schonfeld; Gabriela Iztueta. - 1a ed. - Ciudad Autónoma de Buenos Aires: 10Pines, 2019.

199 p.: il.; 21 x 15 cm.

ISBN 978-987-47360-0-0

1. Informática. I. Wilkinson, Hernán, colab. II. Boccia, Cristian, ed. III. Schonfeld, Emmanuel, ilus. IV. Iztueta, Gabriela, ilus. V. Título.

CDD 005.4

Impreso por La Imprenta Ya  
Todos los derechos reservados.

A Belisa, por estar, aún cuando no estuvimos

# Tabla de contenido

Tabla de contenido .....	9
Prólogo.....	15
Introducción .....	17
CARACTERÍSTICAS PRINCIPALES .....	17
FLUJO DE TRABAJO .....	19
Product Discovery.....	21
FASE DE PRODUCT DISCOVERY .....	23
<i>¿Qué es lo que se busca descubrir?</i> .....	23
Talleres de Product Discovery.....	25
Lean Sales Up – por Jorge Silva .....	25
HERRAMIENTAS .....	28
 <i>User Story Mapping</i> .....	28
 <i>Lean Canvas</i> .....	35
 <i>Elevator Pitch</i> .....	38
 <i>Personas</i> .....	39
 <i>¿Qué nos quita el sueño?</i> .....	40
 <i>Experimentos de usabilidad</i> .....	41
CONCLUSIONES.....	45
User Stories & Backlog .....	46
USER STORIES.....	47

<i>Un poco de historia</i> .....	47
<i>Atributos de las User Stories</i> .....	48
<i>Escribiendo User Stories</i> .....	50
<i>Proceso de descubrimiento</i> .....	54
 <i>Example Mapping</i> .....	55
<i>Partiendo User Stories</i> .....	57
<b>EL BACKLOG</b> .....	58
<i>¿Qué es el Backlog de Producto?</i> .....	59
<i>Atributos importantes</i> .....	59
<i>Herramientas digitales</i> .....	60
<b>CONCLUSIONES</b> .....	61
<b>Estimaciones</b> .....	62
<b>UNA PRIMERA NOCIÓN DEL TAMAÑO</b> .....	63
<b>ESTIMACIONES RELATIVAS USANDO STORY POINTS</b> .....	63
<i>¿QUÉ SE TIENE EN CUENTA AL REALIZAR UNA ESTIMACIÓN?</i> ...	66
 <i>Planning Poker</i> .....	66
<b>ESTIMANDO EL BACKLOG</b> .....	68
<b>COSTO Y TIEMPO DE TRABAJO</b> .....	69
<i>VELOCITY Y BURNDOWN</i> .....	70
<b>ASPECTOS IMPORTANTES EN LAS ESTIMACIONES ÁGILES</b> .....	71
<b>CONCLUSIÓN</b> .....	72
<b>Priorización</b> .....	73
<b>SABER QUÉ ES LO IMPORTANTE ES LO IMPORTANTE</b> .....	74
<i>¿QUÉ DEBE TENERSE EN CUENTA AL PRIORIZAR?</i> .....	75

RELEASES CORTOS .....	76
HERRAMIENTAS .....	77
<i>User Story Mapping</i> .....	78
<i>MoSCoW</i> .....	79
<i>Puntos de valor de negocio</i> .....	80
<i>Tracer Bullets</i> .....	83
<i>Product Roadmap</i> .....	84
<i>Buy a Feature – por Germán Gaitan</i> .....	84
PRIORIZANDO EXPERIMENTOS.....	86
CONCLUSIONES.....	87
Planeamiento Continuo.....	89
INTRODUCCIÓN .....	90
¿QUÉ CONTIENE UN PLAN?.....	91
<i>Atributos</i> .....	92
<i>Aristas</i> .....	93
MIDIENDO LA <i>VELOCIDAD</i> Y REPLANEANDO.....	97
<i>Burndown Chart</i> .....	97
<i>Parking Lot - por Gisela Decuzzi - Una forma de mostrar avance</i> .....	99
PLANNING FAILURE MODES .....	103
<i>Un plan detallado para lidar con la complejidad</i> .....	103
<i>Multitasking</i> .....	103
<i>Deadlines estrictos</i> .....	105
CONCLUSIONES.....	106
El Corazón de los Equipos Ágiles en 10Pines .....	107

INTRODUCCIÓN .....	108
¿CÓMO FORMAMOS LOS EQUIPOS?.....	110
LOS EQUIPOS SON LAS PERSONAS .....	110
<i>Proceso de selección</i> .....	111
<i>Skills de los pinos</i> .....	112
<i>El camino del pino</i> .....	114
UN BUEN AMBIENTE .....	115
<i>¿Qué nos motiva?</i> .....	116
<i>¿Qué hacemos para lograr un buen ambiente?</i> .....	118
<i>Las 5 disfunciones de las empresas</i> .....	122
VISIÓN.....	124
AUTO-ORGANIZACIÓN .....	126
<i>¿Qué logramos con la auto-organización?</i> .....	126
<i>Auto-organización en 10Pines</i> .....	128
<i>Líderes en 10Pines</i> .....	129
<i>Disciplina</i> .....	130
COMUNICACIÓN .....	131
<i>¿Cuál es la manera más eficiente de comunicarse?</i> .....	132
CONCLUSIÓN.....	133
Desarrollando Software, Paso a Paso.....	135
INTRODUCCIÓN .....	136
DESARROLLO ITERATIVO E INCREMENTAL.....	136
TRABAJANDO EN ITERACIONES DE DURACIÓN FIJA .....	138
<i>¿Siempre usamos iteraciones fijas?</i> .....	138
EL TRABAJO DE UNA ITERACIÓN.....	139

¿QUÉ LOGRAMOS TRABAJANDO DE ESTE MODO?.....	140
CONCLUSIÓN .....	141
Desarrollando Software con Excelencia Técnica .....	142
INTRODUCCIÓN .....	143
CLEAN CODE .....	143
DEUDA TÉCNICA.....	144
COSTO DE CAMBIO .....	144
ALGUNAS PRÁCTICAS ESENCIALES .....	146
<i>Testing automatizado</i> .....	146
<i>Test Driven Development - por Hernán Wilkinson</i> .....	152
<i>Refactor Continuo</i> .....	180
<i>Code Review</i> .....	181
<i>Pair Programming</i> .....	182
<i>Integración Continua</i> .....	183
LA PARED DE SCRUM.....	185
MANIFIESTO DE SOFTWARE CRAFTSMANSHIP.....	185
SOBRE PRÁCTICAS TÉCNICAS Y PRÁCTICAS DE GESTIÓN - POR NICOLÁS PAEZ .....	186
CONCLUSIÓN .....	189
Conclusión.....	190
Bibliografía .....	193
Sobre el autor y los invitados.....	195
Agradecimientos.....	197



# Prólogo

A fines del año 2006, cuando asistí al curso de *ScrumMaster* dictado por Tobías Mayer no había muchas empresas por estas latitudes que tomaran las Metodologías Ágiles como algo serio. La mayoría ni siquiera habían escuchado de lo que trataba. Era el tercer curso de *Scrum* dictado en el país y el segundo de Tobías, que volvería unos años más tarde como invitado especial a participar de la primera conferencia Ágiles latinoamericana. Durante dicha conferencia, que organicé junto con algunos otros entusiastas, Tobías tuvo algunos intercambios fuertes en el panel de cierre con Mary Poppendieck y otros detractores de *Scrum*. Valientemente, aunque sin muchos argumentos, Tobias defendía a duras penas los sablazos que recibía de un lado y de otro sobre las fallidas implementaciones de *Scrum* que estaban poniendo en peligro la reputación de todas las Metodologías Ágiles.

Poco más de diez años después, las Metodologías Ágiles de desarrollo se han vuelto tan populares, que prácticamente no existen empresas que no las hayan adoptado o estén en proceso de adoptarlas. Desde pequeñas startups, puramente tecnológicas, hasta grandes corporaciones tradicionales, sin importar el tamaño o el mercado, consideran aplicar estas metodologías en algunas de sus formas. Lamentablemente, como suele ocurrir cuando algo se vuelve popular rápidamente, lo esencial se pierde en el camino. Aquel debate tan apasionado que llevaron adelante en ese panel de cierre tiene hoy tanta vigencia como en aquel entonces. Seguir una receta sin comprender los fundamentos y buscar objetivos de corto plazo sin una visión holística dan como resultado que muchas organizaciones, a pesar de haber adoptado Metodologías Ágiles, no están realmente haciendo las cosas de modo muy diferente a como lo hacían antes.

Debido a esto, en mi punto de vista, el valor principal que aporta el libro de Federico va mucho más allá de un mero relato de experiencias. En todos los capítulos se busca dejar en claro cuáles son las ideas detrás de cada práctica y en definir el porqué antes que el cómo. El libro nos invita

## Prólogo

a recorrer de forma didáctica el camino completo desde la concepción de un nuevo producto hasta los detalles técnicos para desarrollarlo, de la mano de una metodología que maximiza el aprendizaje y la entrega de valor continuo.

Otro aspecto, por el que este libro se destaca, implica escuchar la experiencia de alguien que trabaja día a día con estas herramientas, poniéndolas en práctica no solo para enseñar a otros, sino para hacer su propio trabajo. Sin quitarle valor a todo el material que otras personas dedicadas a la consultoría han escrito, en mi opinión, es diferente la perspectiva de aquellos que realmente tienen que enfrentarse cara a cara con los problemas y resolverlos para lograr ser exitoso.

Estoy seguro de que este libro será un gran aporte a la comunidad y, ojalá, podamos desde Latinoamérica seguir generando material en castellano de calidad que ayude a difundir nuevos conocimientos en las áreas del desarrollo de software ágil.

Emilio Gutter - Fundador @10Pines

# Introducción

Todavía recuerdo mi reunión con Emilio Gutter, hasta ese momento un ex-colega y amigo de la comunidad Ágil, en la cual me invitó a ser parte de 10Pines. Me contó que habían fundado una empresa donde las personas y la calidad humana prevalecían y cuyo objetivo principal era generar valor haciendo lo que mejor sabíamos hacer: desarrollo de software ágil. La oferta fue muy tentadora: formar parte, desde los inicios, de una empresa concebida bajo los valores ágiles. ¡Eso sucedió hace 10 años y hoy sigo aquí!

Durante estos años, los pinos (quienes formamos parte de 10Pines) elaboramos nuestros procesos, seleccionamos herramientas y estandarizamos artefactos. En resumen, co-creamos nuestra Metodología<sup>1</sup>, que recientemente sentí la necesidad de observar con detenimiento, estudiar y describir. Este libro es el resultado de dicho proceso.

## Características principales

Nuestra Metodología está basada en las personas, es liviana (enfocada en producir valor), está centrada en la calidad y amplifica la comunicación. Les cuento con más detalle:

---

<sup>1</sup> La Metodología es todo lo que haces regularmente en tu empresa para desarrollar software. Incluye a quien se contrata, para qué, cómo se trabaja, qué se producen y comparte. Entre los componentes de una Metodología, podemos nombrar los *skills*, los roles definidos, las actividades en las que estos roles invierten su tiempo (programar, testear, reuniones), los procesos (cómo estas actividades encajan en el tiempo), los artefactos que producen, los *standards* que adoptaron y, por supuesto, los valores compartidos como grupo. (Cockburn, 2006)

**Basada en las personas:** Está pensada para que trabajemos felices, satisfechos y hagamos nuestra labor de la mejor manera posible. Nos valoramos por sobre cualquier proceso o herramienta. Creemos que, para el desarrollo de software, nada reemplazará a buenos desarrolladores interactuando efectivamente para crear una solución.

**Es liviana<sup>2</sup>, enfocada en la generación de valor:** Procuramos que nuestra actividad principal sea la programación, dejando los mecanismos de coordinación y control al mínimo. Queremos que sea tan liviana, como el problema a resolver lo permita. Suficiente con lo justo<sup>3</sup>. Que no le sobre nada, pero tampoco que le falte, porque esto sería contraproducente. En busca de este objetivo, continuamente procuramos eliminar el “desperdicio”<sup>4</sup> de la cadena de valor, logrando una Metodología magra, donde cada una de las actividades ejecutadas y los artefactos creados contribuyan a agregar valor.

**Centrada en la calidad:** La calidad externa, definida por Weinberg como “valor para alguien” es subjetiva y difícil de especificar. Por esto, nuestra Metodología busca acortar los ciclos de feedback, trabajando en incrementos pequeños que permitan generar conocimiento rápidamente. Sabemos también que la calidad interna, la excelencia técnica, resulta fundamental para lograr nuestro objetivo: ser capaces de generar valor sustentablemente, aún cuando el contexto cambie. Por todo esto, “horneamos la calidad desde el principio”<sup>5</sup>.

**Amplifica la comunicación:** es una Metodología que procura maximizar el ancho de banda en todas sus fases. Si establecemos un flujo de comunicación efectivo que incluya a todos los actores, tendremos más probabilidades de éxito. El involucramiento de nuestros clientes a través de una relación de colaboración es central en este sentido.

---

<sup>2</sup> El “peso de la Metodología” está dado por la cantidad de “elementos de control” necesarios (deliverables, *standards*, medidas de calidad) y la “ceremonia” (cuánto se demora) de cada uno de ellos. (Cockburn, 2006)

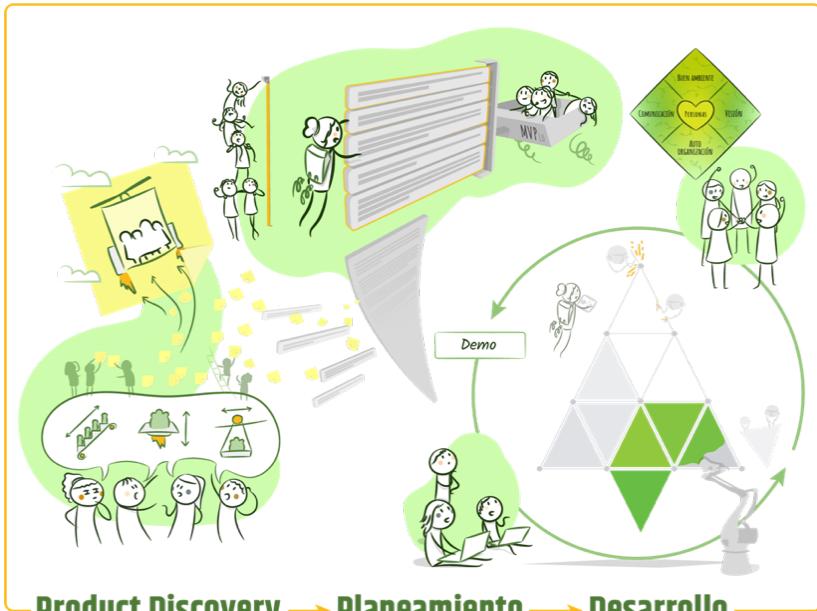
<sup>3</sup> *Barely Sufficient*. (Highsmith, 2009)

<sup>4</sup> En el mundo de *Lean*, “desperdicio” es toda actividad que no agregue valor, viéndolo desde la perspectiva del usuario final.

<sup>5</sup> *Bake quality in*. (Poppendieck, 2003)

Quisiera que los lectores evalúen el cumplimiento de estos atributos en los procesos, herramientas y artefactos que describiré a lo largo de estas páginas.

## Flujo de trabajo



### Product Discovery → Planeamiento → Desarrollo

El gráfico describe las diferentes fases que forman parte de nuestro proceso de desarrollo, juntamente con los artefactos principales generados en cada una de ellas:

Empezamos cada proyecto con una fase que llamamos *Product Discovery*, donde procuramos entender los problemas y necesidades de los usuarios y esbozar una propuesta de valor. De este modo, creamos una visión compartida, nos alineamos y sentamos las bases de una relación de colaboración.

Refinamos nuestra visión escribiendo *User Stories*, incrementos funcionales pequeños que almacenamos en orden de prioridad en el

*Backlog.* Descomponer el trabajo de esta manera nos permite utilizar un proceso de desarrollo iterativo e incremental.

Realizamos una estimación superficial para obtener una noción del esfuerzo (y el costo consecuente), que usaremos para planificar el equipo y los tiempos. Las estimaciones tienen un grado de error y nunca representan un compromiso.

Priorizamos la construcción del producto para comenzar el desarrollo por lo que genere más valor y conocimiento (disminuyendo el riesgo). Lo importante es entender cuál es el Producto Mínimo Viable y cómo construirlo de la mejor manera.

Con toda esta información, creamos un plan liviano, creíble y adaptable, que iremos cambiando a medida que aprendamos.

Conformamos un equipo, que trabajará en un buen ambiente, con una visión clara, auto-organizado y comunicándose de la manera más efectiva.

Desarrollamos el software en incrementos (las *User Stories* creadas), iterando sobre el conocimiento adquirido. Trabajar de esta manera nos permite incrementar nuestro conocimiento y poder adaptarnos, impactando en el valor generado.

Desarrollamos enfocados en la excelencia técnica desde el inicio. Mantenemos el costo de cambio bajo y pagamos la deuda técnica para incrementar nuestra funcionalidad sustentablemente y adaptarnos a los cambios externos.

¡Los invito a continuar leyendo las páginas que siguen para entender profundamente el significado de trabajar y ser parte de nuestro mundo!

# *Product Discovery*



*“La parte más difícil de construir un sistema de software es, precisamente, decidir qué construir”<sup>6</sup>*

---

<sup>6</sup> *The hardest single part of building a software system is deciding precisely what to build.* (Brooks, 1995)

En 10Pines construimos software para diversos tipos de clientes, desde grandes corporaciones como Claro y Burger King, que quieren desarrollar nuevos productos o ampliar la funcionalidad de herramientas existentes, hasta emprendimientos recién nacidos que buscan construir una versión inicial del producto. Si bien las necesidades, el tamaño del problema y obviamente el presupuesto difieren, siempre comenzamos nuestros desarrollos con una etapa que denominamos ***Product Discovery*** donde nos juntamos con los especialistas en el negocio, nuestros clientes, para crear una **visión compartida** de lo que se desea construir.

El objetivo de esta etapa es entender quiénes serán los usuarios y clientes principales, cuáles son sus problemas y necesidades y cómo las vamos a resolver. Además, tener una idea del tamaño, establecer prioridades, identificar riesgos y acordar cuál es el producto mínimo viable<sup>7</sup>.

Existe una diferencia entre la simple comprensión de los requerimientos y el grado de entendimiento e involucramiento que buscamos en esta etapa en 10Pines. Esto es lo que nos permite **alcanzar un intenso grado de colaboración con todos nuestros clientes**. Diseñar una solución es algo muy complejo que se logra a través de la interacción y colaboración de todos estos actores desde el momento en que nos contactan. **El compromiso y la sinergia**, alcanzados por el equipo en esta etapa, son muy importantes. Es por eso que en todos nuestros proyectos diseñamos las soluciones juntos, proponemos y discutimos alternativas y nos enfocamos en alcanzar los objetivos, **maximizando la colaboración y la comunicación**.

---

<sup>7</sup> [https://es.wikipedia.org/wiki/Producto\\_viable\\_minimo](https://es.wikipedia.org/wiki/Producto_viable_minimo)

## Fase de *Product Discovery*

*Product Discovery* es la fase inicial de nuestra Metodología. Sentamos aquí las bases de lo que constituirá el resto del desarrollo. Creemos firmemente en los valores ágiles de comunicación y colaboración. Por esta razón, intentamos maximizar el ancho de banda y establecer lazos de colaboración cercanos desde el comienzo.

Facilitamos esta etapa a través de una serie de talleres que desarrollamos en colaboración con nuestros clientes. El resultado es un conjunto de artefactos que nos permitirá establecer un plan para desarrollar el producto. Es importante remarcar que la documentación producida durante esta etapa es creada, de forma eficiente, a través de la colaboración de todos los involucrados, generando, al mismo tiempo, un importante **caudal de conocimiento compartido**.

### *¿Qué es lo que se busca descubrir?*

Antes de describir el proceso y las herramientas que usamos en esta etapa, detallaremos qué es lo que se intenta descubrir:

**Por qué:** ¿Cuáles son los problemas más importantes que desean resolver? ¿Por qué es necesaria esta herramienta? Entender todo esto en detalle nos permite diseñar soluciones mejores, muchas veces sugiriendo alternativas a las inicialmente propuestas por nuestras clientes.

**Quiénes y qué:** Quiénes son los actores principales y qué procesos de negocios llevarán a cabo en el sistema a desarrollar. Por supuesto, existirán flujos principales y alternativos. En esta etapa detectaremos todos, aunque, después, terminemos priorizando algunos y descartando otros.

**Prioridades:** Si algo aprendí en todos estos años de desarrollo es que el tiempo y el dinero nunca alcanzarán para construir todo lo que se desea. Es por eso que parte de las actividades, que desarrollamos en esta etapa, consiste en identificar cuáles son los *features* indispensables, es decir, aquellos que no pueden faltar o con los que se obtendrán mayores beneficios. Entender las prioridades y trabajar en ellas resultan fundamentales para el éxito del proyecto.

**Producto Mínimo Viable:** Resulta valioso entender cuál es el experimento mínimo que nuestros clientes están dispuestos a testear con usuarios reales. Sus altas expectativas muchas veces los impulsan a buscar la implementación de todas sus ideas, pretendiendo obtener éxito desde el comienzo. Lamentablemente, en la mayor parte de las ocasiones esto no ocurre. Es por eso que los alentamos a diseñar experimentos más pequeños que testejen sus hipótesis con usuarios reales. Nuestro objetivo es **crear conocimiento validado de la manera más rápida y eficiente posible.**

**Modelo de negocio:** Cuando trabajamos con *startups*<sup>8</sup>, creemos que es importante entender cuál es su modelo de negocios, es decir, cómo piensan ganar dinero y cuál es el diferencial del producto. Si el proyecto es para una corporación, entonces es importante entender cuáles son los procesos que esta herramienta optimizará o cómo se obtendrán nuevos ingresos a partir de la misma.

**Riesgos:** Es muy común que en las conversaciones realizadas en esta etapa se detecten riesgos, tanto técnicos como de negocio. Éstos deben ser marcados y discutidos, ya que representan información esencial para la posterior priorización de *features*.

**Estimación inicial de la magnitud:** A partir de las conversaciones sostenidas en esta etapa, se puede tener una idea inicial de cuán grande podría llegar a ser el desarrollo.

Marty Cagan, en su libro “*Inspired, how to create products that customers love*” (Cagan, 2008), afirma que para que un producto sea exitoso, debe ser **valioso, usable y posible de desarrollar**. La fase de *Product Discovery* tiene el objetivo de crear una visión compartida de cómo alcanzar estas metas. Para eso debemos entender dónde está el valor del producto, cómo va a ser usado (su usabilidad) y, por supuesto, si técnicamente es factible de construir.

---

<sup>8</sup> [https://en.wikipedia.org/wiki/Startup\\_company](https://en.wikipedia.org/wiki/Startup_company)

## *Talleres de Product Discovery*

La mejor forma de hacer este descubrimiento es a partir de un conjunto de reuniones, que realizamos junto con nuestros clientes y en las que empleamos un conjunto de herramientas que permiten que todos los participantes se involucren. Estas reuniones, a las que concurren usuarios de negocios, desarrolladores, testers, especialistas en UX y diseñadores, pueden durar desde 4 horas hasta una semana en proyectos de gran magnitud. La diversidad de roles enriquece el resultado, ya que, además de nuestros clientes, que conocen qué se requiere y cuáles son los objetivos, se necesitan desarrolladores que evalúen la factibilidad, testers que contribuyan con su atención a los detalles y especialistas en UX que evalúen la usabilidad. Es fundamental que concurran el *Product Manager*, el *Technical Leader* y un facilitador que se encargará de llevar la reunión adelante, explicar las herramientas que se van a usar y mediar en las conversaciones. Destaco que, más allá de los artefactos producidos, resultan de gran utilidad las conversaciones desarrolladas. ¡Es curioso, pero, muchas veces, detectamos que no existe una visión unificada, ni siquiera, entre diferentes integrantes de la misma empresa cliente! Discutir sobre todas las aristas que tendrá el producto, en esta etapa, resulta esencial para unificar el entendimiento y las visiones de todas las personas involucradas en el desarrollo. Jeff Patton llama a esto “comprensión compartida” (Patton, 2014).

## *Lean Sales Up – por Jorge Sihva*

El proceso de preventa puede ser doloroso, largo y caro. En esta primera etapa debemos convencer al prospecto que somos la mejor opción, ayudándolo, al mismo tiempo, a definir y ajustar sus expectativas (casi siempre altas). Por lo general, esto termina en una sobre-estimación (el clásico “colchón”), promesas poco realistas sobre alcances y fechas y otros comportamientos disfuncionales que pueden ensuciar la relación desde el comienzo.

Cuando trabajaba en empresas tradicionales, cada vez que alguien me hablaba de la etapa de preventa, me preguntaba: “¿qué cosas imposibles habrá prometido el vendedor?” o “¿cuánto tiempo habrá recortado de la estimación para poder venderlo?”. Estas situaciones resultan frecuentes

dentro de un enfoque clásico que procura maximizar las ventas, ajustando lo necesario para que el cliente compre. Del otro lado está el desarrollador, que recibe esa venta optimista y debe entregar algo al cliente.

Como podrán imaginar, el proceso de preventa clásico entra en conflicto con el modelo de empresa que tenemos, principalmente porque:

- El precio es conocido sólo por los vendedores y el cliente.
- Las expectativas del cliente son “endulzadas” para cerrar la venta.
- El alcance del proyecto es definido sin el involucramiento del equipo de desarrollo.
- Las fechas son irreales.
- El equipo de desarrollo no se involucra con los objetivos del cliente.
- Las diferencias entre vendedores y desarrolladores resultan de difícil resolución.

Por otro lado, es una realidad que es necesario vender para mantener la empresa viva. Para poder lograrlo de forma sustentable, elaboramos dos ideas principales:

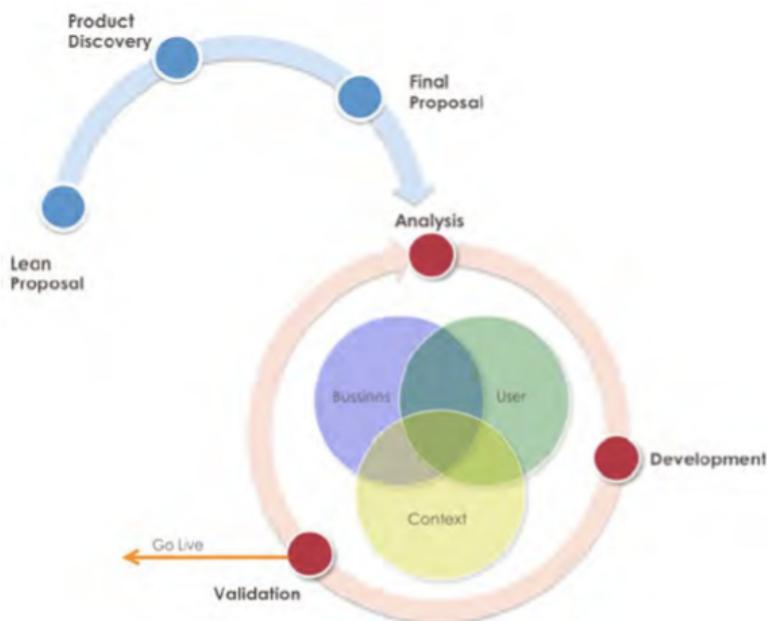
1. Los desarrolladores debemos vender. “Si quieres hacerlo bien, hazlo tu mismo”.
2. Necesitamos integrar las ventas con el desarrollo.

A partir de estas ideas, redefinimos nuestro proceso de pre-venta, fusionándolo al de desarrollo y alineándolo a nuestros valores. Esto significa que no existe más una etapa de preventa separada del ciclo de desarrollo. En su lugar unimos cada uno de los procesos en uno solo. De esta forma, ambos roles, desarrollo y ventas, son conscientes de la complejidad de dichas actividades. Por otro lado, borramos la frontera entre las dos áreas, creando un equipo más completo, capaz no sólo de desarrollar software, sino también de venderlo y explicarle al cliente por qué somos la mejor opción.

Luego de 10 años de trabajo continuo y mejoras, podríamos decir que nuestro proceso de preventa se basa en estas tres heurísticas:

- **Explicar qué es agilidad:** No todos saben lo que implica, por lo cual debemos transmitir estas ideas al cliente.
- **Descubrir tu producto:** Creamos el taller de “*Product Discovery*”, una herramienta poderosa usada para identificar qué construir, haciendo la preventa más exitosa.
- **Entender la psicología de tu cliente:** Desafiar las ideas de tu potencial cliente, entender cómo piensa, cómo decide, qué necesita y qué no.

Luego de todo este aprendizaje sobre las pre-ventas, terminamos con un proceso de desarrollo ágil que posee esta forma:



En esta imagen se observa que la preventa, fase donde se concibe el producto, es parte del proceso de desarrollo. Se entiende la venta de modo más holístico, con mayores probabilidades de éxito.

## Herramientas

A continuación, describiré algunas de las herramientas que usamos habitualmente en los talleres:



### ❖ User Story Mapping

#### Introducción

Aprendí esta técnica cuando vivía en San Francisco, en 2009, en uno de los tantos *open spaces* a los que asistí. Por ese entonces, yo trabajaba con *Scrum* o *eXtreme Programming* desde hacía algunos años y uno de los cuestionamientos que poseía era que el *Backlog* (que trataré en el próximo capítulo) no me dejaba visualizar, de manera nítida, el trabajo que debíamos realizar. ¿No resulta difícil poder visualizar un producto, a través de una lista de *User Stories*? Descubrir esta herramienta fue realmente un hallazgo para mí.

¿Qué es un *User Story Mapping*? En pocas palabras, **es un mapa que describe los procesos de negocios de cada uno de los actores que forman parte del sistema**, a través de las tareas que deben llevarse a cabo para completar cada uno de estos procesos. La técnica fue popularizada por Jeff Patton dentro del mundillo ágil. El resultado, desde el plano visual, será algo similar a lo que puede verse en la siguiente imagen:



¿Por qué me gusta tanto esta herramienta? Básicamente, porque es:

- **Intuitiva:** Puedo explicarla en minutos a un público completamente nuevo y empezar inmediatamente a construirlo.
- **Fácil de Crear:** Crear una tarea implica solamente escribir un *post-it* y pegarlo. Cualquiera puede hacerlo sin crear ninguna disrupción. El tiempo se invierte en descubrir estas tareas entre todos y en las conversaciones que se disparan. Los *post-its* pueden reacomodarse sin ningún esfuerzo, a medida que se descubren nuevas tareas o se repriorizan las existentes.
- **Visual:** El mapa nos permite visualizar a los actores junto a sus actividades principales y a la composición de las mismas. Podemos recorrer estos flujos de negocios, simplemente, leyendo las tareas de izquierda a derecha, algo que resulta intuitivo y natural. También, observar qué es importante, simplemente viendo qué tareas se encuentran arriba de otras. El mapa contiene mucha información, presentada de forma clara.

### Cómo construimos el *User Story Map*

Lo primero que se debe hacer es identificar a los actores principales del sistema, es decir, cuáles serán los tipos de usuarios. Muchas veces, resulta dificultosa la identificación de todos los usuarios en una instancia inicial, pero no se preocupen, porque, a medida que avancemos con la construcción del *Story Map*, identificaremos al resto. Por cada nuevo tipo

de usuario, pegaremos un *post-it* que lo identifique, dejando espacio para descubrir todo lo que éste podrá realizar en el sistema.

Para graficar la construcción del *Story Map*, introduciré un ejemplo muy sencillo extraído de un video de nuestro amigo David Hussman<sup>9</sup>.

Imaginen que deben construir un sistema para un supermercado, que ayude a los empleados que cobran la mercadería. El sistema debe, en pocas palabras, permitir el escaneo de los productos para luego cobrarlos y facturarlos. En consecuencia, el tipo de usuario que identificaremos será el del cajero.



Cajero

— tiempo —————→

Una vez que identifiquemos al actor principal, trataremos de descubrir cómo usará el sistema, es decir, cuáles serán los principales procesos de negocio bajo su responsabilidad. Jeff Patton los denomina **actividades**, término procedente de expertos en UX, tales como Larry Constantine y Don Norman y las define como “grandes cosas que el actor hace y que están conformadas por un conjunto de pasos, pero que no siempre tienen un *workflow* preciso definido”. En mi experiencia facilitando estos talleres, resulta difícil precisar estas actividades en una etapa inicial. Por esta razón, también podemos comenzar preguntando: ¿Qué es lo primero que el usuario hará en el sistema? Ese algo, escrito en forma de verbo, será nuestra 1<sup>ra</sup> tarea. Patton define una **tarea** como “algo pequeño, que el usuario hace en el sistema para llegar a un objetivo”. Parafraseando: ¿Cuál es la primera tarea que el usuario llevará a cabo en el sistema? Cuando la hayamos identificado, la escribiremos en un *post-it* que pegaremos debajo del que identifica al tipo de usuario. Posteriormente, buscaremos determinar qué hará a continuación, escribiremos el *post-it*

---

<sup>9</sup> *A User Story Mapping Example with David Hussman* - [https://www.youtube.com/watch?v=LgIfrpvLM\\_Y](https://www.youtube.com/watch?v=LgIfrpvLM_Y)

correspondiente y lo pegaremos a la derecha del anterior, señalando la cuestión temporal.

Retomando el ejemplo, ¿qué será lo primero que hará el cajero, cuando nos presentemos con nuestro canasto de compras? Pensaremos nuestro sistema de la manera más simple y diremos que “Ingresará el código de cada uno de los artículos que tiene en el canasto”. Llamaremos a esta tarea “Ingresar Código Producto”. ¿Qué haremos una vez ingresado el código de todos los artículos? Calcular el total. Llamaremos a la tarea con el mismo nombre y la pegaremos a la derecha del *post-it* anterior.



Una vez calculado el total, el cajero deberá cobrar. El medio de pago más sencillo de implementar es con dinero en efectivo, así que escribiremos una tarea que lo represente: “Tomar pago con efectivo”. Una vez realizado, “emitiremos el recibo”.

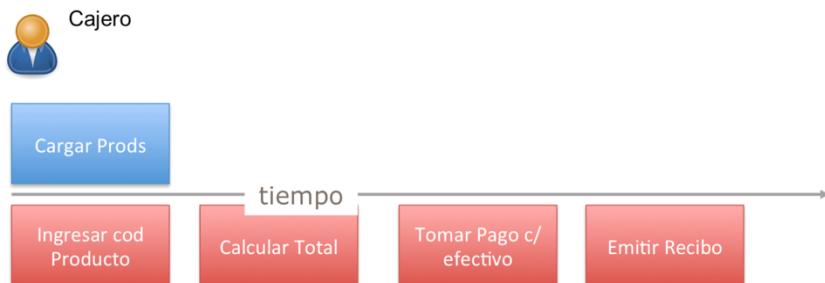


Podemos visualizar rápidamente el flujo de tareas de nuestro tipo de usuario principal.

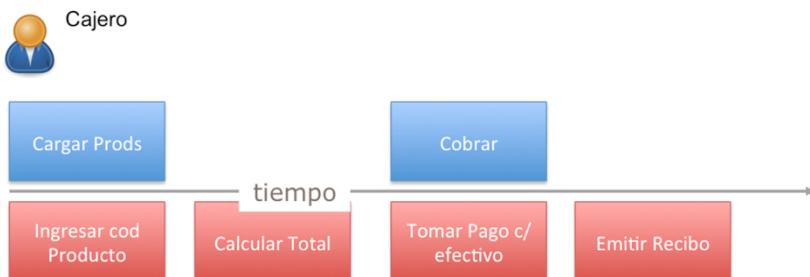
Seguramente que la charla llevará a querer agrupar estas tareas bajo un proceso de negocio o, como Patton las denomina, una actividad que las

identifique. Cuando descubramos esta actividad, escribiremos un *post-it* de otro color y lo pegaremos sobre el conjunto de tareas que engloba.

¿Cómo podríamos llamar a la actividad que engloba la introducción de todos los artículos y el cálculo del total a pagar? Por razones de simplicidad y falta de otros participantes para discutir el nombre de la misma, la llamaré: “Cargar Productos”. Nótese el *post-it* de diferente color sobre la 1<sup>ra</sup> tarea que comprende.



Por las mismas razones, denominaré a la actividad que engloba todas las tareas de cobro: “Cobrar”.

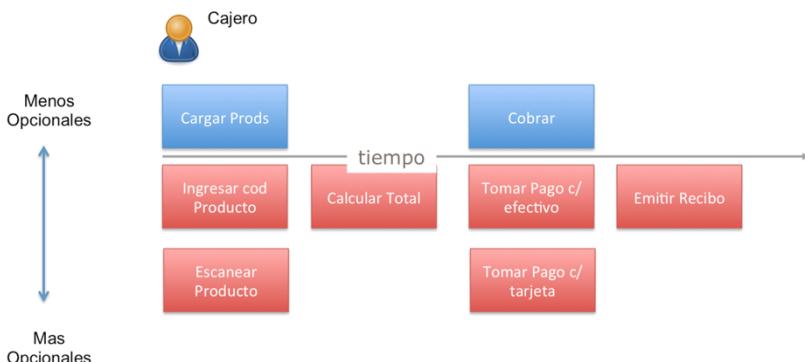


En este ejemplo, hemos partido del descubrimiento de tareas para luego abstraer las actividades. Podríamos haber comenzado también por la identificación de las actividades principales: “Cargar Productos” y “Cobrar” para luego descomponerlas en tareas. En definitiva, ya sea que partamos de la abstracción, es decir, de la actividad y la sepáremos en tareas o viceversa, de las tareas concretas y, en algún momento, abstraigamos la actividad, el resultado final será el mismo.

Una vez descrita la primera actividad (y el primer conjunto de tareas) preguntaremos: ¿qué otras tareas o actividades llevará a cabo en el sistema este usuario? Impulsados por esta interrogación, comenzaremos nuevamente el ciclo detallado.

Seguramente, en algún momento, nos daremos cuenta de que podemos hacer cierta tarea o cierta otra tarea para alcanzar un mismo objetivo (por supuesto que esto se puede extender a un flujo de tareas). ¿Qué haremos para representar esto, visualmente, dentro de nuestro *Story Map*? Pegaremos los *post-it*, que corresponden a las tareas opcionales, debajo de las tareas principales. Nótese que mencioné una tarea principal y una opcional, es decir, que priorizamos y esta priorización queda reflejada en nuestro *Story Map*.

Siguiendo con el ejemplo, imagínese que, en vez de “Ingresar el producto” de forma manual, lo escaneamos (como usualmente se hace en los supermercados). Escribiremos la tarea y la pegaremos debajo de la tarea principal. Lo mismo pasaría si identificásemos una tarea para “Tomar pago con tarjeta”, que será opcional a la principal de “Tomar pago con efectivo”. Nuestro *Story Map* quedará como ilustra el ejemplo:

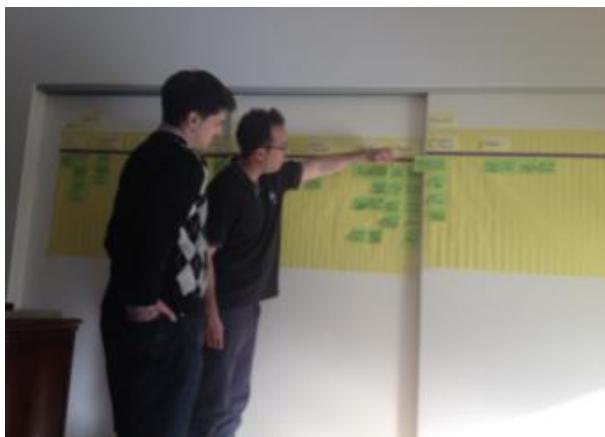


Hasta aquí, pudimos ver lo más importante referido a la herramienta. El resto no es más que seguir identificando flujos de tareas que los usuarios realizan. Cuando identificamos grupos de tareas relacionadas, abstraemos la actividad o proceso de negocio al que pertenecen. Si se tratases de tareas alternativas (o flujos de tareas), pegaremos los *post-it* debajo de las tareas principales. Ésto se da bajo una conversación continua delante del *Story Map*, que va reflejando todo el conocimiento que adquirimos y que

dispara nuevas charlas sin generar ninguna disrupción. Ahora, ¿ven por qué me gusta tanto?

### Recorriendo el *User Story Map*

Una vez finalizada la primera versión del *Story Map*, deberíamos hacer algo que Jeff Patton denomina “Recorrer el story map”, que implica la lectura de los *post-its* que representan cada una de las tareas, imaginando cómo el usuario trabajaría con el sistema. Podemos hacer este recorrido con diferentes personas, de manera de poder lograr diferentes opiniones. Por supuesto que, al recorrer el *Story Map*, descubriremos tareas olvidadas que obviamente agregaremos. ¡Los *post-its* nos brindan esta flexibilidad!

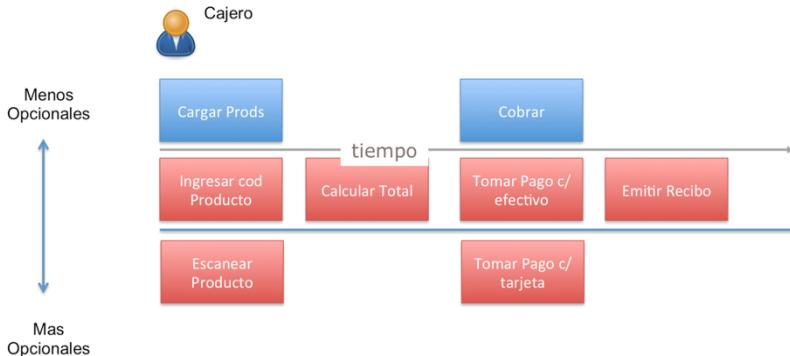


### Priorización usando el *User Story Map*

Antes observamos que es posible especificar que una tarea es más importante que otra, simplemente poniendo la tarea (o conjunto de tareas) en un nivel superior del *Story Map*. ¡Genial! Además de eso, el *Story Map* es una herramienta muy útil para decidir cuál es la primera versión del producto que podemos poner en producción; lo que muchos llaman Producto Mínimo Viable (*MVP - Minimum Viable Product*).

¿Cómo hacemos esto? Separando las tareas que estarán incluidas en el *MVP* de las que no. Para esto, bordeamos las tareas que pertenecen al

MVP o simplemente trazamos una recta que deje los *post-it's* arriba o abajo según corresponda (como muestra la imagen).



Decidir dónde debemos trazar la línea involucra muchísimos factores, muchos más de los que podría abordar en este capítulo. Lo importante a destacar es que la herramienta facilita la elección de una primera versión del MVP en un ambiente de colaboración, donde participan expertos de todas las áreas.

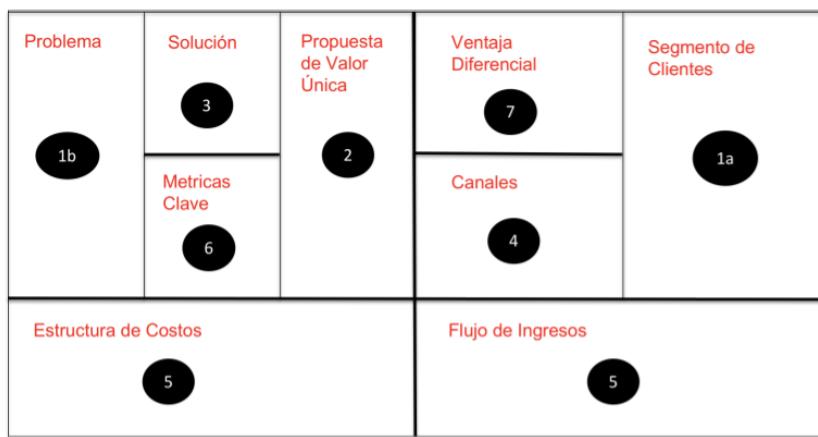
## 🔗 Lean Canvas

¿Ustedes piensan que los desarrolladores deberían entender como funciona el negocio que modelarán? Algunos dirán que no es necesario: mientras se comunique correctamente lo que se debe desarrollar es suficiente. Podrán adivinar que estoy en desacuerdo con esta afirmación. Creo que es importante entender cómo funciona o funcionará el negocio de la empresa que nos contrata. Si estamos desarrollando un producto nuevo (de una *startup*), es necesario entender cuál es el modelo de negocios que piensan implementar o, dicho de otra manera, cómo piensan generar dinero. Si se trata de una aplicación para una organización, entonces debemos entender por qué es necesaria, cuál es su valor o cómo recortará costos.

En el pasado, eran frecuentes los planes de negocio extensos, que requerían un gran esfuerzo. En la actualidad, es común el uso de una

herramienta llamada *Lean Canvas*, que es mucho más simple y liviana y, por ende, más atractiva para desarrollar en nuestros talleres.

*Lean Canvas* es una herramienta que popularizó Eric Ries, a través de su metodología *Lean Startups*, que sirve para especificar el modelo de negocios de una manera muy simple y efectiva. La herramienta se enfoca en el problema detectado y en la solución que proporciona un valor agregado, no existente en el mercado. Se encuentran disponibles diferentes versiones, por ejemplo, en *leanstack*<sup>10</sup> pueden encontrar esta versión (ilustrada por la imagen), definida como una adaptación del *business model canvas* de Alex Osterwalder.



<http://leanstack.com>



En el orden propuesto, iremos descubriendo:

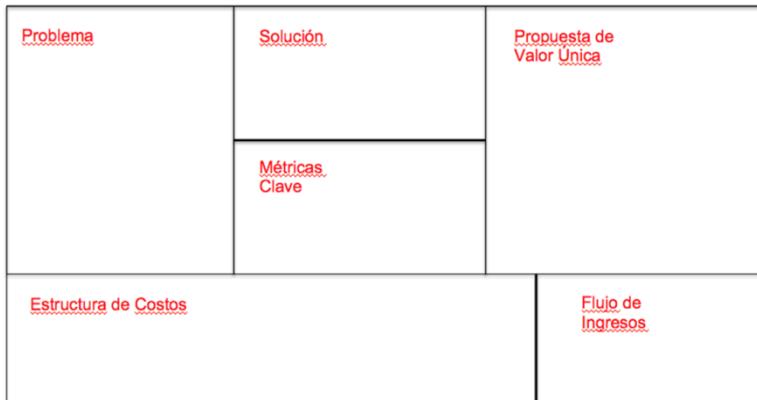
- 1a) A quiénes estará orientado el producto.
- 1b) Los tres problemas más importantes que estos usuarios enfrentan. ¿Cómo los resuelven en la actualidad?
- 2) Un mensaje simple, claro y atractivo que explique por qué tu solución es diferente y por qué valdría la pena comprarla.
- 3) Los tres *features* más importantes.

---

<sup>10</sup> <https://leanstack.com/>

- 4) Cómo llegar al cliente.
- 5) Flujo de ingresos: qué se cobrará y cuánto.
- 6) Cómo medir el progreso. Cómo saber si las hipótesis se cumplen.
- 7) Cuál es la ventaja diferencial del producto (aquella que los competidores no podrán alcanzar).

En *canvanizer*<sup>11</sup>, podemos encontrar una versión aún más reducida:



<http://canvanizer.com>

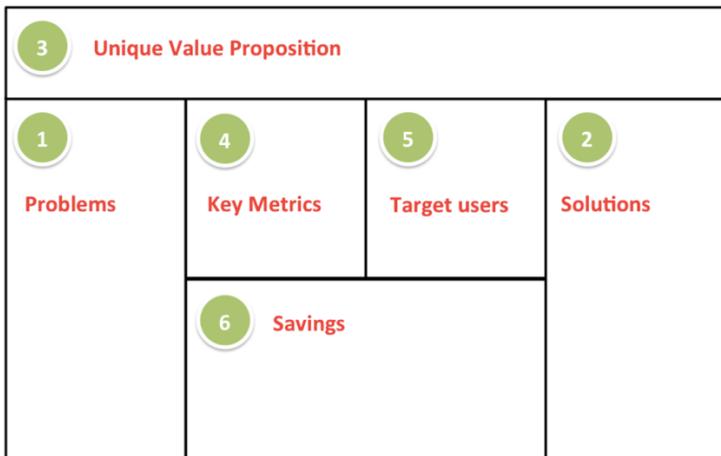


Mi colega Jorge Silva encontró un importante valor en el uso de esta herramienta en proyectos destinados a corporaciones, efectuándole algunos cambios, que incluyeron la adición de una sección de “Ahorros”, que permite describir cómo se ahorrará dinero en ciertos procesos. En este post<sup>12</sup> podrán encontrar más detalles.

---

<sup>11</sup> <https://canvanizer.com/>

<sup>12</sup> <https://blog.10pines.com/2015/07/13/enterprise-lean-canvas/>



[Enterprise Lean Canvas](#)

Tratar estos proyectos corporativos como si estuviesen destinados a una *startup* resulta un acierto, ya que evita caer en la trampa de pensar que el presupuesto es infinito y alinea las visiones de todas las áreas involucradas.

Hacer un taller que implique la creación del *Lean Canvas*, donde participe todo el equipo, permite que todas las personas que construirán el producto tengan una visión compartida del valor que agregará, cómo piensan venderlo y cuál es la competencia. El beneficio de contar con desarrolladores que comprendan la visión del negocio y de sus objetivos redonda en diseños pensados para alcanzarlos, el disparo de nuevas ideas, muchas veces no previstas, y en la propuesta de alternativas que busquen cumplirlos a un costo menor. En estos problemas complejos, la interacción de este grupo heterogéneo de profesionales es lo que maximiza las probabilidades de éxito.

## ⚡ Elevator Pitch

¿Cómo podríamos convencer a un inversor, en un lapso muy breve (por ejemplo, si compartiésemos el ascensor), de que tenemos una idea que vale la pena implementar? Nuestro *pitch* deberá resumir los conceptos claves del producto que tenemos en mente: los clientes y beneficios, qué

pensamos hacer, cuál es nuestro diferencial y cuál es nuestra ventaja competitiva.

Nos gusta incluir este ejercicio en nuestros talleres, porque nos permite entender, de una manera muy resumida, cuál es el modelo de negocios que tendrá el producto a desarrollar. Para realizar este taller, entregamos al equipo un afiche con los siguientes campos a completar:

**Para** (Clientes Claves)

**Quiénes** (Necesidad o Oportunidad)

**El** (Nombre del producto)

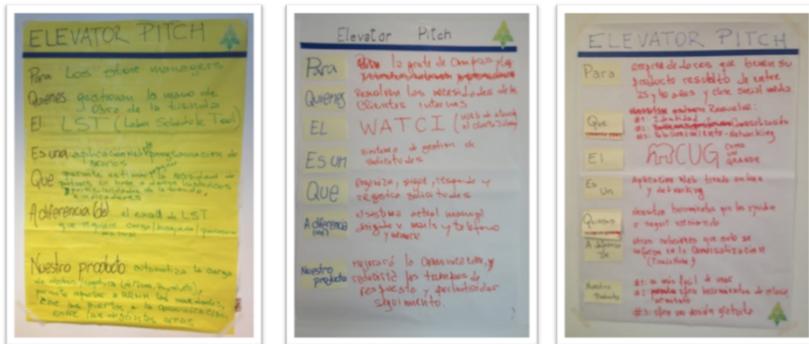
**Es un** (Tipo de aplicación o vertical)

**Qué** (Beneficios claves o razones más importantes por las que lo comprarían)

**A diferencia de** (Competidor principal)

**Nuestro Producto** (Diferencial)

Ejemplos de *Elevator Pitchs* desarrollados para algunos clientes:

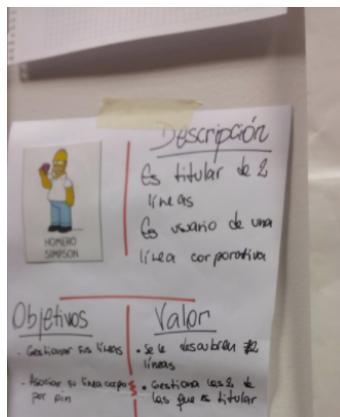


## 🔗 Personas

Uno de los objetivos que perseguimos en esta fase de *Product Discovery* implica entender a los potenciales usuarios del sistema. ¿Quiénes son? ¿Qué tipos de usuario son? ¿Cómo encontrarán valor?

Una técnica muy útil para lograr conocimiento compartido en este sentido es la descripción de *Personas* (término técnico procedente del inglés). Esta herramienta consiste en la identificación de un tipo de usuario del mundo real que intentaremos describir. Cómo es, cuáles son sus características (por ejemplo, si es un usuario básico o uno avanzado). Finalmente, describimos en pocas palabras cuál será el valor que el producto que vamos a desarrollar le brindará. Cabe destacar que esta *Persona*, que usamos como ejemplo, representa en realidad un conjunto de usuarios que comparten las mismas características y que usarán el producto de modo similar.

Este ejercicio puede realizarlo el *Product Owner* (o el *UX Designer* en proyectos que cuenten con este rol) para después informar sus conclusiones o puede efectuarse a través de un taller donde participe todo el equipo, incluyendo a los desarrolladores. ¡La diversidad de roles siempre enriquece el resultado!



### ❖ ¿Qué nos quita el sueño?

Siempre que un nuevo proyecto está por comenzar, existe un conjunto de incertidumbres o, dicho de otro modo, de riesgos que podrían hacer que el proyecto fracase. Es necesario que éstos sean identificados cuanto antes. Esta actividad es breve (se puede hacer en 15/20 minutos) y permite que todos los involucrados hagan explícitos dichos riesgos,

discutiéndose el potencial impacto de cada uno de ellos. El resultado de esta actividad será de gran ayuda en la priorización que deberá establecerse en el futuro, cuando se realice el planeamiento.

¿Cómo es la dinámica de la actividad? Básicamente, se trabaja sobre un papel de gran tamaño y *post-its* que son entregados a todos los participantes. Durante 5' cada integrante del equipo pensará y escribirá riesgos (cada riesgo en un *post-it*), pegándolos en el papel. Al finalizar este periodo, se leerán los riesgos que los integrantes del equipo han pensado y se discutirá la gravedad de cada uno de ellos, evaluándose cómo pueden ser mitigados.

## *Experimentos de usabilidad*

Una de las facetas a las que generalmente le prestamos menos atención y que, sin embargo, tiene mucha importancia es la de *Usabilidad*. Algo que muchos olvidan es que puede comenzar a evaluarse este aspecto sin escribir una sola línea de código y de manera muy económica. Aprender lo más rápido posible si los usuarios entienden la interfaz, qué *features* consideran importantes y cuáles ni siquiera notan pueden ahorrar una importante cantidad de recursos económicos.

### *Cómo realizar un test de usabilidad en cinco pasos - por Juan Manuel Carraro*

Muchas veces hay resistencias o limitaciones para realizar un test de usabilidad, sin todas las condiciones que indica la teoría. Es cierto que un test realizado por un consultor en usabilidad permitirá obtener mejores resultados, pero también es posible hacerlo internamente y hacerse de buena información. El requisito es cumplir ciertos criterios mínimos que aseguren una mayor consistencia metodológica y confiabilidad.

#### **1. Definir los objetivos del test**

El primer paso consiste en definir cuáles son los objetivos del test. Un objetivo puede ser revisar la arquitectura de la información, los rótulos de determinadas secciones o saber si el proceso de compra que tenemos montado en nuestro sitio resulta intuitivo y fácil para los usuarios.

Los objetivos son sumamente importantes porque de ellos podremos derivar las tareas que los usuarios deberán realizar al momento de hacer el test. Por ejemplo, si nuestro objetivo es testear cuán intuitivo y fácil de navegar resulta el catálogo de productos que tenemos en nuestro sitio, una tarea podría ser solicitarles a los usuarios que encuentren un producto que cumpla con determinadas características.

## 2. Construir el guión de tareas

Un guión se divide en conjuntos de preguntas y series de tareas, cada una de los cuales está vinculada a ciertos objetivos. Como mínimo debe tener lo siguiente:

**Introducción:** Aquí es donde el consultor o quien realice el test se presenta, le cuenta al usuario qué va a hacer, cuáles son los objetivos del test, etc. 2 ideas son las que tienen que quedar claras para el usuario: debe pensar en voz alta para que el consultor pueda tomar notas mientras él trabaja y debe tener en claro que no se está evaluando a la persona sino a la interfaz, por lo que los errores serán bienvenidos.

**Cuestionario previo al test:** Aquí se trata de identificar o verificar el perfil del usuario. Algunas de las preguntas que se pueden incluir son: cuántas horas al día le dedica a Internet, qué tipo de sitios visita habitualmente o si realiza compras online.

**Descripción de un escenario:** Como el contexto del test no deja de ser una situación artificial, que intenta reproducir las circunstancias reales, se le entrega al usuario información para situarlo en un determinado contexto. Por ejemplo, si estamos testeando un *Home Banking* con clientes que no pertenecen al mismo banco se les pide que actúen como si fueran clientes de la entidad y se les da información de sus cuentas, de su perfil o servicios contratados.

**Tareas:** Las tareas constituyen el eje del guión y son aquellas acciones requeridas al usuario durante el test. Algunos ejemplos: agregar un producto al carrito de compra, ingresar al sitio con usuario y contraseña, registrarse, etc. Es sumamente importante que el usuario comente en voz alta lo que hace para que el consultor pueda tomar nota de sus dudas, dificultades o errores producidos durante el proceso.

**Preguntas:** También pueden incluirse preguntas cerradas o abiertas dentro del test para conocer la opinión del usuario respecto a

determinado elemento. Esto ayuda a evaluar elementos secundarios que por sus características no pueden testearse mediante tareas.

**Cuestionario posterior al test:** Una vez que el usuario terminó de realizar todas las tareas y respondió todas las preguntas, se le solicita que exprese sus sensaciones y opiniones con respecto a la interfaz. Este segmento del guión de tareas es altamente cualitativo. Es importante incentivar a los usuarios para que se expresen y evitar el “Me pareció todo bien”.

### 3. Identificar el perfil de los usuarios

Uno de los factores que más influye en los resultados del test es que las personas que lo realicen sean lo suficientemente representativas del *target* del sitio. Para ello es importante trabajar previamente en la identificación de los rasgos y las cualidades de los usuarios teniendo en cuenta las diferentes dimensiones que componen un perfil:

**Geográfica:** Analiza la ubicación geográfica de los usuarios.

**Demográfica:** Analiza las características de los usuarios en tanto individuos sociales.

**Psicográfica:** Procura conocer a la persona entera en su interacción con el ambiente.

**Conductual:** Mide las actitudes de los usuarios hacia el consumo de un determinado producto o servicio en particular. La forma, el lugar y las ocasiones en que dicho consumo se produce.

Es incluso muy común que un sitio tenga diferentes segmentos de usuarios claramente identificables. En este caso es importante definir en primera instancia el perfil genérico y luego los perfiles particulares.

Siendo más específico, es posible vincular determinados perfiles con ciertas tareas: un perfil avanzado realizará tareas más complejas en comparación a las que un perfil novel pueda ejecutar.

### 4. Realizar los tests

Con un guión de tareas consistente, correctamente revisado y con al menos un par de usuarios de prueba, el test de usabilidad debería desarrollarse sin sobresaltos.

Es fundamental tener resueltas previamente todas las cuestiones logísticas para que no haya imprevistos durante las sesiones. Se deberá verificar que el ordenador funcione adecuadamente y tenga el prototipo o las páginas cargadas, que los dispositivos que utilicemos para grabar las sesiones (audio, video, grabadores de pantalla, etc.) funcionen bien y demás.

Cada sesión individual no debería durar más de una hora, sesiones más largas tienden a cansar al usuario y la atención decae. Normalmente es suficiente realizar un test con 5 usuarios, ya que así se identificarán cerca del 80% de los problemas de usabilidad.

Es recomendable realizar 2 o 3 sesiones en la mañana, tener un tiempo entre sesión para registrar algunas notas y luego aprovechar las tardes para pasar en limpio la información completa de cada test, mientras todavía los detalles estén dentro del alcance de nuestra memoria.

## 5. Redactar un informe

El modo de organizar el informe final con los resultados es muy personal. En general, es recomendable comenzar por lo más destacado y terminar con los detalles. Esto ayuda a que si el informe debe ser leído por alguna persona con poco tiempo, en las primeras 3 páginas pueda tener un resumen de lo más importante.

En el informe se deben clasificar los errores o problemas encontrados para luego poder priorizarlos y armar un plan de implementación de mejoras si fuese necesario. Las categorías deben ser simples y reducidas. Un ejemplo sería:

**Problemas graves:** Aquellos que impiden al usuario concluir la tarea solicitada.

**Problemas importantes:** Aquellos que permiten la conclusión de la tarea, pero con un esfuerzo mayor al normal y luego de numerosos errores.

**Problemas menores:** Aquellos que generan dudas o confusión en los usuarios, pero permiten realizar la tarea sin mucha demora.

También los errores pueden clasificarse por tipo: errores de la arquitectura de la información, del diseño o a nivel de los textos.

Algunos datos útiles para incluir son la cantidad de clics y el tiempo que lleva a los usuarios realizar cada tarea. Esta información puede ayudar luego a extraer interesantes conclusiones.

Hasta aquí se han resumido muy suavemente los pasos más importantes para realizar un test de usabilidad.

Si bien son importantes la habilidad y los conocimientos previos de quien prepare y realice el test, la experiencia ayudará a encontrar la mejor forma de conducirlo y obtener resultados cada vez más reveladores.

## Conclusiones

En este primer capítulo describí la primera fase de nuestro proceso de desarrollo: el *Product Discovery*. Todos nuestros proyectos comienzan con los talleres descritos, que **nos permiten crear una visión compartida de los objetivos y sentar las bases de una relación basada en la confianza donde fluya la comunicación**.

Todas las personas involucradas en el proyecto participan en estos talleres. Es importante escuchar todas las opiniones y saber que existe un alineamiento dentro de la empresa que pretende construir el producto. La interacción de todos estos perfiles, de negocios y técnicos, enriquece la conversación y potencia la innovación.

Realizar estos talleres en el proceso de preventa genera un grado de confianza que permite obtener fluidez, minimizando el desperdicio, ya que el desarrollo será una continuación que utilizará los mismos artefactos y se hará con las mismas personas que participaron del *Product Discovery*.

# *User Stories & Backlog*



Ya terminamos la fase de *Product Discovery*, que nos permitió tener una visión compartida acerca de lo que deseamos construir y por qué, quiénes serán los usuarios principales y cuál será el modelo de negocios. Ahora, debemos refinar nuestro entendimiento de la funcionalidad detectada, descomponiéndola en ítems o incrementos de funcionalidad que usaremos primero para planear y luego para construir el producto incrementalmente, con el involucramiento cercano de los *Product Owners*.

Kent Beck bautizó estos incrementos de funcionalidad con el nombre de *User Stories* o historias de usuario. El proceso que imaginó, y plasmó en la Metodología *eXtreme Programming*, implicaba que los usuarios pensaran la funcionalidad, la escribieran en una tarjeta, desencadenando esto el relato de la “historia” del usuario. Noten que lo importante de todo esto no es la tarjeta, sino la conversación que ésta provoca entre usuarios y desarrolladores para refinar el entendimiento de la funcionalidad.

Por una cuestión de organización y gestión, “planchamos” las *User Stories* descubiertas en una lista priorizada (que luego estimaremos) llamada *Product Backlog*. Considero este artefacto esencial para realizar el planeamiento y luego para gestionar el progreso, aunque sé, como contrapartida, que se pierde la visión completa del producto.

En este capítulo ahondaré sobre estas herramientas, describiendo sus características principales, cómo las creamos y cómo las usamos.

## ***User Stories***

### *Un poco de historia*

Durante los años 90, Kent Beck (¿quizás, la persona más influyente en el nacimiento del mundo de desarrollo ágil?) sugirió que la manera que usábamos para especificar el producto a desarrollar era tremendamente ineficiente. ¿Qué hacíamos? Una parte del equipo, conformado generalmente por los analistas, intentaba describir mediante un documento, en la fase de Análisis del Proceso del Desarrollo en Cascada, todo lo que el sistema a desarrollar debía contener. Luego, este documento era entregado a los desarrolladores para que empezaran a

trabajar a partir de él. Difícil, ¿no? Difícil especificar lo que se tiene que hacer en un documento, que este sea completo y entender qué hacer a partir del mismo. Beck propuso un cambio radical, paradigmático y revolucionario desde la simplicidad: **que los usuarios de negocio escribieran una tarjeta por cada una de las funcionalidades que deseaban ver en el sistema** y que dicha tarjeta operase como el disparador de un **conjunto de conversaciones que sirvieran para clarificar qué se deseaba** o, en otras palabras, para hacer el análisis.

Ron Jeffries capturó los componentes de las *User Stories* en la famosa fórmula de las 3C's, en inglés: *Card, Conversation & Confirmation* (en español, sería TCC por Tarjeta):

- Una tarjeta, tangible, que representa una funcionalidad.
- Una conversación que se da entre todos los integrantes del equipo, clientes, usuarios, desarrolladores, testers, etc.
- La confirmación, formal, que los objetivos han sido alcanzados.

Así nacieron lo que hoy llamamos *User Stories*, que detallaré en las secciones que siguen. Sin embargo, nunca olviden que estas toman su nombre a partir de cómo deben ser empleadas: **justamente para contar historias**.

## *Atributos de las User Stories*

Antes de contarles cómo escribir *User Stories*, les voy a contar algo que considero más importante: cuáles son los atributos que estas deben poseer. En un rapto de honestidad brutal, les diré que, si tienen estos atributos, poco me importa si usan el formato clásico de escritura de *User Stories*.

Las *User Stories* deben estar escritas en el lenguaje de los usuarios finales, deben explicar el qué (por sobre el cómo), deben contener alguna funcionalidad visible al *Product Owner* y deben ser gestionables.

Profundizo en cada uno de estos atributos:

**Escritas en el lenguaje del usuario final:** ¿Queremos entender el dominio que estamos modelando? Entonces ¿qué mejor que describir

cada una de estas funcionalidades empleando la jerga de estos usuarios en sus tareas diarias? Esto nos forzará a tener las conversaciones necesarias para entender, en profundidad, cada uno de los conceptos que estamos modelando. Por ejemplo, si estamos construyendo un software para un cajero de un supermercado, deberemos entender cómo denominan a cada uno de los productos, qué significa un arqueo de la caja y cualquier otro concepto que forme parte de su área de competencia.

**Deben explicar el qué:** Es importante en este momento que se ponga el foco en describir qué es lo que se quiere construir por sobre el cómo. Esta característica se vuelve fundamental en los casos en que las *User Stories* sean escritas sin el aporte del equipo de desarrollo. *Product Owners*: No prescindan de un aporte tan fundamental como el del área técnica para el diseño de una solución, porque se perderían de una perspectiva fundamental. Desarrolladores: Si las *User Stories* ya describen la solución, vuelvan un paso atrás y pregunten qué se quiere lograr. Mantengan su cabeza abierta para evaluar otras alternativas.

**Contener alguna funcionalidad visible al *Product Owner*.** Debe implicar una vista nueva, algún mensaje que reciba (basado en un conjunto de condiciones), el envío de un email, etc. Esta característica es fundamental, teniendo en cuenta que emplearemos un proceso de desarrollo iterativo e incremental donde el *Product Owner* estará involucrado. Hacer incrementos de funcionalidad pequeños para obtener feedback lo más rápido posible es clave. Por supuesto que, para lograr esto, cada funcionalidad que desarrollemos incluirá trabajo en cada una de las capas que formen parte del *stack* de tecnología que hayamos elegido (por ejemplo, en una arquitectura clásica contendrá funcionalidad de la vista, el modelo y la base de datos).

**Gestionables:** Queremos “partir” el trabajo en ítems relativamente pequeños, para entenderlos mejor y para reducir la incertidumbre. El tamaño de estos ítems es bastante subjetivo, ya que depende del equipo, las tecnologías y de la duración de la iteración. Para dar una idea, prefiero *User Stories* que puedan terminarse en un lapso comprendido entre 2 y 5 días. Hacer cosas más pequeñas representa demasiado *overhead* administrativo. Hacer cosas más grandes resulta peor aún, ya que el feedback se hace demasiado lento y puede no haber sensación de progreso. El equipo técnico es quien mejor conoce el esfuerzo requerido

para desarrollar una *User Story* y es el que deberá aconsejar partirla si esta fuese demasiado grande.

## *Escribiendo User Stories*

Ya sabemos que las *User Stories* sirven para contar historias. También enumeramos sus atributos más importantes. Ahora describiré el formato clásico para escribirlas. Sin embargo, no quiero que lo incorporen como una regla que no pueden romper. En mi opinión, ustedes deben buscar el mejor modo de escribirlas en su contexto particular, siempre cumpliendo con los atributos que les mencioné.

El formato clásico para las *User Stories* es:

Como <un rol>

Deseo <hacer algo>

Para <obtener algún valor>

Por ejemplo, siguiendo con nuestro cajero:

Como cajero

Deseo ingresar un producto manualmente

Para seguir con la compra en los casos en que el scanner no funcione

Este es el *template* clásico de escritura de *User Stories*. Noten que está descrita desde la perspectiva de uno de los usuarios finales y constituye alguna funcionalidad a emplear. Escribirlas de este modo nos asegura que lo desarrollado será algo visible.

La 3<sup>ra</sup> sentencia, que explica el valor, tiene como objetivo que los usuarios expliquen por qué necesitan esa funcionalidad. En mi experiencia, muchas veces se hace difícil o un poco inútil explicar la razón de una funcionalidad tan pequeña. Creo que uno explica el porqué de módulos grandes de funcionalidad y no el de sus partes.

Esta breve descripción servirá para tener una idea, superficial, de la funcionalidad esperada, aunque probablemente no contenga todos los detalles necesarios para que el equipo pueda empezar a construirla.

Tenemos que ampliar la descripción. Para tal fin, usaremos los “**criterios de aceptación**”, que **no son más que descripciones adicionales, ejemplos de uso, reglas que se deben cumplir o incluso mocks de las pantallas**. Cualquier información que ayude a entender la funcionalidad, en el mejor formato (el más claro), “sumará” dentro de los criterios de aceptación.

Para agregar un ejemplo, podemos usar este template, tan conocido para los desarrolladores que hacen tests automatizados:

Given (dado)

When (cuando)

Then (entonces)

Es decir, dado un conjunto de precondiciones, cuando ejecutamos cierta acción, deberíamos ver cierto resultado. Escribir los criterios de aceptación de esta manera permite traducirlos en tests automatizados, que pasarán a constituir “documentación viva” (como lo denominó Gojko Adzic (Adzic, 2011)).

Por ejemplo, si quisiera dar un criterio de aceptación que describa qué pasa con el cálculo de todos los ítems, podría especificarlo así:

Dado que la suma de mis ítems es 100

Y el costo de la manteca es 20

Cuando ingreso manualmente el código de la manteca

Entonces la suma de mis ítems será 120

A veces, en vez de utilizar un ejemplo, podemos especificar una regla, que nos permita entender qué hacer en cualquier situación. Imaginen que el supermercado aplica un descuento del 10% en la compra de 3 productos iguales.

El criterio de aceptación podría ser una regla escrita del siguiente modo:

Si escaneo o ingreso 3 ítems iguales => aplico 10% sobre esos productos

Así, el costo de 3 mantecas será  $3 \times 20 - 6 = \$54$

Otras veces, los criterios de aceptación no corresponden a ninguna de estas características o simplemente es mejor describirlos de otro modo. Si tuviera que especificar la funcionalidad para emitir un recibo, que consiste en enviar un conjunto de información a un controlador fiscal, la *User Story* consecuente podría plasmarse del siguiente modo:

Como cajero

Deseo emitir un recibo fiscal

Y dentro de los criterios de aceptación, debería especificar todos los datos que enviaré:

- Total
- IVA
- Descuentos

Un punto muy importante, cuando escriban los criterios de aceptación, es que **todo debería ser lo más concreto posible**, desplazarse de la subjetividad. No pueden especificar que algo debe ser rápido, o debe verse bien. ¿Qué implica que algo sea rápido? ¿Qué implica que se vea bien? Todo esto debe estar claramente definido.

Cuando estamos construyendo una aplicación que tiene una interfase visual (una *app web* o *mobile*), la mayoría de las *User Stories* tendrán asociadas una vista en la que se desarrollará esta funcionalidad. El mejor modo para describirla es a través de un *mockup*, que podría ser dibujado a mano (muchas veces lo hemos realizado junto con el usuario y luego fotografiado) o ser confeccionado en una herramienta que sirva para este fin como *balsamiq*<sup>13</sup>. Dentro de estos *mockups* podemos agregar cualquier descripción que sirva para refinar el *scope* de la funcionalidad a describir. Retomando el ejemplo del recibo de la *User Story* anterior, podríamos haberlo imaginado en un *mockup*, como el de la imagen que sigue, donde especificamos la información esperada de un modo más visual:

---

<sup>13</sup> <https://balsamiq.com/>

Autoservicio "10 Pinos"		FORMATO MONEDA
Manteca	x 3	\$60
Leche	x 1	\$30
Total		\$90
Descuentos		\$ 6
IVA		\$ 17,64
A pagar		\$ 101,64
		MOSTRAR 2 DECIMALES

Este *mockup*, precario (porque lo hice yo), tiene toda la información de los criterios mostrados en el ejemplo anterior, de modo más prolíjo y con las aclaraciones pertinentes que hacen a la *User Story* mucho más comprensible. Una imagen vale más que mil palabras. Utilicen esto a su favor.

Por supuesto que las ***User Stories*** no tienen que ser la única herramienta de documentación/especificación disponible. A continuación, detallo otras que me han resultado muy útiles:

**Workflow de estados:** En muchas herramientas que he construido, hemos necesitado describir un *workflow* de estados para algunas de las entidades que modelamos. Hacerlo a través de las *User Stories* no hubiera tenido sentido. Es mucho mejor describirlo a través de un gráfico, que contenga los diferentes estados y sus transiciones.

**Flujo de Navegación de Pantallas:** Podríamos decir que se trata de un caso particular del anterior. Para visualizar la navegación entre las

diferentes vistas, lo que hemos hecho es imprimir las pantallas (una en cada hoja) para luego vincularlas con flechas, describiendo las situaciones que causan los cambios de pantalla (Algunas herramientas permiten crear una aplicación para navegar entre las diferentes vistas y testear la usabilidad del producto de modo realista).

Estos son ejemplos de herramientas adicionales que usamos para pensar y describir la funcionalidad que deseamos construir y que luego podemos vincular desde las *User Stories*.

Un punto muy importante que quiero mencionarles: como en el caso del código, la duplicación es mala (“La raíz de todos los males en el software”, dice “Uncle Bob”<sup>14</sup>). Si tuviéramos especificaciones duplicadas, tendríamos una carga mayor de trabajo para mantenerlas actualizadas y correríamos adicionalmente el riesgo de no entender cuál es la última versión.

Me gustaría cerrar esta sección diciéndoles que, como en cualquier otra parte de su Metodología, en la escritura de las *User Stories* **deben ser críticos para refinar el proceso** y encontrar el modo más claro y eficiente.

## Proceso de descubrimiento

Paso a contarles cuál es el proceso de descubrimiento y refinamiento de las *User Stories*. El resultado del *Product Discovery* nos brinda los artefactos que nos sirven como base. En particular, el *User Story Mapping* es muy útil. Partiendo de las tareas que cada uno de los roles “hace” en el sistema inferimos la funcionalidad necesaria para escribirlas. Ya vimos el caso donde, para la tarea de escaneo manual que habíamos detectado, creamos una *User Story* para detallarla con mayor profundidad. También habíamos descubierto que el cajero podía tomar un “pago con tarjeta de crédito”. Seguramente, será mejor construir una funcionalidad tan compleja usando múltiples *User Stories*, de manera de poder incluir al *Product Owner* durante la construcción y medir el progreso intermedio.

---

<sup>14</sup> “The root of all evils in software in software design” (Martin, 2008)

En este trabajo de descubrimiento inicial, en el que vamos identificando las *User Stories* e incluyéndolas en un *Backlog* inicial (artefacto del que hablaré en breve), no debemos preocuparnos por entender su *scope* exacto, ya que iremos refinándolas a medida que avancemos en la construcción del producto. El objetivo no es comenzar el desarrollo inmediatamente, sino armar un *Backlog* que describa la funcionalidad más importante, priorizarlo y hacer una estimación que sea usada, posteriormente, al efectuar la planificación del proyecto.

## 🔗 Example Mapping

Descubrí esta técnica en la conferencia XP 2016 y la he usado en un par de ocasiones para escribir *User Stories* junto a todo el equipo. Su creador es Matt Wynn<sup>15</sup>.

La idea de esta herramienta consiste en estructurar la conversación que se da al intentar escribirlas, partiendo de la base de que todas tienen un título, un conjunto de reglas, un conjunto de ejemplos que ayudan a clarificar esas reglas y un conjunto de preguntas abiertas. Utilizamos tarjetas de diferentes colores para cada uno de estos componentes:

**Título:** amarillas

**Regla:** azules

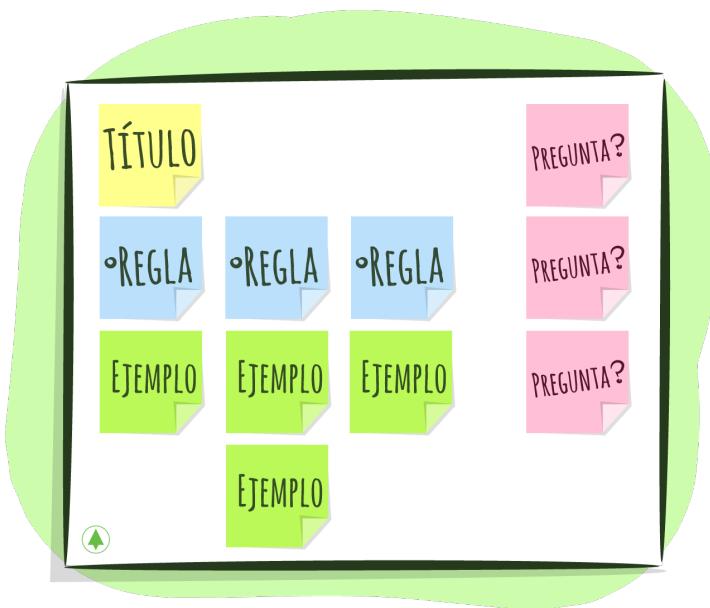
**Ejemplos:** verdes

**Preguntas:** rojas

Por cada *User Story*, se construirá un “mapa” usando estas tarjetas, como muestra la imagen que sigue:

---

<sup>15</sup> <https://cucumber.io/blog/example-mapping-introduction/>



Noten que los colores tienen el objetivo de poder visualizar ciertos *smells* en las *stories*:

Muchas tarjetas rojas nos dicen que aún tenemos mucho que aprender.

Muchas tarjetas azules nos dicen que es grande y complicada y quizás convenga partirla.

Muchas tarjetas verdes para una regla nos dicen que quizás se trate de más de una.

En las ocasiones en las que facilité esta actividad, me resultó útil bosquejar las pantallas en la cuáles sucedían estas funcionalidades. Creo que una tarjeta para poder hacer esto sería una buena adición a la herramienta.

Esta técnica sirve para depurar las *User Stories* de una manera estructurada y efectiva. Si lo hace todo el equipo, repartiéndoselas y luego presentándolas, la actividad puede ser muy enriquecedora.

## Partiendo *User Stories*

Muchos equipos que comienzan a trabajar con *User Stories* encuentran incómodo trabajar en *slices* verticales tan pequeños. Suelen decirme vehementemente que tenga en cuenta que involucra trabajo en todas las capas, que no es eficiente trabajar así. Puede llegar a sonar ineficiente, pero puedo convencerlos de lo contrario. Es posible trabajar en incrementos pequeños de funcionalidad eficientemente, usando buenas prácticas de desarrollo de software, como *Test Driven Development* y *Continuous Refactor*.

Lo que tenemos por ganar es mucho:

- Recibir el feedback de los usuarios del negocio rápidamente.
- Medir el progreso del proyecto en base a funcionalidad terminada.
- Hacer *Releases* pequeños.

Podemos tener en cuenta estas maneras de “partir” las stories cuando nos parezcan demasiado grandes:

**Partir usando agrupaciones lógicas:** Por ejemplo, si incluyera el alta de una persona y de sus datos de facturación, podríamos crear una para el alta de la persona y otra para el alta de sus datos de facturación.

**Separar el manejo de las condiciones excepcionales:** Escribir una para el “camino feliz” (sin tener en cuenta las condiciones de error) y otra para manejar los casos que puedan fallar.

**Separar las distintas operaciones:** Por ejemplo, hacer una para cada operación de un ABM.

**Separar desde la *GUI*:** Es decir, hacer una que contenga la funcionalidad, sin los detalles de diseño, y otra que contemple el diseño.

**Hacer un *Spike*:** Cuando existe demasiada incertidumbre (y digo demasiada, porque siempre debemos resolver algo), podemos hacer un *Spike* que sirva para investigar y, posteriormente, la *User Story* para hacer el trabajo propiamente dicho.

## El *Backlog*



Cuando empezamos a construir el *Backlog*, nos focalizamos en descubrir todo el trabajo que tenemos por delante, sin ir a los detalles, haciendo un barrido horizontal de todo el producto y procurando identificar toda su funcionalidad. Hacemos, en este momento también, un trabajo importante de priorización: ¿cuáles son las *User Stories* que forman el *core*? ¿Qué es lo más importante para el negocio? ¿Qué es lo más riesgoso? ¿Cuáles son las *User Stories* que debemos construir para edificar la arquitectura de la aplicación?

Las *User Stories* del *Backlog* inicial no están refinadas, es decir, no está escrito el detalle de cada una de ellas. Antes de empezar, de la primera

iteración, debemos tener un conjunto de *User Stories* mínimo que habilite un flujo continuo de trabajo del equipo de desarrollo. Este proceso de refinamiento se realiza empezando por el tope del *Backlog*, por las *User Stories* prioritarias. ¿Cuántas refinar? No existe un máximo estipulado, pero tengan en cuenta que no deseamos refinárlas si no estamos seguros de “consumirlas” luego.

El *Backlog* no es un artefacto estático, es un artefacto que muta en el transcurso del proyecto. Se modifica para reflejar, en todo momento, nuestro entendimiento del producto en proceso de construcción.

Todo el equipo trabaja activamente en el *Backlog*, pero es el *Product Owner*, la persona que mejor conoce el negocio, quien debe encargarse de gestionarlo de modo prolíjo. Él procura que las *User Stories* estén bien escritas y de priorizarlas. Muchas veces, a partir de conversaciones mantenidas entre los miembros del equipo o de feedback externo, pueden surgir cambios en funcionalidad que inicialmente se había pensado, que deben reflejarse prolíjamente en el *Backlog* a través del *Product Owner*.

## *¿Qué es el Backlog de Producto?*

Formalizando, el *Backlog* es una lista priorizada y posiblemente estimada de las *User Stories* del proyecto. Lo utilizamos para almacenar, comunicar, compartir y principalmente gestionar todo el conocimiento que adquirimos durante el transcurso del proyecto, a través de las *User Stories*. Es, además, una herramienta de gestión fundamental durante la construcción del sistema que nos permite visualizar qué *User Stories* fueron terminadas, están en desarrollo o quedan por hacer.

## *Atributos importantes*

Repasemos los atributos más importantes de este artefacto:

**Es un artefacto “vivo”:** Su vida comienza en la fase de descubrimiento y planeamiento del *Release* con un conjunto de *User Stories*, de las que se posee información superficial. Durante la fase de construcción, cada una de estas se refinará, es decir, se profundizará el conocimiento del *scope* hasta contener toda la información necesaria para comenzar a desarrollar.

Además, seguramente se descubrirán otras, ya que es imposible detectar toda la funcionalidad de un producto en un momento inicial. ¡Nuestra cabeza no funciona así! Necesita retroalimentación. Observar funcionalidad dispara nuevas ideas, que deben ser plasmadas en nuevas *User Stories*. También es posible que otras resulten modificadas, repriorizadas o, incluso, eliminadas. El trabajo en el *Backlog* se extiende durante el transcurso del proyecto.

**Contiene ítems con niveles de refinamiento diferentes:** Es decir, con diferentes niveles de entendimiento. Encontraremos *User Stories* cuyo *scope* esté completamente definido y otras que serán simplemente ideas. También encontraremos algunas pequeñas, estimables y otras de gran tamaño, comúnmente llamadas épicas, conviviendo dentro del mismo *Backlog*.

**Es liviano:** Es muy fácil agregar, modificar, borrar y priorizar *User Stories*. Tiene que serlo, ya que el trabajo que se hará sobre él será intensivo.

## *Herramientas digitales*

Creo que es fundamental contar con una herramienta digital para la correcta gestión del *Backlog*, tanto para su conformación, como para la posterior administración del proyecto.

En 10Pines trabajamos con clientes que utilizan diferentes herramientas como Jira<sup>16</sup>, VersionOne<sup>17</sup> o Pivotal Tracker<sup>18</sup>. Otros clientes tienen su propia herramienta ágil que se adapta exactamente a su metodología, como 8thlight<sup>19</sup> con Artisan<sup>20</sup>.

---

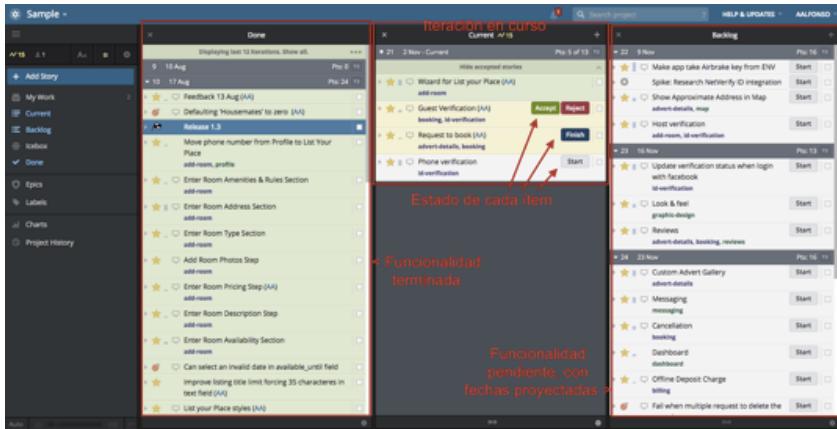
<sup>16</sup> <https://www.atlassian.com/software/jira>

<sup>17</sup> <https://www.versionone.com/>

<sup>18</sup> <https://www.pivotaltracker.com/>

<sup>19</sup> <https://8thlight.com/>

<sup>20</sup> <https://artisan.8thlight.com/>



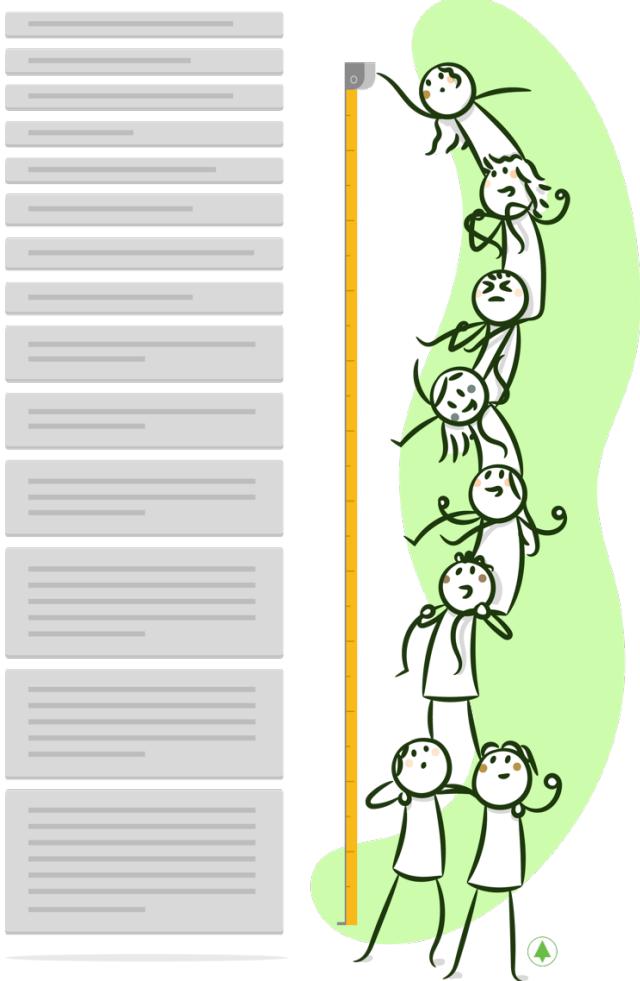
Mi herramienta preferida es Jira porque permite visualizar claramente el *Backlog*, las *User Stories* y los *story boards*. Además, el *workflow* es configurable y soporta *Scrum* y *Kanban*. Como desventaja, contiene demasiada información en cada vista, lo que dificulta ubicar lo realmente necesario.

## Conclusiones

Descomponemos el trabajo en un conjunto de incrementos funcionales pequeños y visibles al *Product Owner*, para poder involucrarlo en el desarrollo y brindar feedback tempranamente. Hacer los incrementos pequeños permite reducir la incertidumbre y medir el progreso frecuentemente. Llamamos a estos incrementos *User Stories*. Conversar con los usuarios para que nos “cuenten” sus historias resulta esencial. La información procedente de este intercambio no puede compararse con ninguna otra escrita.

Usamos un *Backlog* para gestionar las *User Stories* y facilitar las conversaciones. Se trata de un artefacto “vivo”, que el equipo consultará y modificará durante el transcurso del proyecto.

# Estimaciones



## Una primera noción del tamaño

Una vez que tenemos esta primera versión del *Backlog*, necesitaremos saber cuán grande es el producto a construir, para tener un estimado del costo y también para planear un equipo que, teniendo en cuenta las restricciones de tiempo y dinero, lo desarrolle.

**Hacemos la estimación inicial de la manera más rápida y eficiente posible**, sin perder tiempo en adivinanzas y especulaciones, usando la técnica de *Planning Poker*<sup>21</sup> para asignar *Story Points* a cada una de las *User Stories* descubiertas. Noten que estimamos incrementos de funcionalidad visibles. Esto es fundamental para la posterior gestión del proyecto, ya que medimos el progreso del mismo en base a funcionalidad terminada.

Deberemos arriesgar el tiempo que involucraría 1 *Story Point*, para inferir cuánto nos llevaría completar todo el proyecto. Esta primera estimación carga con mucha incertidumbre, ya que se hace con poca información (tanto del negocio, como de la parte técnica, posiblemente). Usaremos esta estimación para construir el plan inicial, pero será fundamental durante las primeras semanas entender mejor la dimensión del producto y también la capacidad real del equipo (*Velocity*).

He mencionado unos cuantos conceptos sobre los que, desde mi punto de vista, vale la pena profundizar. En las secciones que siguen, les contaré más sobre estimaciones relativas, *Story Points*, *Planning Poker*, la *Velocity* y sobre cómo usamos estos conceptos para el planeamiento y la gestión del proyecto.

## Estimaciones relativas usando *Story Points*

Durante muchos años de mi vida realicé estimaciones en tiempos absolutos. Para hacerlas, desagregaba los componentes que, entendía, debían desarrollarse y asignaba tiempos (que luego sumaba) para cada uno de ellos. Usé puntos de función y complejas planillas de cálculo

---

<sup>21</sup> [https://es.wikipedia.org/wiki/Planning\\_poker](https://es.wikipedia.org/wiki/Planning_poker)

creadas dentro del marco de estándares de calidad como *CMMI*<sup>22</sup> (que procuraban tener métodos estandarizados para la organización). Ninguna de estas técnicas resultó, en mi experiencia, en estimaciones precisas, generando siempre comportamientos disfuncionales que terminaron siendo perjudiciales para la organización. Un ejemplo que todos conocerán es el que consta en agregar un “padding” a la estimación, para “cumplir” con los tiempos. ¿Les parece que tiene sentido?

Cuando empecé a trabajar con *Scrum*, descubrí el concepto de estimaciones relativas usando *Story Points*. Con esta técnica, no estimamos cuánto esfuerzo demanda completar una *User Story*, sino **cuánto demanda una con respecto a otra**. Asignamos puntos, que llamamos *Story Points*, como resultado de estas comparaciones relativas. Así, una *User Story* que tiene asignado 2 requerirá el doble de esfuerzo que otra de 1 y 2/3 de una tercera de 3. **Estos puntos amalgaman todos los factores que pueden influir en el esfuerzo**, entre los que puedo nombrarles (Cohn, Agile Estimating and Planning, 2005):

- **Cantidad de Trabajo:** ¿Es un formulario con 3 o 10 campos? ¿Involucra mucho testing?
- **Incertidumbre:** ¿Conocemos el negocio? ¿Las funcionalidades están cerradas o hay puntos abiertos? ¿Es una tecnología que conocemos o es nueva y no tenemos experiencia? ¿Tenemos que interactuar con servicios desconocidos?
- **Complejidad:** Volviendo al ejemplo del formulario, ¿los componentes usados son sencillos? ¿Involucran validaciones? ¿Están interrelacionados?

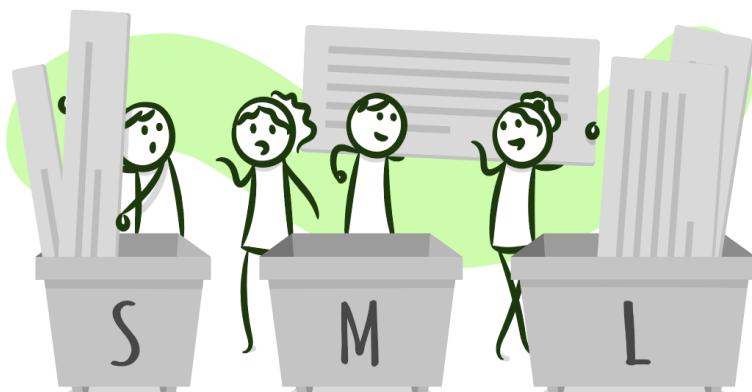
Los valores que pueden ser usados como *Story Points* pertenecen a una escala discreta, por ejemplo, la sucesión de Fibonacci. **Limitarnos a valores discretos simplifica notablemente el proceso.** Cuando estimamos de esta manera, podemos imaginar que tenemos baldes que representan cada uno de los valores y luego decidimos a cuál asignar cada *User Story*. De este modo, evitamos discusiones que aportan muy poco valor, como si debiéramos asignar 2.5 o 2.7 ¡No disponemos de tanta

---

<sup>22</sup> [https://es.wikipedia.org/wiki/Capability\\_Maturity\\_Model\\_Integration](https://es.wikipedia.org/wiki/Capability_Maturity_Model_Integration)

información como para ser tan precisos! Los valores discretos incorporan intrínsecamente este grado de incertidumbre. La sucesión de Fibonacci es particularmente intuitiva en este sentido, ya que las distancias entre los números de la sucesión crecen a medida que los números se vuelven más grandes.

Si bien es bastante frecuente usar *Story Points*, éstos no son fundamentales dentro del concepto de estimaciones relativas. Pueden usar tales de remera, como en el gráfico que se encuentra abajo, o cualquier otra medida. El punto es que cada número, por sí mismo, carece de sentido. Toma sentido cuando se lo ve en comparación con otro.



¿Por qué usamos estimaciones relativas? La primera razón es que **vuelven el proceso más sencillo y liviano**, ya que no nos detenemos en tratar de adivinar lo que no conocemos o en investigaciones que haremos cuando empecemos a trabajar. Además, las estimaciones relativas nos desplazan de razonamientos tales como “esta *User Story* la desarrollará una persona determinada por sus *skills* o *seniority*” que no tienen ningún sentido. Simplemente decidimos, como equipo, cuanto más grande o más chica es una *User Story* con respecto a otra. La segunda razón es que realizar las estimaciones de esta manera nos protege de cualquier intento de presión (por parte de jefes/*project managers*, etc.) a través de la existencia de este nivel nuevo de indirección. Estas presiones siempre

generan, como ya mencioné, conductas que no son beneficiosas para nadie.

## ¿Qué se tiene en cuenta al realizar una estimación?

Un punto a tener en claro, antes de estimar, es qué significa completar una *User Story*. No basta con haber “codeado” la funcionalidad. Debemos, como mínimo, ejecutar exitosamente todos los casos de prueba y el *Product Owner* debe validar y aceptar la *User Story*. Además, deben existir *tests* automatizados para toda la nueva funcionalidad, que cobrarán vital importancia para su posterior regresión.

Todos estos puntos, y quizás algunos otros que se agreguen para algún proyecto u organización en particular, constituyen la *definition of done*<sup>23</sup>. No tiene sentido escribirlos en cada *User Story* porque están implícitos, aplican a todas. **Es muy importante que todo el equipo entienda lo mismo.** En otras palabras, que para todas y todos signifique lo mismo completar una *User Story*. Una actividad que podemos realizar, para asegurarnos de ésto, consiste en armar un afiche con la lista de los criterios definidos. Una vez creado, podemos pegarlo cerca de nuestros escritorios, para que funcione como un radiador de información<sup>24</sup>.

### ❖ Planning Poker

Usamos esta técnica para asignar estimaciones a las *User Stories*. Para ello, invitamos a todo el equipo que participará del proyecto (desarrolladores, testers, diseñadores y *Product Owners*), entregándoles mazos similares al que se encuentra abajo, donde cada una de las cartas representa un número de *Story Points*:

---

<sup>23</sup> <https://www.scruminc.com/definition-of-done/>

<sup>24</sup> <https://www.agilealliance.org/glossary/information-radiators/>



La explicación del funcionamiento de esta técnica yace en la teoría de la sabiduría de las masas<sup>25</sup>, que argumenta que **la participación de todo el equipo incrementa la precisión de las estimaciones** al escuchar las perspectivas de todos los integrantes. Que una persona sola, por más que sea un experto, realice todas las estimaciones implica un riesgo mucho mayor ya que es muy probable que omita factores poco visibles en su rol.

El proceso para hacer las estimaciones con esta técnica es el siguiente:

1. El *Product Owner* describe la *User Story*, qué desea lograr y cómo aportará valor.
2. Se debate brevemente. ¿Qué implica? ¿Qué riesgos existen? Escuchar las diferentes perspectivas es fundamental para enriquecer las nuestras.
3. Se deja un momento para que cada persona piense y elija su carta.
4. A la cuenta de 3, todos al mismo tiempo mostramos la carta seleccionada: ¡No queremos que nadie se sienta influenciado!
5. Contamos cuántas personas eligieron cada carta. En general, la distribución es una campana de *Gauss*, ya que la mayoría suele elegir un valor y sólo unos pocos eligen uno superior o inferior.

---

<sup>25</sup> [https://es.wikipedia.org/wiki/Sabidur%C3%ADa\\_de\\_los\\_grupos](https://es.wikipedia.org/wiki/Sabidur%C3%ADa_de_los_grupos)

6. Es bueno escuchar a los *outliers*, es decir, aquellas personas que tienen estimaciones muy bajas o altas ya que pueden tener información que el resto desconoce u omite. Por supuesto que también pudo desconocerse o malinterpretarse algún punto de la *User Story*. De igual modo, resulta relevante escuchar sus argumentos y, en base a ellos, decidir si se desea repetir la estimación.
7. ¡Siempre llegamos a un consenso! Al menos, es mi experiencia. Que los valores sean discretos ayuda mucho. Después de todo, sólo estamos decidiendo cuántos *Story Points* asignar a una *User Story* y no cuántos días vamos a tardar en completarla.

## Estimando el *Backlog*

Para finalizar esta sección, les contaré el proceso que usamos para estimar un conjunto de *User Stories*, que podría llegar a ser el *Backlog* completo o sólo el próximo *Release* (personalmente no creo que valga la pena hacer una estimación de más de 2 meses de trabajo).

Empezamos por seleccionar una *User Story* que esté entre las prioritarias y que consideremos de las más pequeñas del *Backlog* y le asignamos 1 punto. Luego estimamos, usando *Planning Poker*, la primera *User Story*, es decir la que está en el primer lugar del *Backlog*, estableciendo una comparación con la precedente a la que habíamos asignado 1 punto.

Una vez establecida una estimación para la segunda *User Story*, seguiremos con la tercera, la cuarta y así sucesivamente. Durante los comienzos, puede llegar a surgir la necesidad de algunos ajustes, pero puedo asegurarles que el sistema se estabiliza muy rápidamente y estas primeras estimaciones establecen el parámetro para todos las *User Stories* que restan en el *Backlog* (y para todas las que se descubran en el futuro).

Como ya lo mencioné, **no creo que tenga sentido invertir una cantidad exagerada de tiempo en esta actividad**. Si ya definimos el Producto Mínimo Viable (hablaremos sobre este concepto que llamamos *MVP* en la próxima sección), nos limitaremos a estimar solamente sus *User Stories*. Si el *MVP* fuera demasiado grande, sería bueno partirlo en *releases*. Debe tenerse en cuenta que, mientras más *User Stories* estimemos, mayor será el riesgo de perder el tiempo en ítems que después no sean

construidos. **Tampoco vale la pena invertir tiempo en entender cada *User Story* en profundidad.** Una técnica que empleo habitualmente, para no extenderme, consiste en delimitar el tiempo que se va a usar para la estimación de cada *User Story* (5 minutos puede ser adecuado). Otra opción consiste en, de ser el equipo numeroso, repartir las *User Stories* entre 2 grupos para que puedan hacer las estimaciones en paralelo dando lugar a la exposición de los resultados.

## Costo y tiempo de trabajo

Para comenzar un proyecto, se debe tener una idea de la dimensión del producto a construir, principalmente para conocer el costo y para poder planificar en base a él.

Para hacer la traducción de los *Story Points* del *Backlog* a una unidad de tiempo, simplemente estimamos 1 *Story Point* con el equipo que elegimos infiriendo, de este modo, el tiempo total del *Backlog*.

Visualicemos esto mediante un ejemplo. Imaginen que estimamos nuestro *Backlog* inicial en 75 *Story Points* y pensamos en un equipo de 3 personas para desarrollarlo. El paso siguiente sería estimar cuánto nos llevaría completar 1 *Story Point* con este equipo.

Por cuestiones de simplicidad, podríamos estimarlo en 1 día. Es decir que proyectamos completar 1 *Story Point* por día.

De este modo, demoraríamos 75 días en terminar o alrededor de 8 iteraciones de 2 semanas (10 puntos por iteración). Podríamos incluso inferir, siendo que tenemos nuestro *Backlog* priorizado, qué funcionalidades entregariamos en cada una de las iteraciones para planificar de acuerdo con esto. No es una actividad en la que encuentre demasiado valor, por lo que no la fomento.

Si quisieramos hacer un presupuesto, deberíamos multiplicar este número por la cantidad de personas en el equipo y éste, a su vez, por la cantidad de horas que, en promedio, trabajan esas personas.

En nuestro ejemplo:

$$75 \text{ días} \times 3 \text{ personas} \times 7 \text{ horas promedio/día} = 1575 \text{ horas}$$

Si el costo es u\$s 60/hora, el presupuesto para el proyecto será:  
 $1575 \times 60 = \text{u\$s } 94500$

Podríamos pensar que existe una similitud entre las horas descritas y las horas-hombre<sup>26</sup> de la gestión tradicional de proyectos, pero no son exactamente lo mismo. La diferencia, sutil, radica en que las estimaciones para cada una de las *User Stories* se basan en un equipo, mientras que para la gestión tradicional de proyectos se trata de estimaciones del tiempo que demora un “recurso” en completar una actividad.

## ***Velocity y Burndown***

Al comenzar a trabajar, **mediremos la cantidad de *Story Points* que podemos terminar en cada iteración**, lo que nos dará una pauta de la verdadera capacidad del equipo. **Llamamos a ésto *Velocity***<sup>27</sup>, que es un término que proviene de la física y que incluye la celeridad y la dirección.

Midiendo la *Velocity* de la primera iteración, podríamos inferir, con más información que antes de empezar, cuánto nos queda por delante. Sigamos con el ejemplo de la sección anterior e imaginemos que el equipo completa 2 *User Stories*: una de 5 puntos y otra de 3. La *Velocity* será entonces de 8 puntos. De los 75 puntos estimados, completamos 8 (en la jerga decimos “quemamos 8” y ya veremos la causa), o sea que nos quedan 68. Podemos inferir entonces que, si seguimos con esta *Velocity*, necesitaríamos 8.5 iteraciones ( $68 / 8$ ) para terminar.

Al completar la 2<sup>da</sup> iteración y medir la *Velocity*, dispondremos de más información, ya que podremos obtener un promedio de las *velocities* de las 2 primeras iteraciones. Por ejemplo, si hubiera sido igual a 6, el promedio de las 2 iteraciones sería igual a 7, lo que nos indica que necesitaríamos 8.8 iteraciones más para terminar. Para visualizar la *Velocity*, podemos usar el gráfico de *burndown*:

---

<sup>26</sup> <https://es.wikipedia.org/wiki/Hora-persona>

<sup>27</sup> <https://www.agilealliance.org/glossary/velocity>



En este gráfico, el eje “Y” indica la cantidad de puntos por quemar y el “X” muestra el tiempo.

En este ejemplo, sobreestimamos nuestra capacidad (o subestimamos la complejidad del proyecto). **Con esta información, debemos actualizar nuestro plan:** ¿Podemos extender la fecha de entrega? ¿Podemos acotar el *scope*, dejando algunas funcionalidades para un *release* posterior? ¿Podríamos aumentar la *Velocity* sumando un integrante más? La confianza y la colaboración existentes entre los miembros del equipo permiten dar lugar a estas conversaciones.

**Mike Cohn dice que la *Velocity* es el gran ecualizador, ya que nos permite actualizar nuestro plan automáticamente.** Mientras que las estimaciones relativas sean consistentes, la medición de la *Velocity* nos permitirá inferir un calendario actualizado sin necesidad de efectuar ningún tipo de re-estimación.

## Aspectos importantes en las estimaciones ágiles

Me gustaría contártelos, para finalizar, cuáles son los puntos fundamentales del enfoque ágil en mi opinión. No son los *Story Points*, ni el *planning poker*, sino el proceso que usamos para estimar. En resumen:

- **Se hacen sobre *User Stories*,** es decir, sobre incrementos de funcionalidad visibles al usuario final. El progreso, posteriormente, será medido en base a esto.
- **No se invierte mucho tiempo en hacer un análisis detallado** para hacer la estimación porque la precisión que presumiblemente se podría alcanzar no justifica la inversión. Ésto no significa que las estimaciones dejen de hacerse por completo ya que es necesario tener una noción del tamaño, para hacer la planificación.
- **No implican un compromiso.** La presión ejercida sobre el equipo, ya sea para acortar los tiempos o para cumplir con la fecha inicialmente estimada, es la principal causa de disfuncionalidad que conozco. Queremos evitar que el equipo sienta la necesidad de poner un *pad* a la estimación o quiera cortar camino para cumplir con las estimaciones y consecuentemente agregue deuda técnica. Ninguno de estos comportamientos resulta beneficioso.

## Conclusión

En esta sección estudiamos cómo hacer una estimación inicial del producto. Necesitamos tener una idea de la dimensión del mismo para poder planificar el equipo y proyectar tiempos (de acuerdo a los *deadlines* y expectativas del negocio). No deseamos, ni podemos, saber exactamente cuánto llevará. Tampoco, invertir demasiado tiempo en intentarlo, sin enfrentarnos con los problemas prácticos. No lo considero un buen uso del tiempo. Asumir un compromiso con esta información tampoco me parece razonable y es causa de muchas disfuncionalidades. Cuando comencemos a desarrollar, podremos verificar si las asunciones y estimaciones resultaron correctas, plasmando el conocimiento adquirido en nuestros planes. Usando *Story Points* y midiendo la *Velocity*, ésto es automático.

# Priorización



## Saber qué es lo importante es lo importante

Un consultor amigo siempre decía que las metodologías ágiles no son más que un compendio de prácticas para reducir el riesgo. En este sentido, desarrollar las funcionalidades más importantes primero es fundamental, ya que muy probablemente el dinero no alcance para todas las que el cliente desea. Al usar un proceso de desarrollo iterativo e incremental, podemos hacer *Releases* tempranos que las implementen, dejando el resto para otros posteriores. De esta manera, optimizamos el retorno de inversión y minimizamos el riesgo del proyecto.

¿Cuál es la funcionalidad más importante? El *Product Owner* y todas las personas vinculadas al negocio, seguramente, conocen mejor la funcionalidad que aporta más valor. En otras palabras, la que generará mayores ingresos, resolverá más problemas u optimizará los procesos más importantes. Este no es el único factor que debemos tener en cuenta al priorizar. Otro, sumamente importante, es el riesgo tecnológico. Debemos “atacar” rápidamente todas las áreas tecnológicas que provoquen mayor incertidumbre, ya que, si no pudiésemos resolver los problemas del modo imaginado, esto podría provocar que el producto final no aportase el valor de negocio inicialmente esperado. Reducir el riesgo tecnológico también es un factor preponderante a la hora de priorizar.

Es importante que todo el equipo colabore en la priorización del producto. El *Product Owner* será, sin embargo, el actor más importante, quien determine lo que el negocio espera. Por su parte, el equipo técnico debe colaborar activamente, identificando los riesgos tecnológicos y las dependencias. La priorización, así como el diseño, debe ser el resultado de la colaboración del equipo en pos de un objetivo compartido.

El trabajo de priorización no concluye con la escritura del *Backlog* preliminar. Desde el inicio de su construcción, se genera un grado de conocimiento que puede impactar en la prioridad de las funcionalidades planeadas. También pueden tener lugar eventos externos (como un anuncio de un competidor o un cambio en un proveedor) que modifiquen nuestro plan. El *Backlog* será la herramienta utilizada para

reflejar todas las decisiones de priorización tomadas durante el transcurso del proyecto.

En este capítulo, trataremos principalmente la priorización dentro del contexto de desarrollo de un producto nuevo, transmitiendo los factores a tener en cuenta y algunas herramientas utilizadas.

## ¿Qué debe tenerse en cuenta al priorizar?

El factor más importante es el valor de negocio, que, en su acepción más sencilla, implica cuánto dinero ganaremos. Sin embargo, este podría no llegar a ser tan directo. Existen otras aristas que influyen, como ganar nuevos clientes o una porción de mercado donde no existan otros competidores. En definitiva, **el valor de negocio que una funcionalidad entrega es el beneficio que la organización obtiene a partir de él**. Nuestro objetivo debe ser maximizarlo.

El 2<sup>do</sup> factor a tener en cuenta es el riesgo tecnológico. Si existieran componentes, *frameworks* o servicios que no sabemos como funcionan, es importante despejar esta incertidumbre lo más rápido posible. No deseamos progresar en el desarrollo del producto para luego darnos cuenta de que algo no va a funcionar y que, en consecuencia, el valor entregado sea inferior al esperado. En el mundo ágil dicen falla rápido (“*fail fast*”) o, para decirlo en otras palabras tomadas de Alistair Cockburn, aprende rápido (“*learn fast*”).

Otra perspectiva desde la que podemos mirar este tópico de priorización es la del aprendizaje. Estamos aprendiendo y debemos lidiar con riesgos e incertidumbres. En consecuencia, deberíamos priorizar todo aquello que nos permita aprender.

Otro factor que podríamos llegar a tener en cuenta, aunque de menor importancia, es el esfuerzo (o costo). Conociéndolo, el *Product Owner* podrá decidir que una funcionalidad es demasiado costosa para el valor que arroja o, por el contrario, construir otra menos valiosa, solamente porque el esfuerzo requerido es menor. Para ponerlo en términos formales, la razón costo-beneficio puede influir en la priorización.

Ninguno de estos factores decide la priorización por sí mismo, ya que tanto los de negocios, como los tecnológicos, deben ser tenidos en cuenta. Como ya mencioné anteriormente, las decisiones de priorización serán el resultado de la colaboración entre todas y todos.

## ***Releases cortos***

Antes de interiorizarnos en muchas de las herramientas utilizadas, les mostraré la heurística más importante a la hora de priorizar y de planear. Acoten, tanto como sea posible, la cantidad de trabajo. En otras palabras, en lugar de hacer *releases* grandes y riesgosos, háganlos simples y breves. Les resumo las ventajas que obtendrán:

- **Maximiza el retorno de inversión del proyecto:** Desde el punto de vista financiero, no hay dudas de que es sumamente beneficioso, ya que, en un periodo de tiempo mucho menor, obtendremos parte del valor de negocio prometido (que puede ser usado para financiar el resto del proyecto).
- **Minimiza el riesgo:** Tener mucho trabajo en progreso representa un riesgo, porque las condiciones (internas o externas) pueden cambiar. Al construir el MVP, debemos acotar este *release* al mínimo posible, para testear la hipótesis única de valor (que tiene un riesgo muy grande) al menor costo posible.
- **Minimiza la incertidumbre:** No olvidemos que, antes de construir el producto y testearlo con usuarios reales, solo hipotetizamos.

Verán menciones a esta heurística en toda la literatura ágil. Kent Beck incluyó, entre las reglas de planeamiento de *eXtreme Programming*, una que denominó “releases cortos y frecuentes<sup>28</sup>” (Beck, Planning eXtreme Programming (The Xp Series), 2000). Probablemente fue quien luego propició uno de los principios ágiles<sup>29</sup> que afirma: “buscamos satisfacer a los clientes a través de la entrega continua de software con valor”. La declaración de interdependencia<sup>30</sup>, escrita por *Project Managers*, específica

---

<sup>28</sup> <http://www.extremeprogramming.org/rules/releaseoften.html>

<sup>29</sup> <https://agilemanifesto.org/iso/es/principles.html>

<sup>30</sup> <http://pmdoi.org/>

que “se aumenta el retorno de inversión, haciendo foco en tener un flujo continuo de valor”. Las reglas de Lean Software Development<sup>31</sup> (Poppendieck00, 2003) van en la misma dirección. Una de ellas proclama: “debemos entregar software rápidamente, ya que implica una ventaja competitiva”. Finalmente, Kanban (Anderson, 2010) (Kniberg, 2011) hace de todo esto su *leit motiv*. Sus reglas principales comprenden “visualizar el flujo de valor”, para “limitar el trabajo en progreso” y, de esta manera, “maximizarlo”.

En estos días, muchos de nuestros clientes, que ya poseen un producto en funcionamiento, usan *continuous delivery*<sup>32</sup>. El precepto es el mismo. De este modo, agregan valor diariamente o, incluso, varias veces en un día. Los *releases* pequeños y frecuentes de los que hablaba Beck en los años 90 se llevaron al extremo.

Ahora sí, veamos algunas de las herramientas que usamos frecuente en distintas etapas del proceso de priorización.

## Herramientas

Utilizo frecuentemente las herramientas que a continuación describiré porque permiten visualizar y facilitar el proceso de priorización. Además, resultan simples para entender y livianas de implementar. Existen otras más sofisticadas, por ejemplo, las financieras, que calculan el retorno de inversión, sobre las que no profundizaré aquí.

Empezaré por el *User Story Map*, detallando su empleo a la hora de priorizar, durante el *Product Discovery*. Seguiré con una herramienta extremadamente simple, pero muy útil al mismo tiempo, llamada *MoSCoW*. A continuación, les mostraré otras, pensadas para la visualización de la razón costo-beneficio y finalmente terminaré con otra, llamada *product roadmap*, muy útil a la hora de facilitar la priorización de funcionalidades nuevas para un producto ya implementado.

---

<sup>31</sup> [https://en.wikipedia.org/wiki/Lean\\_software\\_development](https://en.wikipedia.org/wiki/Lean_software_development)

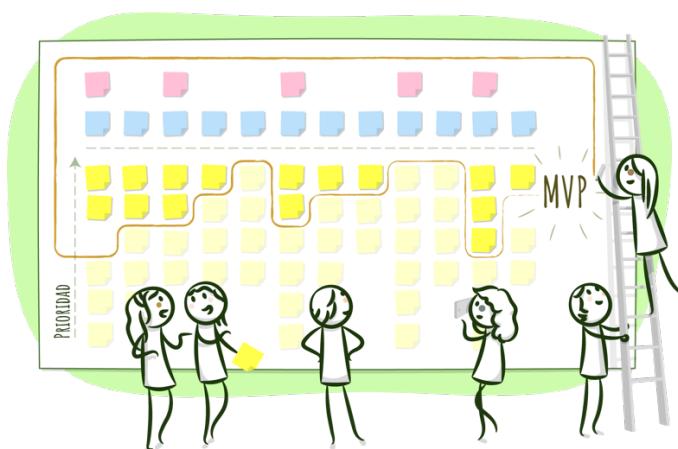
<sup>32</sup> <https://continuousdelivery.com/>

## 🔑 User Story Mapping

Esta herramienta nos permite visualizar el producto completo rápidamente, resultando de gran utilidad para facilitar la priorización. Lo usamos durante las sesiones de *Product Discovery*, ya que posibilita la comprensión del orden de importancia de las tareas detectadas y también el conjunto mínimo necesario para una versión inicial.

Como vimos en el capítulo 1, la manera de indicar que una tarea resulta más importante que otra consiste en situarla sobre la de menor importancia. Esta decisión siempre es precedida por interesantes discusiones plasmadas, de modo visual, en el *Story Map*.

El *User Story Map* es una herramienta muy útil también para decidir cuál será el MVP, es decir, el producto que nos permite testear la hipótesis única de valor. Simplemente trazamos una línea por debajo de todas las tareas incluidas en él. Aquellas, situadas por sobre ésta, representan el MVP. Frecuentemente, vemos una gran cantidad de *post-it*s por sobre la línea, lo que nos permite comprender, instantáneamente, que el MVP seleccionado es demasiado grande. En esos casos, procuramos mover la línea hacia arriba. Como ya mencioné en la sección anterior, prefiero *releases* cortos.



Soy un fan de esta herramienta. Todas las conversaciones empleadas en la toma de estas decisiones tienen lugar frente al “mapa” del producto, quedando inmediatamente plasmadas.

## MoSCoW

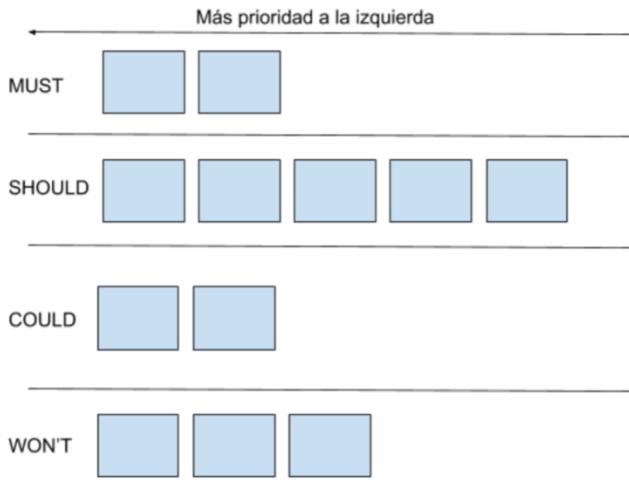
Esta técnica de priorización es la más sencilla de todas. Consiste en categorizar las *User Stories* o las épicas según su importancia. Las categorías, cuyas iniciales dan nombre a la técnica, son:

- *Must*: Las obligatorias.
- *Should*: Deberían estar.
- *Could*: Podrían estar.
- *Won't*: No van a estar.

Pese a su simpleza, esta herramienta resulta sumamente útil, ya que nos obliga a consensuar, entre todo el equipo, una categoría para cada uno de los ítems.

Podemos usarla para priorizar un conjunto de *User Stories* o épicas de un producto nuevo y también funcionalidades que deseamos agregar a uno en funcionamiento. Un ejemplo muy interesante de esta última situación tuvo lugar dentro de un equipo con el que colaboré: 3 *Product Owners* debían priorizar las épicas del bimestre venidero. Resultó muy interesante observar como reconocían la imposibilidad para completar el trabajo, al visualizar la gran cantidad de épicas categorizadas en “*Must*”, y decidían las que podían posponerse. Como ya les mencioné, la utilidad de la herramienta yace en que **obliga a un conjunto de personas a consensuar categorías para un conjunto de ítems de un modo muy visual**.

Con este equipo, solíamos priorizar también las épicas de cada categoría, sobre todo, las de la categoría “*Must*”. Así, lográbamos categorizar las épicas por bimestre, así como también conocer el orden de las que desarrollaríamos próximamente.



### ⌚ Puntos de valor de negocio

Otra herramienta de gran utilidad durante el proceso de priorización es la asignación de “puntos de valor de negocio” (*business value points*) a las diferentes funcionalidades, haciendo una comparación similar a la utilizada al estimar *Story Points*, pero, esta vez, **asignando valores relativos en lugar de esfuerzos relativos**. Las piezas de funcionalidad a las que asignamos puntos deben ser necesariamente mayores que las *User Stories*, ya que estas últimas, por sí mismas, son unidades funcionales muy pequeñas para representar un valor de negocio.

Como en la técnica de *MoSCoW*, también nos fuerza a consensuar y explicitar valores. Decimos que una épica es más importante que otra, pero ¿cuánto? ¿estamos de acuerdo en eso? Explicitar esos puntos de valor de negocio puede resultar un ejercicio muy fructífero. Además, una vez estimados el valor y el esfuerzo, podemos establecer la razón costo-beneficio de cada una de ellas.

Existen varios modos de visualización, que transmitiré mediante el ejemplo del sistema para el cajero de supermercados del capítulo 1. Imaginen que necesitamos agregar las épicas que presento a

continuación, a las cuales asignaremos los siguientes puntos de valor de negocio y esfuerzo. Obtendremos algo similar a la tabla siguiente:

Épica	Valor de Negocio	Puntos de Esfuerzo	Razón (Beneficio)
Pago con <i>Bitcoins</i>	2	1	2
Pago con Tarjeta de Descuentos	5	3	1.66
Reportes en tiempo real para Jefes	8	5	1.6

La 4<sup>ta</sup> columna es la razón entre el valor de negocio y los puntos de esfuerzo, es decir, la división entre el valor de la 2<sup>da</sup> columna y de la 3<sup>ra</sup>. Aquí se puede ver como la épica más beneficiosa es “Pago con *Bitcoins*”, con un beneficio de 2. El valor de negocio es el más pequeño, pero el esfuerzo también. El *Product Owner* podría decidir priorizar “la fruta que cuelga baja en el árbol” (*low hanging fruit*<sup>33</sup>). Paradójicamente, la épica menos beneficiosa es la de mayor valor de negocio, justamente porque implica un esfuerzo importante.

En mi experiencia, el costo es un factor sustancialmente menos importante que los demás (valor de negocio y riesgo). En contadas ocasiones observé a un *Product Owner* tomar la decisión de priorizar una funcionalidad solamente porque su desarrollo no fuera costoso. Para resolver esto, podemos añadir “pesos” para cada uno de los factores. Siguiendo el ejemplo anterior, si especificáramos un peso de 0.7 para el valor de negocio y 0.3 para el esfuerzo, obtendríamos los siguientes resultados.

---

<sup>33</sup> <https://www.merriam-webster.com/dictionary/low-hanging fruit>

Épica	Valor de Negocio (0.7)	Puntos de Esfuerzo (0.3)	Razón (Beneficio)
<b>Pago con Bitcoins</b>	2 [1.4]	1 [0.3]	4.66
<b>Pago con Tarjeta de Descuentos</b>	5 [3.5]	3 [0.9]	3.88
<b>Reportes en tiempo real para Jefes</b>	8 [5.6]	3 [0.9]	6.22

Jim Highsmith argumenta la visualización de los beneficios como porcentajes ya que esto prevendría que, al trabajar con diferentes equipos, se exageren los puntos de valor (Highsmith, 2009). Los resultados para el ejemplo serían los que presento a continuación, donde calculamos el porcentaje de valor para cada una de las épicas, el porcentaje de esfuerzo y luego la razón.

Épica	Valor de Negocio (0.7)	% de Valor	Puntos de Esfuerzo (0.3)	% de Esfuerzo	Razón (Beneficio)
<b>Pago con Bitcoins</b>	2	0.13	1	0.14	0.92
<b>Pago con Tarjeta de Descuentos</b>	5	0.33	3	0.42	0.78
<b>Reportes en tiempo real para Jefes</b>	8	0.53	3	0.42	1.26
<b>Total</b>	15		7		

Todas las técnicas mostradas son similares, ya que nos fuerzan a explicitar valores y esfuerzos y nos devuelven un indicio de lo que podría ser la funcionalidad de mayor beneficio. Pueden incluso agregarse columnas, modificando consecuentemente la fórmula, para incorporar otros factores, por ejemplo, una penalidad (si la épica no fuera desarrollada) o el riesgo (como en el ejemplo del *paper* de Karl Wiegers<sup>34</sup>). En definitiva, el valor de la herramienta consiste en que fuerza a volcar toda la subjetividad existente en nuestras cabezas sobre una tabla con números explícitos, que calculan un beneficio a través de una fórmula acordada.

## Tracer Bullets

Cuando empezamos con un producto nuevo, el equipo técnico debe pensar cómo encarar su desarrollo, es decir, cómo construir el producto mínimo viable iterativa e incrementalmente y cómo testear que la arquitectura y tecnologías elegidas soporten los requerimientos funcionales y no funcionales. En esta fase de la priorización, de más bajo nivel, nos centramos en atacar los riesgos e incertidumbres y en dividir el trabajo de la mejor manera.

Con este objetivo, elegimos un conjunto de *User Stories* que nos permitan construir el esqueleto de la aplicación, los componentes que representan la arquitectura de la misma. En esta etapa, ponemos énfasis en la parte iterativa del proceso de desarrollo. Nos enfocamos en la obtención de un esqueleto visible que pruebe los componentes principales y dejamos los incrementos para más adelante. Esta heurística es lo que Andy Hunt y Dave Thomas llaman una Bala Trazadora o *Tracer Bullet*<sup>35</sup>.

Esta *Tracer Bullet* permite testear todas las capas de nuestra arquitectura y crear una columna vertebral a partir de la cual hacer incrementos de funcionalidad. Además, nos permite mostrar un avance al usuario rápidamente (aunque no tenga la calidad final) y también correr tests de carga/*stress* que validen la arquitectura elegida.

<sup>34</sup> <https://www.processimpact.com/articles/prioritizing.pdf>

<sup>35</sup> <https://www.artima.com/intv/tracer.html>

## 🔗 Product Roadmap

Esta es una herramienta utilizada por una experimentada *Product Owner* con la que tuve la oportunidad de trabajar, para priorizar y organizar el trabajo de los próximos meses para los diferentes grupos de los que formaba parte. Después de conocerla, leí el libro *Managing your Project Portfolio* (Rothman, 2009), que la explica detalladamente.

Al observar la imagen siguiente, probablemente alcancen a comprender su funcionamiento. En un eje, se especifican las unidades de tiempo y, en el otro, los diferentes equipos. Luego grafican el periodo que, estiman (de modo superficial), consumirá cada una de las épicas que tienen priorizadas en la columna correspondiente al equipo al que se asignará.

	11/3/2019	18/3/2019	25/3/2019	1/4/2019	8/4/2019	15/4/2019	22/4/2019	29/4/2019	6/5/2019	13/5/2019	20/5/2019	27/5/2019	3/6/2019	10/6/2019
Juan, Mica y Juli	Reportes Managers de Tienda							Cambio Usabilidad Cajeros						
Martín, Matías y Lalo	Pago con tarjeta de descuentos	Pago con cheques de viajero	Pago con Mercado Pago					Reportes por email/whatsapp						
Maggie, Nahuel	Pago con Bitcoins			Migración nueva factura fiscal				Devolución Items Rápida						

La única herramienta digital que conozco, sin haber utilizado, es *Roadmunk*<sup>36</sup>. Lo bueno de que sea un *plugin* de *Jira* reside en que las mismas épicas planeadas en el *roadmap* son usadas posteriormente en la herramienta de gestión, una vez iniciada la construcción.

## 🔗 Buy a Feature – por Germán Gaitan

En ocasiones, el producto que estamos desarrollando impacta en diferentes áreas de la organización, cuyos objetivos y expectativas son diferentes (y hasta contrapuestos!). En estos casos, si los *stakeholders*, que representan estas áreas, no logran establecer un consenso acerca del valor de las funcionalidades, podemos recurrir a otras técnicas que nos permitan realizar la priorización de una manera subjetiva, pero aún colaborativa, como es el caso de *Buy a Feature*<sup>37</sup>:

Cada *stakeholder* escribe las funcionalidades de su interés en tarjetas, como si se tratara de productos de góndola. Incluso pueden destinarse algunos

<sup>36</sup> <https://roadmunk.com/jira-roadmap-integration>

<sup>37</sup> *Buy a Feature*: Compre una funcionalidad

minutos a darles un aspecto más atractivo o *marketinero* (utilizando *stickers*, colores, etc.). Una vez preparadas, las tarjetas se colocan sobre una mesa donde todas quepan de forma bien visible y se dan unos minutos para que todos las puedan leer y debatir acerca de su significado.

Entregamos a cada *stakeholder* una cantidad equitativa de fichas de póker, dinero de fantasía o hasta porotos. En mi opinión lo mejor es utilizar fichas de póker -cuanto más pesadas, mejor- ya que brindan una sensación de valor genuino. En general se da a todos la misma cantidad de fichas (5 por ronda, por ejemplo), pero pueden entregarse cantidades diferentes en función de su jerarquía en la organización o de algún otro atributo que determine lo que vale su voto.

Comienza la primera ronda. Cada *stakeholder* “gasta” sus fichas repartiéndolas entre las funcionalidades que son de su interés. Pueden aplicarse algunas restricciones, como que no deben colocarse todas las fichas en la misma funcionalidad o alguna otra variante que asegure un voto diversificado. Al término de esta fase, se cuentan las fichas de cada una. La más votada es retirada de la mesa, seleccionada como prioritaria. Si dos funcionalidades comparten el primer lugar, se espera a la siguiente ronda para definir.

La actividad se realiza en varias rondas, entregando siempre las mismas cantidades de fichas. Las tarjetas de funcionalidad que permanecieron en la mesa conservan las fichas que vienen acumulando de las rondas anteriores. En cada ronda se retira la tarjeta con más fichas y se coloca debajo de la seleccionada en la ronda anterior en orden de prioridad.

En general se realizan tres o cuatro rondas de compra de funcionalidades. Todo depende de la cantidad de participantes y de la cantidad de fichas de las que cada uno dispone. Lo que indica que ya se iteró suficiente es que, al finalizar una ronda, quedan sobre la mesa varias tarjetas con muchos votos, que pueden ser priorizadas con las fichas que sumaron hasta ese momento.

Esta experiencia suele ser divertida y muy distendida. Favorece las charlas de negociación de funcionalidades entre *stakeholders* en cada ronda. Su resultado es una priorización rápida y con altísima aceptación.

## Priorizando experimentos

Cuando hablamos de valor de negocio en esta etapa, en la que todavía no existe un producto, en realidad hipotetizamos sobre el potencial que, creemos, entregará. Nos queda validar estas hipótesis con usuarios reales, ¿estarán dispuestos a pagar por nuestro producto?

Haremos esto mediante experimentos, que testean si lo supuesto se cumple. Teniendo esto en cuenta, la priorización consiste en determinar el modo de testeo de las hipótesis. En otras palabras, pensar y priorizar experimentos y, de acuerdo con ésto, decidir qué funcionalidad incluir en cada uno de ellos. De este modo, el producto mínimo viable (*MVP*) no es más que el primero de nuestros experimentos, que valida la hipótesis única de valor.

Todos estos conceptos están tomados de la metodología de *Lean Startups*, de Eric Ries (Ries, 2011). Planear, construir y validar con usuarios reales. Los lineamientos generales son muy similares a los de las metodologías ágiles, pero van un paso más allá.

Uno de los puntos fundamentales de esta metodología consiste en definir cómo vamos a medir el funcionamiento del experimento. De haber realizado el *Lean Canvas*, mencionado en el capítulo 1, recordarán que una de las secciones consistía en describir explícitamente estos experimentos. En muchos de los desarrollos en los que trabajé, no existía certidumbre acerca de cómo se validaría lo construido. Sin embargo, esto es fundamental, ¡debemos saber lo que se quiere lograr!

Cheryl Quirion compartía estos experimentos visualmente con todo el equipo de desarrollo, usando una técnica que denominó *Lean Visual Strategy*<sup>38</sup>. Compartir esta información con el equipo permite que esté alineado y entienda por qué lo está construyendo. Hacerlo de manera visual ayuda a que todas y todos incorporen esta información mejor y más rápidamente.

---

<sup>38</sup> <https://leanvisualstrategy.com/what-is-lean-visual-strategy-4b4dc9df610d>

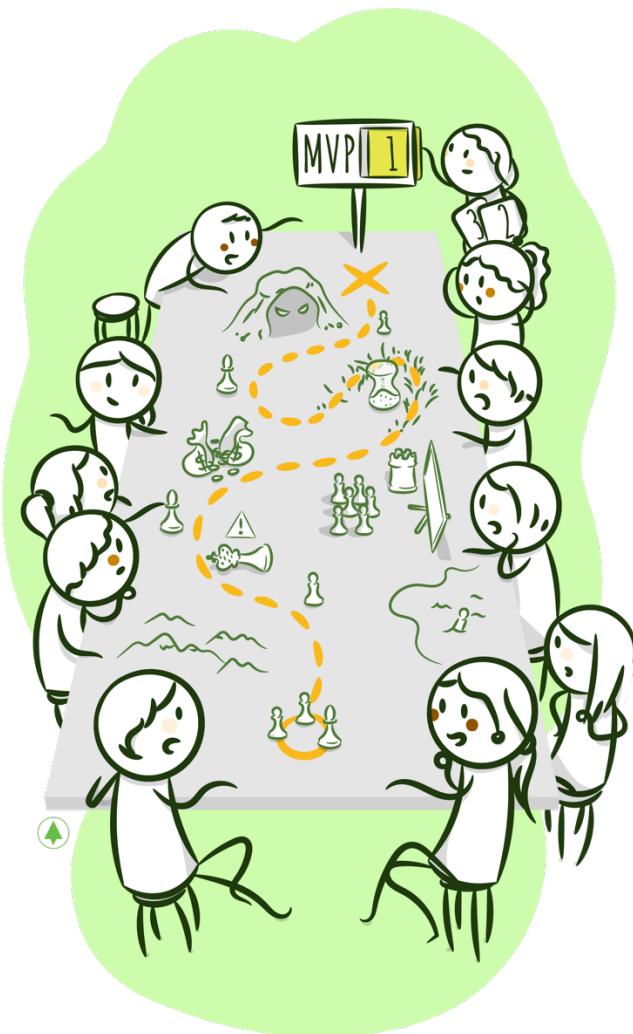
## Conclusiones

Realizar una buena priorización antes de empezar con el desarrollo es fundamental. Empezamos por determinar la funcionalidad necesaria para realizar el primer experimento, llamado *MVP*. Luego, el orden en que lo construiremos de manera de minimizar el riesgo y maximizar el valor.

Este camino que trazamos es creado colaborativamente y compartido por todas y todos. Las herramientas que mencioné contribuyen en el proceso. Una vez iniciada la construcción resulta fundamental mantener un *Backlog* actualizado. Queremos estar seguros de estar trabajando, en todo momento, sobre las funcionalidades más importantes, es decir, las que aporten más valor y reduzcan más el riesgo.



# Planeamiento Continuo



## Introducción

Ya hicimos el *Product Discovery* y descubrimos qué se quiere construir y cómo brindará valor el producto. Entendemos mejor qué es lo más importante y también qué se considera indispensable para la primera versión. Escribimos las *User Stories* y las estimamos. Tenemos una idea vaga, por supuesto, del tamaño. **Resta ahora pensar cuál es la mejor manera de construirlo, dadas las restricciones existentes**, que pueden ser de tiempo (por ejemplo, ¿es necesario que el *MVP* esté en el mercado para cierta fecha?) o de dinero (los clientes cuentan con un presupuesto).

Es momento para definir un plan, una primera versión de él, que esboce el camino para alcanzar nuestros objetivos. Retomando lo dicho por Marty Cagan: "para que un producto sea exitoso, debe ser **valioso, usable y factible de desarrollar**". Aquí debemos pensar la última parte: ¿Podemos construirlo en los tiempos necesarios y con los recursos con los que contamos? ¿Cómo lo haremos? ¿Qué tecnología usaremos? ¿Quiénes trabajarán en la construcción? ¿Por cuánto tiempo? ¿Cómo atacaremos los riesgos identificados? ¿Cuál será el costo? El plan representa una primera versión de las respuestas a todas estas preguntas y la visión conjunta necesaria, de todo el equipo, para alcanzar estos objetivos.

Un proyecto, según la definición de Johana Rothman en *Manage It!* (Rothman, *Manage It!: Your Guide to Modern, Pragmatic Project Management*, 2007): "es un esfuerzo novedoso para crear un nuevo producto o servicio, cuya entrega señala la finalización. Los proyectos implican riesgos y generalmente poseen recursos limitados". Tiene un *driver* (o un conjunto de *drivers*), una duración específica, marcada por el momento en que se alcanzan los objetivos, y restricciones de tiempo/dinero que los *Project Managers* deben gestionar. El plan define el curso de acción del mismo.

Haremos un plan para marcar el rumbo, pero no para seguirlo en detalle, ya que no es nuestro objetivo. Deseamos construir el mejor producto posible. En el libro *Planning eXtreme Programming*, Beck y Fowler (Beck, *Planning eXtreme Programming (The Xp Series)*, 2000) metaforizan los cambios necesarios durante la ejecución de un proyecto con las "constantes correcciones a realizar en el volante, cuando conducimos

para mantener el recorrido”. Nuestro plan se ajustará, ágilmente, con los cambios surgidos durante la construcción y, principalmente, con todo el aprendizaje que ésta dispare.

En este capítulo hablaremos sobre planeamiento: qué tienen nuestros planes y cuáles son sus características principales. Luego, evaluaremos en detalle las aristas principales englobadas en la gestión de un proyecto. A continuación, detallaré algunas herramientas que permiten visualizar el progreso del mismo. Finalmente, nos ocuparemos de algunas situaciones frecuentes que atentan contra los planes y su ejecución.

## ¿Qué contiene un plan?

Componentes esenciales:

**La visión de negocios:** ¿A dónde queremos llegar? ¿Cuáles son los objetivos principales? Todas y todos debemos compartirlos. En consecuencia, deben quedar claramente plasmados. Haberlos obtenido a través de las sesiones colaborativas del *Product Discovery* simplifica esto enormemente.

**El Backlog:** el *Backlog* inicial, que contiene *User Stories* sin mucho detalle. Representa el entendimiento colectivo de cómo va a descomponerse el trabajo en incrementos de funcionalidad que después nos permitan trabajar con un proceso de desarrollo iterativo e incremental. Podemos también tener identificadas las *User Stories* que tienen que completarse para el *MVP* (de hecho, lo mejor es que el plan corresponda a este grupo solamente).

**El equipo:** definido en base a las tecnologías escogidas, al tamaño de lo que se quiere construir y a las restricciones de tiempo y dinero. **Será multidisciplinario**, es decir, contará con todos los *skills* necesarios para hacer un incremento de funcionalidad y contará con **personas asignadas full-time** al proyecto.

**El schedule inicial:** En base a las estimaciones realizadas y a la *Velocity* prevista, se puede fijar un *schedule* inicial. Podrían existir *deadlines* fijados por el negocio (por ejemplo, pensando en una presentación o en la competencia). Estos deben figurar visibles en el plan, ya que deberemos ajustar otras aristas si la *Velocity* no fuera la esperada.

**Consideraciones tecnológicas:** En el plan, incluimos la visión tecnológica del producto, es decir, el *stack* de tecnología, las opciones para *hosting*, las herramientas principales, etc.

## *Atributos*

Repasemos los atributos principales de nuestros planes:

**Está basado en funcionalidades:** El atributo más importante. El plan es, en síntesis, un conjunto priorizado de funcionalidades. Está pensado para trabajar iterativa e incrementalmente. El progreso se irá midiendo en base a los incrementos finalizados.

**Es un artefacto liviano:** Contiene un bosquejo del camino creado a partir de la información de la que disponemos hasta el momento. No se hace futurología ni se desperdicia tiempo en detalles que podrían ser decididos responsablemente más tarde. Hacer esto volvería el plan más pesado, quitaría tiempo para empezar a construir y daría una falsa sensación de seguridad.

**Es fácil de cambiar:** Está pensado para “abrazar el cambio”. Creamos el plan partiendo de la incertidumbre. Aprenderemos muchísimo durante el camino. Todo este nuevo conocimiento deberá plasmarse en el plan.

**Es corto:** No creo en planes que duren muchos meses.

**Es creíble:** He visto *Project Managers* crear planes con *deadlines* y *milestones*, para luego transmitir al equipo el “cronograma”. ¿Qué sentido tiene esto? ¿Podrías, como desarrollador, comprometerte con algo que no creaste y con lo que no crees? El plan debe ser producto de una visión conjunta consensuada.

**Visible a todo el equipo:** Claramente, si fue creado y consensuado por todas y todos, el plan es accesible en todo momento. ¿Podrías alcanzar las metas propuestas, si el equipo no dispone de toda la información?

**No es un artefacto estático:** Este plan inicial, esbozado a partir del *Product Discovery* y que contiene mucha incertidumbre aún, se refinará a medida que aprendamos: ¿Es factible tecnológicamente? ¿Podemos construir el producto a la *Velocity* que habíamos estimado previamente? Este nuevo conocimiento, de negocios, tecnológico y del equipo,

generado en la etapa de construcción, debe plasmarse en el plan, ya que sería una pena no usarlo a nuestro favor. Por las razones previamente mencionadas, Mike Cohn denominó a su libro “*Estimating & Planning*” (Cohn, Agile Estimating and Planning, 2005). No es un plan. Es planeamiento, continuo. Y Jim Highsmith (Highsmith, 2009) llamó a esta etapa “Especulación”. Es un nombre muy acertado, ya que, en cualquier plan, existe incertidumbre. “Cuando especulamos, establecemos un objetivo y una dirección, pero, al mismo tiempo, esperamos cambios en el camino”.

## *Aristas*

Hablemos por unos minutos como *project managers*, para entender las variables que llevan en sus cabezas cuando gestionan un proyecto. En este plan definimos 3 aristas:

- **Scope:** El alcance. Los planes ágiles se basan en un conjunto de funcionalidad priorizada.
- **Recursos:** Humanos y monetarios. Estas restricciones determinarán lo que podamos lograr.
- **Tiempos:** El calendario tentativo, incluyendo *milestones* y *deadlines*, que deben ser tenidos en cuenta para alcanzar los objetivos.

Existe una 4<sup>ta</sup> arista, la de calidad, que no se especifica en el plan, sino que es el resultado de todos estos factores.

Estas aristas están vinculadas, unas con otras, en modos no lineales. Si una de ellas se modifica, alguna de las otras deberá modificarse también. Por ejemplo, cuando se descubre nueva funcionalidad (*scope creep*, decían en mi época), el proyecto implicará más tiempo. Podríamos intentar sumar “recursos” para aumentar la *Velocity*, pero los “recursos” son personas, que deben ser entrenadas, con el impacto consecuente sobre el proyecto (como enunció Fred Brooks en su famosa Ley de Brooks<sup>39</sup>). Si

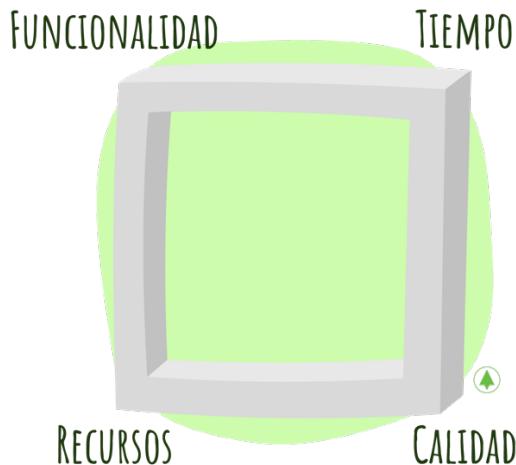
<sup>39</sup> [https://en.wikipedia.org/wiki/Brooks's\\_law](https://en.wikipedia.org/wiki/Brooks's_law)

no quisiéramos correr la fecha y no sumáramos “recursos”, la calidad seguramente se vería afectada.

En mis “años mozos”, conocí el Triángulo de Acero de la Gestión de Proyectos<sup>40</sup>, un modelo que permitía visualizar estas restricciones: si se “tira” de una de las aristas, alguna de las otras deberá moverse forzosamente.



En el libro de *Management 3.0* (Appelo, 2010), Jurgen describe el Cuadrado de Acero, incluyendo en la 4<sup>ta</sup> arista la calidad.



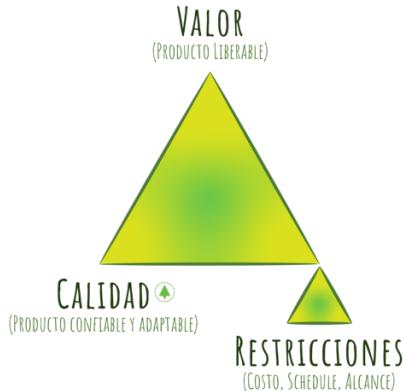
---

<sup>40</sup> [https://en.wikipedia.org/wiki/Project\\_management\\_triangle](https://en.wikipedia.org/wiki/Project_management_triangle)

La idea del cuadrado de acero es que la modificación de una de las aristas en una dirección tiene un efecto similar en alguna de las aristas adyacentes o el efecto contrario en la opuesta. Por ejemplo: ampliar la funcionalidad implica más recursos o extender la fecha o una calidad inferior (interna y/o externa). Una pérdida de recursos llevaría a tener que incluir menos funcionalidad, bajar la calidad o extender el tiempo.

El *Project Manager* tiene como objetivo gestionar estas aristas para que el proyecto sea exitoso. La pregunta es: **¿Qué significa que el proyecto sea exitoso?** El Standish Group popularizó una métrica errónea, basándose en el cumplimiento de las aristas del triángulo de acero. Así, un proyecto resultaba exitoso en caso de completar el *scope* planeado a tiempo y con los recursos estimados. Aprendí, a través de Jim Highsmith, que debemos tener en cuenta los factores de negocio al evaluar el resultado ¿Cuánto valor de negocio entregamos? ¿Resolvimos los problemas de los usuarios? ¿Es usable? Las decisiones que tomemos, durante la ejecución del proyecto, deben estar guiadas por estas preguntas. Nuestro objetivo debe ser la construcción de un producto que genere la mayor cantidad de valor de negocio y que sea usable, cumpliendo con las restricciones, es decir, dentro de los tiempos y el presupuesto con el que se cuenta. Con este objetivo en mente, el plan es otro artefacto que nos ayudará a ir decidiendo y visualizando cuál es la mejor manera de lograrlo.

Highsmith sugirió modificar el triángulo de acero para hacerlo compatible con los valores ágiles. Si el objetivo es maximizar el valor de negocio, este debía estar incluido en el triángulo:



Como pueden ver, en una de las aristas se encuentra el valor, en forma de un producto implementable, la 2<sup>da</sup> equivale a la calidad interna, que permite al producto ser confiable y adaptable, y finalmente la 3<sup>ta</sup>, conformada por las restricciones (costo, calendario y *scope*).

Mike Cottmeyer escribió un post<sup>41</sup> en el que coincide en mover el foco de las restricciones al valor que el producto puede generar, pero no cree que el triángulo sugerido represente correctamente las relaciones entre las diferentes aristas, como sí lo hace el triángulo de acero. Sugiere incluir la calidad y el valor como parte del *scope*, es decir, de la *Definition of Done*. ¿Estaría algo terminado si el *Product Owner* no aceptara su valor o si su calidad fuera pobre?



---

<sup>41</sup> <https://www.leadingagile.com/2010/01/replacing-the-iron-triangle-of-project-management/>

Personalmente, no logro visualizar las relaciones del triángulo original en ninguna de estas últimas variantes. Creo que entregar la funcionalidad imaginada, con la calidad esperada, es lo que permite generar el beneficio planeado. Si no cumplimos con el presupuesto, no podremos entregar nada. Y si no lo hacemos dentro de los tiempos esperados, otro lo hará. El valor de negocio es un *tradeoff* complejo entre todas estas aristas y por eso no creo que pueda ser ubicado dentro de ellas.

Para finalizar, detallaré las heurísticas que sigo cuando gestione un proyecto. El costo es una restricción fija, depende del presupuesto con el que contamos. No es una arista que se pueda modificar. Tampoco lo es la calidad interna. Creo que, en cualquier emprendimiento tecnológico, el *codebase* debe mantenerse en buena forma, bien diseñado y testeado. De otra manera, no será ni confiable ni extendible. Restan el tiempo y la funcionalidad, aristas que iremos moviendo con *sliders*, para obtener el mejor producto posible en tiempos razonables.

## Midiendo la *Velocity* y replaneando

¿Cómo medimos el progreso del plan? Sabemos que, al esbozarlo, se efectuaron muchas especulaciones. Al empezar a desarrollar, habiendo priorizado las áreas con mayor valor e incertidumbre, aprenderemos del producto y del proyecto. Usaremos todo este conocimiento adquirido para “replanear”. Esta es la base del planeamiento ágil: es continuo y se extiende a lo largo de todo el desarrollo del proyecto. No seguir un plan, sino ir descubriendo lo mejor para el producto y, en base a ésto, modificarlo.

Como ya vimos, todo el conocimiento ganado del producto lo reflejamos en el *Backlog*, donde re-priorizamos, agregamos y borramos *User Stories*. Veamos ahora algunas herramientas que sirven para visualizar el progreso.

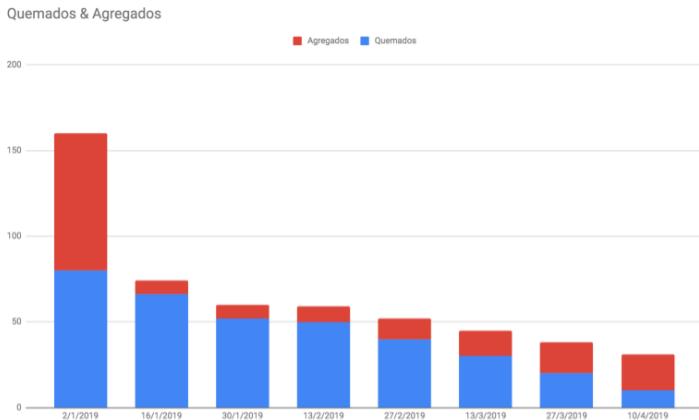
### *Burndown Chart*

Este gráfico nos sirve para visualizar la pendiente de progreso del equipo, es decir, para determinar la cantidad de trabajo restante y, en base a la *Velocity*, estimar su finalización.

Simplemente graficamos una línea que represente la cantidad de puntos por “quemar”. Esta cantidad está dada por la suma de los puntos de las *User Stories* que restan por hacer. Cada vez que finalizamos una historia, “quemamos” sus puntos y actualizamos la pendiente. De esta manera, podemos calcular la pendiente y deducir la fecha de finalización estimada. Al momento de esbozar el plan, estimamos una pendiente (y una fecha de finalización acorde) que luego podremos contrastar con el progreso real.



Noten que, al graficar los *Story Points* remanentes, no podemos distinguir los “quemados” de los agregados, es decir, los correspondientes a nuevas *User Stories*. En otras palabras, si estuviéramos “quemando” menos puntos de los que estimamos, no podríamos, mediante este gráfico, discernir si esto se debe a una capacidad inferior del equipo a la estimada o al descubrimiento de nueva funcionalidad. Si quisieramos presentar esta diferencia, podríamos hacerlo del siguiente modo:



Pueden existir muchas sutilezas en este juego de colaboración y comunicación que es el desarrollo de software. El gráfico podría mostrar que estamos “quemando” menos puntos de los planeados, pero esto puede deberse a que la funcionalidad no está claramente expresada en las *User Stories* (y, por ende, cuesta “aceptarlas”). Estas herramientas disparan conversaciones que permiten clarificar razones y buscar mejoras progresivas.

## *Parking Lot<sup>42</sup> - por Gisela Decuzzi - Una forma de mostrar avance*

Uno de los desafíos con los que me suelo encontrar en los proyectos en los que trabajo es la necesidad de comunicar el avance de las tareas y denotar qué tan cerca (o lejos) estamos de alcanzar nuestros objetivos. Esto puede parecer trivial. Sin embargo, al descomponer el trabajo en pequeñas historias, dependiendo de cómo nos organicemos, resulta difícil entender esta situación de un pantallazo. Muchas veces tenemos

---

<sup>42</sup>[https://leadinganswers.typepad.com/leading\\_answers/2007/02/summarizing\\_pro.html](https://leadinganswers.typepad.com/leading_answers/2007/02/summarizing_pro.html)

funcionalidades desarrolladas que a simple vista parecerían completas, pero, por una cuestión de organización y prioridades de negocio, no lo están.

Déjenme que cite un ejemplo, basado en un caso real. Nos encontrábamos realizando una migración del ingreso a un sistema existente. Descompusimos los temas en grandes secciones: Ingreso a la aplicación, registraciónde usuario (*sign-up*), recuperación de contraseña, bloqueo de usuarios, etc. Rápidamente generamos y tomamos las historias más importantes que le daban forma al desarrollo. Teníamos la sensación de que, tan solo a 2 *sprints* de haber empezado, ya habíamos terminado todo lo necesario. En realidad, habíamos manejado los casos principales, sin sumergirnos en profundidad. Por ejemplo, los usuarios podían registrarse sin ser verificada la seguridad de las *passwords* o ingresar al sistema sin permanecer logueados a partir de una sesión anterior o solicitar el *reseteo* de su contraseña con un pedido anterior aún vigente. Estos *features* para el negocio eran muy importantes, dado que representaban los diferenciales sobre el sistema de ingreso existente. Sin embargo, como los usuarios de negocio no estaban en el día a día, terminaba explicándoles una y otra (y otra) vez qué funcionalidad nos faltaba. Lo cierto es que no teníamos buena visibilidad del estado de avance del proyecto.

Prefiero no invertir demasiado tiempo en la generación de reportes complejos, costosos de mantener o dependientes de una persona. **Una buena herramienta de reporte es aquella que, de un pantallazo, comunica información compleja de forma entendible para el que la recibe.** Esto era lo que buscaba. Algunos colegas me sugirieron un reporte llamado *Parking Lot*.

Antes de entrar en detalle acerca de cómo lo implementamos en varios proyectos, permítanme destacar algunas características de los proyectos donde este reporte fue de gran utilidad:

- **Agrupación en grandes temas:** en particular mapear *épicas* y descomponerlas en historias, pero al comunicar siempre hacer referencia a las épicas.
- **Tamaño:** cada gran tema tiene su complejidad y su tamaño consecuente, que uno pretende comunicar al exterior. Estimamos cada historia usando *Story Points* y el tamaño de una

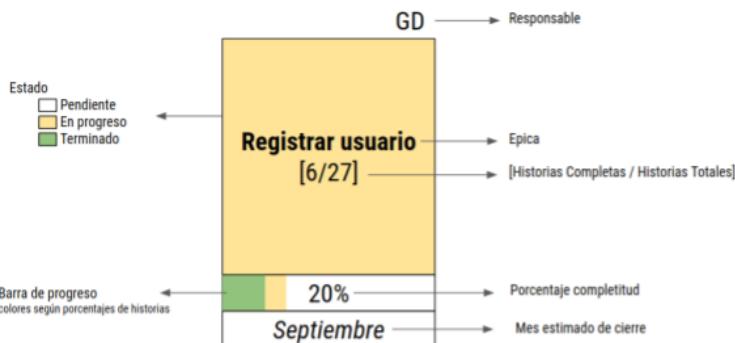
- épica lo consideramos simplemente como la suma de los puntos de sus historias.
- **Avance:** al comunicar, simplificar el ciclo de vida de las historias en pasos sencillos: “Pendiente”, “En Progreso”, “Terminado” con hitos claros de cuando se cambia de un estado a otro. Podrían existir más estados intermedios (por ejemplo, en revisión, en testing, rechazado, etc.) pero, fuera del equipo de desarrollo, no se requiere tanta granularidad dado que lleva a confusiones.
  - **Tema bloqueado:** cuando nos encontramos con dependencias no resolubles por el equipo de desarrollo y necesitamos de alguien más para desatascar este tema, resulta información valiosa para comunicar hacia fuera del equipo.

La idea del *Parking Lot* es usar la metáfora de un estacionamiento, organizado por sectores para distintos tipos de vehículos. Con esta inspiración, dimensionamos los temas en tamaños relativos, usando rectángulos más grandes mientras mayor esfuerzo se requiera.

Recomiendo utilizar las épicas como grandes temas, dimensionados mediante la sumatoria de los puntos de cada historia que lo conforma, para, con esas proporciones, formar rectángulos más amplios. Si notamos que varias épicas componen una agrupación que tiene sentido comunicar (por lo general nos pasa), simplemente encerramos los temas en un rectángulo.

Cada uno de los rectángulos que representa un gran tema queremos que comunique:

1. **Estado:** para esto podemos usar una convención de colores, por ejemplo:
  - a. **Blanco:** Pendiente
  - b. **Naranja:** En progreso
  - c. **Verde:** Terminada
2. **Avance:** hace referencia a su nivel de completitud usar un porcentaje de la épica e indicarlo con una barra de progreso (similar al medidor de batería de los celulares).
3. **Descripción del tema:** indicar en pocas palabras a qué nos estamos refiriendo, el nombre de la épica.



**Lectura General:**

El tema **Registrar usuario** cuyo **responsable** es **GD** se encuentra **en progreso**, teniendo **6 de 27 items completos**, lo cual representa un **20%** de la tarea **habiéndole tareas actualmente en progreso** y **teniendo una fecha estimada de cierre en el mes de Septiembre**

Ejemplos en distintos estados:



**Lectura General:**

El tema **Registrar usuario** cuyo **responsable** es **GD** se encuentra **BLOQUEADO**, teniendo **6 de 27 items completos**, lo cual **representa un 20% de la tarea**. No se puede estimar fecha de finalización.



**Lectura General:**

El tema **Registrar usuario** cuyo **responsable** es **AR** se encuentra **pendiente de avance**, teniendo como fecha de finalización **estimada el mes de Noviembre**.



**Lectura General:**

El tema **Registrar usuario** cuyo **responsable** es **GD** se **terminado** y en **estado productivo**. (al estar concluido este tema no seguirá apareciendo en futuros reportes de avance)

Si aplicáramos esto a todo el proyecto, obtendríamos un gráfico similar a:

GD	AG	AA	JA	PS
Registrar usuario [6/27]	Ingreso al sistema	Recuperar usuario / contraseña [10/20]	Remember me [1/13] 	Soporte a usuario final [35]
20% Septiembre	100% Julio	50% Septiembre	8% ???	0% Enero 2020
<b>- MVP -</b>				

## *Planning failure modes*

### *Un plan detallado para lidiar con la complejidad*

Muchos *Project Managers* que conocí procuraban manejar la complejidad inherente a un proyecto de desarrollo de software mediante un plan detallado que buscaba adelantarse a todas las posibles circunstancias. Esto brindaba una falsa sensación de seguridad: ¿qué podría salir mal con tal nivel de planificación?

Prefiero seguir los valores ágiles para manejar la complejidad: presentar toda la información de modo visible y transparente, para que, junto al equipo, decidamos el curso de acción a medida que el proyecto progresá. La evaluación es constante y se refleja en el plan, que va mutando para alcanzar los objetivos propuestos.

## *Multitasking*

Este es un *failure mode* típico de grandes corporaciones: planear como si fuéramos a trabajar en un solo proyecto, pero hacerlo en varios, además

de sumar reuniones y otras actividades. El impacto en cada proyecto y en la productividad es inmenso y suele pasar desapercibido<sup>43</sup>.

¿Conocen la Ley de Little<sup>44</sup>? Dice que, si estamos haciendo X tareas a la vez, demoraremos X veces más en terminar cada una de ellas. Bastante fuerte, ¿no? Visualicemos la fórmula mediante un ejemplo: imaginen que tienen un negocio y son capaces de atender 2 personas por hora (*Throughput*). Asumiendo que dedican media hora (0.5 horas) para cada una (*Leadtime*), según la ley mencionada, la cantidad promedio de personas en el local (*wip*) será 1.

$$\Rightarrow \text{throughput} * \text{leadtime} = \text{wip}$$

Con nuestros datos:

$$\Rightarrow 2 * 0.5 = 1$$

Despejando el *lead time*, podemos ver que:

$$\Rightarrow \text{leadtime} = \text{wip} / \text{throughput}$$

Esto implica que si en vez de atender 1 persona atendiésemos 2 simultáneamente, el tiempo para terminar con cada una de ellas será de 1 hora y no 30 minutos. Noten que el tiempo para completar las 2 tareas implica lo mismo, es decir, 1 hora, por lo que no se perdió productividad. Sin embargo, cada uno de las tareas demora el doble.

$$\text{tarea a la vez} \Rightarrow 1/2 = 0.5 \text{ hr}$$

$$2 \text{ tareas a la vez} \Rightarrow 2/2 = 1 \text{ hora}$$

Para hacer estos cálculos, asumimos que podemos *switchear* de tarea sin ningún *overhead* (como las computadoras). Para las personas, estos cambios no son gratuitos lo que, en consecuencia, repercutirá en el tiempo de cada uno.

---

<sup>43</sup> Si buscan un juego para visualizar el costo del multitasking, les recomiendo *The Multitasking Name Game* de Henrik Kniberg:  
<https://www.crisp.se/gratis-material-och-guider/multitasking-name-game>.

<sup>44</sup> <https://berriprocess.com/es/2016/01/03/la-ley-de-little/>

## *Flow*

Siempre me produjo mucha molestia la interrupción en el “momento del codeo”, ya que me resulta difícil alcanzar una concentración adecuada: entender el problema e imaginar el diseño para, finalmente, empezar a escribir código. Este proceso no ocurre de un momento a otro. Sin embargo, no fue hasta la lectura de *Peopleware* (De Marco, 1999), cuando supe de un estado, llamado “*flow*” por los psicólogos, que implica una condición de concentración y meditación extrema, donde se pierde la noción del tiempo y el esfuerzo. Juan Pablo, un desarrollador y músico que trabaja en 10Pines, lo explicó de este modo: “Ciertas actividades necesitan de un gran impulso inicial, un esfuerzo equiparable con el despegue de un avión. Mientras se avanza en dicho estado mental, se gana altura. Una vez arriba, mantenerlo no cuesta tanto esfuerzo. Sin embargo, una interrupción implica el reinicio del proceso”. Muy posiblemente, trabajos ligados al desarrollo o a la música necesiten de este estado. Existen otros roles, como el de *Project Manager*, que desempeñan múltiples actividades pequeñas y casi simultáneas, para quienes alcanzar este estado no resulta necesario.

Piensen entonces las consecuencias que implicará *staffear* a un desarrollador en más de un proyecto: cada uno de ellos demorará más y los integrantes no tendrán los tiempos necesarios para alcanzar el *flow*, además del *overhead* que se producirá por cada uno.

## *Deadlines estrictos*

¿Les resulta familiar este diálogo?

- *Project Manager*: Tenemos que terminar este proyecto para el 21 de Julio.
- Desarrollador: ¿Ya está definido el alcance?
- *Project Manager*: Todavía no.
- Desarrollador: ...

Yo lo escuché en muchas ocasiones; ¿les parece razonable?

De similar modo, nos encontramos frecuentemente con *Project Managers* que sostienen fechas irrealizables, buscando forzar la productividad del

equipo. Así, los integrantes trabajarán más horas, resignando tiempo personal. En general, estas personas suelen ser muy convincentes, logrando que el equipo se sacrifique para después sufrir grandes problemas de calidad o la renuncia de integrantes, extenuados por el ritmo de trabajo. No creo que este tipo de motivación obtenga buenos resultados. “Con esta presión los desarrolladores no trabajan ‘mejor’, trabajan más rápido”.

Creo en la visibilidad, la transparencia y la honestidad para fijar una fecha y medir el progreso. Creo que todo el equipo debe estar al tanto de las razones de un *deadline* de haberlo. Además, debe creer que es factible alcanzarlo mediante un ritmo sustentable. No tengo miedo de caer bajo la Ley de Parkinson<sup>45</sup>. En equipos donde existe confianza y colaboración, las tareas que terminan antes, simplemente terminan antes. Toda esta información se refleja claramente en nuestro plan, en todo momento.

## Conclusiones

Tener un plan no significa otra cosa que saber adónde se quiere llegar y tener trazada una ruta hacia dicho punto. Cuando comenzamos, nos encontramos frente a una gran incertidumbre. Delineamos un plan liviano, que nos permite aprender rápidamente. Comenzamos a trabajar enfocados en agregar valor y reducir el riesgo, incorporando el conocimiento adquirido en nuestro plan.

Muchas veces escuché decir que en *Agile* no se planea. ¿Tiene sentido pensar en que se puede empezar a trabajar sin un mínimo de descubrimiento y planeamiento? Creo que no. El punto es cuánto resulta razonable y eficiente. En otras palabras, cuánto invertiremos en hacerlo: ¡No pierdan el tiempo adivinando!

Finalmente, recuerden que el plan es simplemente otro de los artefactos que nos ayuda a entregar un producto con valor. Nuestro objetivo no es seguirlo.

---

<sup>45</sup> <https://explore.easyprojects.net/blog/parkinsons-law-the-secret-to-project-management-success>

# El Corazón de los Equipos Ágiles en 10Pines



## Introducción

El desarrollo de software se hace en equipos. Un equipo no implica la reunión de un grupo de personas. Es mucho más que eso, tiene identidad. Los integrantes colaboran. Tienen el mismo objetivo, un sentido de compromiso y solidaridad y sienten que aportan valor. Tom de Marco los denominó “equipos consolidados” (De Marco, 1999) (en inglés, *Jelled Teams*, que toman forma, como la gelatina cuando se cocina).

¿Tuvieron la suerte de trabajar en un equipo así? Conformado por gente capaz y motivada, que comparte objetivos y da lo mejor para alcanzarlos. Se respira la energía del grupo y la sinergia que puede alcanzarse es increíble.

¿Qué hace que un equipo funcione bien? ¿Cuáles son sus valores? ¿Cómo se forma?

Primero, y como diría cualquier chef, es fundamental la materia prima, los ingredientes: **En 10Pines, buscamos personas con “madera de pino”**, sólidas técnicamente y que amen desarrollar software. Además, deben ser honestos y querer ser parte de la empresa.

Para que estas personas rindan al máximo, debemos **crear un ambiente dentro del cual puedan florecer**. Donde confíen, puedan expresarse sin ningún tipo de miedo y volcar su talento. Un lugar que les permita seguir creciendo, desarrollarse, aprender y donde se sientan motivados y energizados.

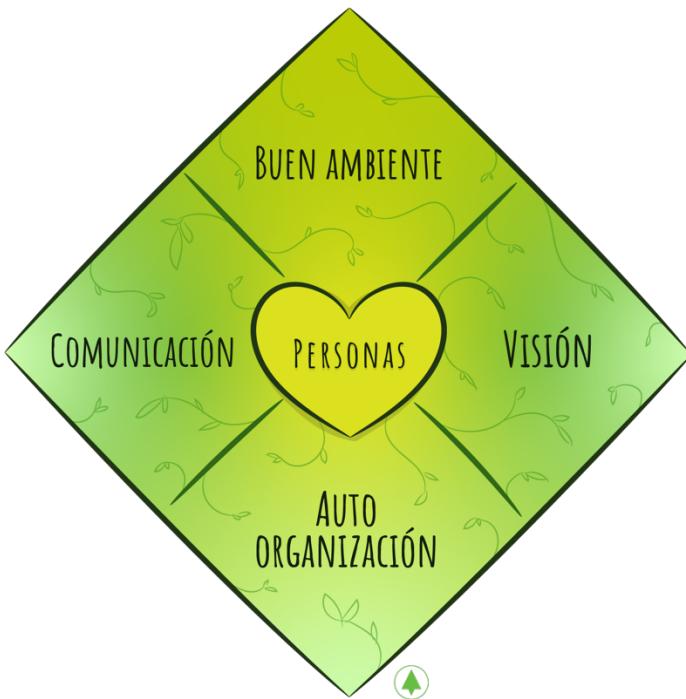
La visualización de los objetivos por parte del equipo dispara la inteligencia colectiva. **En 10Pines, trabajamos para establecer una visión compartida** tanto del futuro de la empresa como de cada uno de los proyectos que enfrentamos.

**A partir de esta visión, nos organizamos** para trazar un plan y trabajar en pos de los objetivos. **Estamos empoderados** para contribuir a la solución y lo hacemos con toda nuestra energía. 10Pines es una empresa horizontal que yace en el poder de las personas y de la auto-organización.

Finalmente, reconocemos que **la comunicación es clave para el éxito**. Por esta razón, **nuestra Metodología maximiza el ancho de banda**, procurando la mayor eficiencia posible. Trabajamos en un *open-space*,

participamos en las reuniones (reduciendo *hands-off*), acortamos ciclos y usamos herramientas de gestión y comunicación.

Describiré nuestra receta usando el formato propuesto por Alistair Cockburn en su Corazón de la Agilidad<sup>46</sup>. Lo denominé “El Corazón de los Equipo Ágiles en 10Pines”.



El corazón está compuesto por las personas. Sin un buen ambiente, no es posible auto-organizarse efectivamente. Establecer una visión clara es parte de una buena comunicación, pero es tan importante que, creo, merece estar en su cuadrante. La auto-organización funciona de modo eficiente con personas como las descritas, en un buen ambiente y con una visión clara. Pueden ver que estos atributos están relacionados, son interdependientes.

---

<sup>46</sup> <https://heartofagile.com/>

## ¿Cómo formamos los equipos?

Primero, seleccionamos las personas más adecuadas para integrarlos. Muchos me han preguntado: ¿cómo es posible que todos participen? Quienes así consultan piensan que esta modalidad favorecería la toma individual de decisiones. Esto no sucede. Somos responsables y buscamos el bienestar colectivo.

Los equipos están conformados por integrantes dedicados a tiempo completo. Esto es mejor para la productividad, la calidad y para las personas, que pueden concentrarse sin sufrir interrupciones. Dentro del equipo existen todos los *skills* necesarios para desarrollar un incremento de funcionalidad. Equipos interdisciplinarios permiten una gestión mucho más simple y efectiva.

Pueden notar que los equipos no presentan ninguna característica especial. Son conformaciones circunstanciales, auto-organizadas para atender clientes y proyectos. No existen fricciones o contradicciones con la empresa<sup>47</sup>. Comparten valores, principios y prácticas.

## Los equipos son las personas

Analicemos cada uno de los atributos que hacen a los buenos equipos, empezando por el corazón. Los equipos son las personas. Las empresas, también. No se puede desarrollar software con gente desmotivada, en la

---

<sup>47</sup> **Equipos Ágiles en empresas no Ágiles?**

En el pasado, trabajé en equipos ágiles, dentro de empresas que no lo eran. Reconocía la importancia de la auto-organización para lidiar con la complejidad y alcanzar un mejor grado de colaboración. Sin embargo, la empresa poseía una estructura jerárquica. Además, competíamos duramente por lograr un bono que era repartido entre pocos. Eso generaba grandes discrepancias que repercutían tanto en la performance colectiva como en la motivación de cada uno de los integrantes. Reportar a un jefe externo o lidiar con personas, que no comparten la modalidad de trabajo y los valores, afecta la moral y la motivación. Me pregunto si en estas situaciones existe un techo de cristal que limita a estos equipos.

cual no confiamos. Tampoco, ser ágiles. **Necesitamos buenos desarrolladores, honestos, responsables y colaborativos.**

En esta sección les hablaré sobre las personas, el ingrediente fundamental de los equipos. Les contaré cómo seleccionamos las semillas, sus atributos y describiré el “camino del pino”, nuestro plan de carrera.

### *Proceso de selección*



Como sabemos que las personas son importantes, invertimos mucho tiempo en el proceso de selección. No buscamos programadores con ciertos *skills* para ingresar a un proyecto particular. **Buscamos programadores que deseen desarrollar su vida profesional en 10Pines.** En consecuencia, somos muy estrictos en la evaluación técnica. También queremos conocer, tanto como sea posible, su parte humana. Con estos objetivos hemos ideado nuestro proceso de selección.

Como mencioné anteriormente, **la excelencia técnica es parte de nuestro ADN.** No creemos que se pueda hacer software con

programadores que no sean fuertes técnicamente, que no sepan diseñar o testear. Los postulantes desarrollan un ejercicio usando el lenguaje de programación de preferencia. A continuación, debatimos las decisiones de diseño tomadas, las abstracciones creadas y el proceso utilizado para llegar a ellas (¿usaron *Test Driven Development*?). No hacemos foco en el aspecto tecnológico o en los detalles del lenguaje.

El costado humano resulta difícil de evaluar en un periodo de tiempo tan corto. Para lograrlo, **organizamos una entrevista grupal de la que participan todos los integrantes de la empresa**. El objetivo es conocer al candidato o candidata en profundidad, evaluando si es alguien con quien nos gustaría trabajar. Piensen que compartiremos muchísimas horas de nuestro día con él o ella.

### *Skills de los pinos*

¿Qué significa “saber” desarrollar software? ¿Qué implica “ser” un buen programador? Considero que los buenos desarrolladores con los que he trabajado a lo largo de los años poseen esta combinación de *skills*:

- **Comprenden el dominio en profundidad:** ¿Cómo puedes modelar algo sin conocerlo plenamente? Los buenos programadores buscan entender en profundidad los conceptos que forman parte del problema que enfrentan. Investigan, preguntan y se empapan del dominio.
- **Saben diseñar:** Tienen habilidad para plasmar en el modelo las abstracciones correctas. Ser un buen diseñador se aprende, a partir de cursos y compartiendo proyectos junto a programadores experimentados. Sin embargo, creo que los mejores tienen algo innato.
- **Dominan el lenguaje de programación:** Dominar el lenguaje permite enfocarse en la resolución del problema, creando, además, soluciones más “elegantes”<sup>48</sup>. Conocer los usos y

---

<sup>48</sup> Sé que la “elegancia” constituye un concepto subjetivo, pero también sé que todos los programadores podemos distinguir código elegante.

costumbres (lo que llamamos *idioms*<sup>49</sup>) hace que nuestro *codebase* luzca bien.

- **Técnicas:** Considero que un buen programador debe ser prolíjo y disciplinado para construir software, usando *test driven development*, refactorizando continuamente e integrando tan pronto como sea posible. Estas técnicas se vuelven fundamentales cuando trabajamos en un equipo, para poder sincronizar y colaborar correctamente.
- **Herramientas:** Finalmente, es importante conocer en profundidad las herramientas que usamos diariamente en nuestro trabajo: la *IDE*, sus atajos, sus diferentes vistas y sus funcionalidades. También el versionador de código. El desarrollo debe fluir sin que existan “fricciones”.



Junto con las habilidades “duras”<sup>50</sup> que les nombré, se van desarrollando otras, “blandas”, como la comunicación, la gestión de proyectos y el

---

<sup>49</sup> [https://en.wikipedia.org/wiki/Programming\\_idiom](https://en.wikipedia.org/wiki/Programming_idiom)

liderazgo. La comunicación implica una doble faz: la capacidad de hablar ante propios y extraños y la capacidad de facilitación dentro del equipo. Las de gestión se relacionan con el conocimiento de la Metodología de 10Pines, por ejemplo, facilitar los talleres de *Product Discovery* o gestionar un *Backlog*. El liderazgo implica la capacidad para entender el sistema y ayudar a co-crearlo, estableciendo las reglas y restricciones. En otras palabras, los líderes ya tienen los *skills* para trabajar en la creación del ambiente de trabajo que deseamos. También tienen la capacidad de guiar al resto de los pinos a través de sus caminos.

## *El camino del pino*

Llegar a ser un buen programador implica un recorrido, que nosotros llamamos “Camino del Pino”. Durante el trayecto, crecemos y nos desarrollamos tanto humana como profesionalmente. Hemos definido un marco de referencia: comenzamos como “*Padawans*”, copiando y aprendiendo de pinos más experimentados. Nos convertimos luego en “*Knights*”, programadores experimentados y con conocimientos de gestión. Después de muchos años, alcanzamos el “rango” de “*Masters*”, programadores expertos, conocedores de la profesión en profundidad. El lector podrá deducir de qué saga hemos extraído los nombres.

Durante las etapas iniciales del camino, como “*Padawans*”, los hitos consisten en mejorar los *skills* de programación, es decir, aprender lenguajes, *frameworks*, técnicas, herramientas y participar con responsabilidad y compromiso de las actividades del equipo. Los “*Knights*” son programadores capaces de resolver la mayoría de los problemas de forma autónoma. Tienen una visión más amplia del sistema, que incluye factores como la seguridad y la performance. Toman responsabilidad en parte de la gestión del proyecto y ayudan a sus pares. Dominan varios lenguajes de programación y manejan las herramientas que usamos diariamente (*git*, *ide*) a la perfección. En la última etapa del camino están los *Masters*, nuestros líderes, encargados de crear y fomentar el mejor ambiente laboral posible. Ellos guían a los más jóvenes por sus

---

<sup>50</sup> Ciencias duras y blandas -

[https://es.wikipedia.org/wiki/Ciencias\\_duras\\_y\\_blandas](https://es.wikipedia.org/wiki/Ciencias_duras_y_blandas)

caminos y también trabajan para que 10Pines siga creciendo, obteniendo nuevos clientes y manteniendo la relación con los existentes.

Una empresa autogestionada, como la nuestra, suma actividades a sus integrantes que otras no tienen, como *staffear* los equipos de trabajo, organizar eventos o definir las compras de hardware. Los pinos asumimos compromisos gradualmente que impactan dentro de nuestro crecimiento profesional.

Evaluamos de forma continua el progreso, para saber en qué lugar del camino estamos y, en consecuencia, hacer ajustes y planear objetivos. Para ello, elegimos un “jardinero”, de mayor *seniority*, que nos acompaña en el recorrido.

Este camino es fundamental, tanto para cada uno de nosotros, los pinos, que, como trabajadores del conocimiento, valoramos la motivación intrínseca, como para 10Pines, que desea que las personas crezcan aquí. En consecuencia, invertimos mucho esfuerzo en “señalizarlo”. Evaluamos constantemente cómo está funcionando y qué podemos hacer para mejorarlo.

## Un buen ambiente



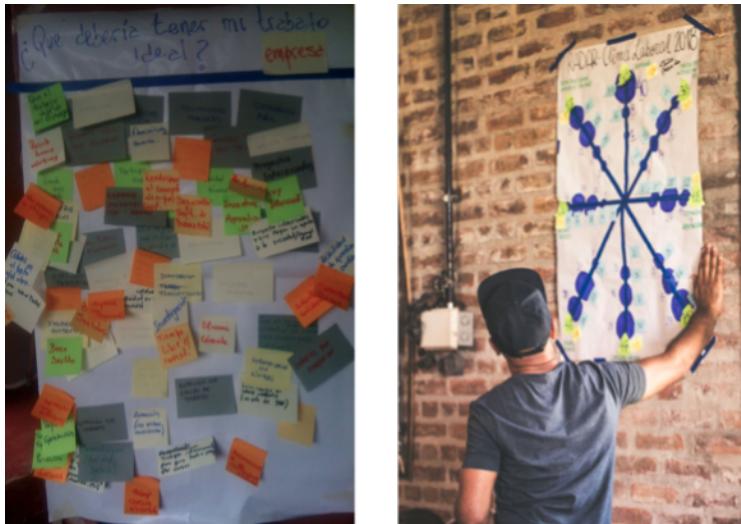
Una vez que tenemos buenas semillas, **debemos crear un ambiente donde puedan florecer en su mejor expresión, desarrollándose al máximo.** Debe ser un lugar que nos permita volcar todo nuestro talento, donde nos sintamos libres, sin miedo, confiados, motivados y energizados. Debe ser un ambiente de innovación, que despierte la imaginación y dispare las ideas. También, un lugar que nos haga sentir orgullosos por lo que hacemos. Los programadores necesitamos saber que lo que estamos haciendo aporta valor.

Trabajar en un ambiente así permite que los talentos de todas las personas se combinen y hagan sinergia para obtener los mejores resultados. No quedan dudas de que, cuando todos colaboramos, el producto resultante es mejor. Como resultado adicional, esto nos estimula, nos motiva. Esta es la manera en la que trabajamos en 10Pines. No imagino hacerlo de otro modo.

¿Cómo podemos crear este ambiente? ¿Cómo podemos configurar las reglas de este sistema complejo, que es la organización, para alentar estos sentimientos y comportamientos? En esta sección, les contaré lo construido, entre todas y todos, en 10Pines.

### *¿Qué nos motiva?*

En uno de nuestros primeros retiros estratégicos, hicimos una actividad que tuvo el propósito de consensuar los atributos que debería tener nuestro trabajo ideal. En otras palabras, los factores que contribuirán a nuestra felicidad, a sentirnos bien, cómodos y contentos en nuestra labor diaria.



La 1<sup>ra</sup> foto muestra la actividad que hicimos para obtener estos atributos y luego priorizarlos. La 2<sup>da</sup>, una retrospectiva donde evaluamos cómo estamos respecto a cada uno de ellos.

Los factores que acordamos son:

- Proyectos interesantes
- Gestión participativa
- Calidad humana
- Calidad técnica
- Sueldos
- Crecimiento profesional y aprendizaje constante
- Comodidad laboral
- Imagen y reputación

Los que conocen la teoría X y la teoría Y de McGregor<sup>51</sup> sobre *Management* podrán notar que, dentro de los atributos que definimos, existen algunos motivadores extrínsecos, como los sueldos y la comodidad laboral. Sin embargo, la mayoría son intrínsecos, es decir,

<sup>51</sup>

[https://es.wikipedia.org/wiki/Teor%C3%ADa\\_X\\_y\\_teor%C3%ADa\\_Y](https://es.wikipedia.org/wiki/Teor%C3%ADa_X_y_teor%C3%ADa_Y)

motivadores que provienen de los objetivos desafiantes, de la asunción de responsabilidades, logros obtenidos y reconocimiento consecuente.

Tenemos en cuenta estos atributos en cada una de las decisiones que tomamos: ¿Qué proyectos deberíamos tomar? ¿Cómo deberíamos aumentar nuestros sueldos? Estos valores representan nuestra guía, el consenso de lo importante para el grupo.

Además, una vez al año, durante nuestro retiro estratégico (una reunión realizada entre todas y todos, que trataré en breve), inspeccionamos estos atributos en profundidad usando la retrospectiva del radar<sup>52</sup>: Puntuamos cada una de las “aristas” para luego graficar en la herramienta los promedios obtenidos. De esta manera, podemos visualizar y debatir cómo estamos: ¡Muchas mejoras profundas han surgido de las conversaciones disparadas por esta retrospectiva!

## *¿Qué hacemos para lograr un buen ambiente?*

### *Promover la confianza*

¿Podrías trabajar en un lugar en el que no confías en la empresa o en tus colegas? Creo que no. Resguardarías la información. No te arriesgarías a cometer errores, no colaborarías.

**En 10Pines establecemos una relación de confianza desde el momento del ingreso.** Un ejemplo de esto lo constituye la carga de las horas trabajadas que cada uno realiza para luego calcular su sueldo. Cada uno de los procesos creados o de las decisiones tomadas están basados en la confianza. Es nuestro valor más importante. Es la base de un lugar sano y colaborativo.

---

<sup>52</sup> Esta retrospectiva es muy interesante para hacerla con sus equipos.

Genera una base de entendimiento y de valores compartidos. Nos permite tomar decisiones durante la ejecución del proyecto (guiadas por estos valores). Sentirnos un equipo. Pueden ver más información en este post: <https://medium.com/the-liberators/retrospective-do-the-team-radar-1794057653e9>

### *Ser transparentes*

La transparencia fomenta la confianza y, por ende, la colaboración. En 10Pines, la información está disponible para su consulta. **Somos una empresa de números abiertos<sup>53</sup>**: cualquiera puede ver los ingresos de la empresa, la rentabilidad de los proyectos y, también, lo percibido por cada pino.

Cuando desarrollamos un proyecto, usamos todos los recursos disponibles para aumentar la transparencia, como, por ejemplo, radiadores de información<sup>54</sup> o herramientas de gestión online. Buscamos que los integrantes del equipo cuenten con la mayor cantidad de información posible, acerca de los objetivos, del progreso y de todas las aristas que puedan tener alguna incidencia en el resultado final.

### *Compartir objetivos*

Cuando conformamos equipos para construir un producto, compartimos los objetivos, que definimos en el *Product Discovery*. ¿Qué pasaría si éstos fueran personales? Por ejemplo, si se midiera la performance de los programadores únicamente por su desempeño individual (*User Stories* terminadas) o si los integrantes del equipo pertenecieran a otras áreas, con fines contrapuestos. Seguramente, enfrentaríamos comportamientos competitivos y disfuncionales. Las reglas que especificamos en un sistema complejo determinan las conductas de sus agentes que, deseamos, sean colaborativas. El éxito del grupo debe ser el éxito de cada uno de los integrantes. **Los objetivos de 10Pines son los objetivos de todos nosotros.**

### *Hacer los procesos justos*

Un punto muy importante dentro de la transparencia es que los procesos sean justos, que las reglas estén claras. En el artículo “*Fair Process*:

<sup>53</sup> [https://en.wikipedia.org/wiki/Open-book\\_management](https://en.wikipedia.org/wiki/Open-book_management)

<sup>54</sup> <https://www.agilealliance.org/glossary/information-radiators/>

*Managing in the Knowledge Economy*<sup>55</sup>, se explica su importancia para los trabajadores del conocimiento.

**En 10Pines, los procesos y políticas se encuentran claramente definidos en una web de acceso irrestricto.** Cuando surgen aspectos que no tenemos contemplados en ninguna política, nos juntamos, debatimos, consensuamos una y la publicamos en la web mencionada. Al ser una empresa de estructura horizontal, todos nos encargamos de respetar y hacer respetar estos procesos co-creados.

### *Tomar las decisiones en equipo*

Con este nivel de transparencia, confianza y colaboración, las decisiones deben tomarse de manera compartida. De esta manera, aseguramos que los integrantes estén comprometidos. **En 10Pines, cualquier grupo que se conforme posee un carácter abierto**, es decir, cualquier pino puede integrarlo. Y tomamos las decisiones entre todos.

Trabajar de este modo puede resultar más lento e incómodo. Sin embargo, como uno de los líderes de 10Pines, puedo asegurarles que es muy gratificante. Valoramos sentirnos escuchados.

### *Crear espacios de reconocimiento personal*

Otro de los factores necesarios para sentirnos bien es el reconocimiento. Que una persona con la que trabajamos nos diga que estamos haciendo las cosas bien aumenta la autoestima y motiva intrínsecamente.

La pirámide de Maslow<sup>56</sup> muestra que el reconocimiento es parte de nuestras necesidades esenciales, después de la seguridad y de la confianza. El ser humano precisa tanto ser reconocido como demostrar reconocimiento.

**En 10Pines, nos gusta reconocer un buen trabajo.** Lo hacemos todo el tiempo, por ejemplo, cuando alguien nos ayuda a resolver un problema

---

<sup>55</sup> <https://hbr.org/2003/01/fair-process-managing-in-the-knowledge-economy>

<sup>56</sup> [https://es.wikipedia.org/wiki/Pir%C3%A1mide\\_de\\_Maslow](https://es.wikipedia.org/wiki/Pir%C3%A1mide_de_Maslow)

o prepara una charla. Publicamos estos reconocimientos mediante “La muralla de Kudos”. Esta herramienta, basada en las cartas de *Kudos* de *Management 3.0*, fue desarrollada por uno de nuestros pinos, Joaquín, para “enviar” *kudos* virtuales a otro/s pino/s a través de *Slack*<sup>57</sup>, que luego compartimos durante la *standup* semanal. ¡Se siente muy bien aparecer en la muralla y también agradecer a alguien de esta manera!



### *Fomentar momentos de distensión*

Me gusta salir a tomar una cerveza y mantener conversaciones fuera del ámbito laboral con mis compañeros de trabajo. Estos momentos permiten relajarnos y conocernos. Crean lazos fuertes que luego nos permiten trabajar mejor. **En 10Pines, jugar juegos de mesa, compartir un *after-office* o ver películas constituyen una costumbre positiva.** Entre nosotros existen vínculos que van más allá del trabajo.

---

<sup>57</sup> <https://slack.com>

## *Las 5 disfunciones de las empresas*

¿Conocen este libro de Patrick Lencioni (Lencioni, 2002)? Me ayudó a entender algunas situaciones completamente disfuncionales que se daban en algunas de las empresas donde trabajé. El autor las describe a través de esta pirámide:



La primera disfunción, en la base, es la **falta de Confianza**, que nace de la no-voluntad a mostrarse vulnerable frente al grupo. Si los miembros del equipo no confían entre ellos, entonces no piden ayuda y ocultan información. En definitiva, hacen lo que es mejor para ellos mismos y no para el grupo.

La falta de confianza genera **miedo al conflicto**. Si los miembros del equipo no confían entre ellos, entonces no discuten. Si en un equipo nadie discute, ¡desconfíen! En 10Pines, tienen lugar debates, argumentos y discusiones, dentro de un ambiente saludable. Son actividades constructivas. Cuando uno está comprometido, convencido de que su postura es la correcta, discute con vehemencia.

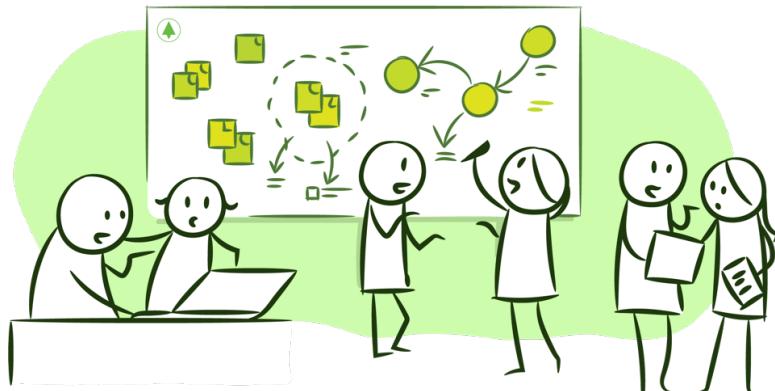
La imposibilidad de razonar constructivamente es la principal causa por la cual **los miembros no se comprometen con las decisiones** que se toman. Si no participaste de la discusión, ¿tú lo harías?

Miembros que no se comprometen son **miembros que no son responsables**. ¿Puede requerirse el seguimiento de un plan a un integrante en estas circunstancias?

Finalmente, los miembros no responsables crean un ambiente donde la 5<sup>ta</sup> disfunción florece: **No atención a los resultados**.

Este libro me ayudó a entender que, en ocasiones, ciertos comportamientos son el resultado de problemas más profundos. Sólo observamos los síntomas. He visto, por ejemplo, equipos que se comunican a través de herramientas (como Jira) para dejar “evidencia” o testers que incluyen un reporte de *bugs* antes de aceptar la *User Story*. En algunas ocasiones, cometí el error de centrarme en los síntomas, queriendo hacer los procesos más eficientes, cuando los problemas eran humanos, de falta de confianza y colaboración. Las personas, como agentes inteligentes dentro de este sistema complejo que es una organización, maquillan los problemas, desplazan el eje o se acomodan según intereses propios.

## Visión



“Si no sabes para dónde vas, cualquier camino te llevará allí” - Lewis Carroll<sup>58</sup>

Como ya mencioné en el capítulo 1, comenzamos los proyectos haciendo un *Product Discovery* del que participa todo el equipo y que permite construir una visión compartida que consensúa los objetivos y el plan para alcanzarlos. Dicha visión nos permite volcar todo nuestro talento en busca de las mejores soluciones. **Es el disparador de la inteligencia colectiva, el energizante que dispara las interacciones necesarias.** Sienta las bases para que se produzca el mejor tipo de colaboración. Somos agentes inteligentes, empoderados, en un ambiente que nos alienta a dar lo mejor de nosotros.

---

<sup>58</sup> “If you don't know where you are going, any road will get you there”.



10Pines funciona exactamente del mismo modo. **Todos los pinos construimos la visión de la empresa**, que repasamos y refinamos cada año, durante nuestro retiro estratégico. Esta visión nos guía, nos permite tomar las decisiones diarias.



## Auto-organización



Compartida la visión, reconocidos los objetivos, **el equipo se auto-organiza para empezar a trabajar**. Todos participamos opinando, debatiendo o planteando alternativas. Estamos empoderados para contribuir a la solución.

Habiendo contemplado una empresa auto-gestionada durante los últimos 9 años, no me quedan dudas: representa la mejor forma de trabajo. **Permite alcanzar resultados óptimos porque combina los talentos del grupo. E impacta en nuestro bienestar:** representa uno de los factores esenciales en la creación de un buen ambiente laboral.

### *¿Qué logramos con la auto-organización?*

¿Por qué se obtienen mejores resultados cuando se trabaja de esta manera? A continuación, describo algunas razones:

**Tenemos perspectivas parciales, que se combinan:** Crear la solución entre todos los integrantes del equipo permite que la misma sea equivalente a la suma de los talentos individuales: ¿Por qué prescindir de

ideas, perspectivas y mentes de personas inteligentes y capaces? *Management 3.0* (Appelo, 2010) detalla el “principio de la oscuridad”, que explica la razón por la cual, desde el punto de vista de la complejidad, resulta positiva la auto-organización: “Los miembros de un equipo de desarrollo poseen un modelo mental incompleto del sistema”. Cuando éstos se combinan, dicho modelo se completa, obteniendo ideas mejores que las que hubieran podido obtenerse por separado.

**Tenemos diferentes *backgrounds* y *skills*, que se potencian:** Nuestro trabajo, como desarrolladores de software, es un trabajo de creación y de innovación que requiere de un equipo diverso, con diferentes *backgrounds* y *skills*, que interactúe libremente. El video de ABC Nightline sobre *IDEO*<sup>59</sup> describe muy bien cómo debe crearse un ambiente de innovación: no existe un jefe, sino un facilitador, en una estructura completamente horizontal donde los integrantes tienen perfiles muy diversos. Se brinda un objetivo y se deja al equipo auto-organizarse. Debido a que el problema es altamente caótico, se fija un lapso para su finalización. Suena familiar, ¿no?

**Somos personas formadas, en contacto con los problemas:** El 5<sup>to</sup> principio en el libro de “*Lean Software Development*” (Poppendieck00, 2003) habla sobre Empoderamiento: “Empleados educados y empoderados, guiados por un líder, pueden tomar mejores decisiones que sus *managers*”, sugiere. El método científico de Taylor no fue pensado para organizaciones del conocimiento. En éstas, delegar “es mucho más efectivo porque estos ingenieros, trabajando en la línea de producción, tienen mejor información y, también, la formación necesaria para tomar esas decisiones”. El desarrollo de software es exactamente igual. Los programadores somos profesionales capacitados, que trabajamos en la “línea de producción”.

**Somos los que mejor sabemos como contribuir:** Cada integrante es quien mejor conoce sus propios *skills* y, por ende, quien mejor sabe cómo contribuir al equipo en función de los objetivos compartidos. Por esto, Jim Highsmith, en *Agile Project Management* (Highsmith, 2009), especifica que “cada una de las personas se hace responsable del pasaje de trabajo (de acuerdo con la necesidad), del manejo de su propia carga y

<sup>59</sup> <https://www.youtube.com/watch?v=M66ZU2PCIcM>

de la efectividad del equipo”. ¿Existe modo superador de gestionar un proyecto?

**Es la mejor manera de atacar la complejidad:** Los “sistemas adaptativos complejos” describen científicamente cómo funciona la auto-organización: la solución “emerge” a partir de la interacción de agentes inteligentes que toman decisiones diarias con un impacto en la solución final. Los proyectos de desarrollo de software son sistemas adaptativos complejos. La auto-organización permite atacar la complejidad inherente, a través de la participación de todos los integrantes (agentes) en el producto final. Dejarla en manos de una sola persona sería demasiado arriesgado.

**Nos hace sentir mejor y más comprometidos:** Cuando una persona capaz está motivada, quiere participar en las decisiones y en el devenir del proyecto. Siente, al hacerlo, una motivación intrínseca: una necesidad interna por alcanzar una conclusión exitosa. John Buck y Gerard Endenburg, en *“The Creative Forces of Self Organization”*<sup>60</sup>, argumentan que, cuando se trabaja de esta manera, no existen fricciones entre los intereses personales y los de la empresa. Nuestra energía se encauza hacia objetivos compartidos. Sentirnos motivados, energizados y trabajar auto-organizadamente en un equipo nos vuelve personas más responsables con nuestros pares y nosotros mismos. Esta responsabilidad nace de los deseos intrínsecos. Es diferente de la que podría originarse al ser reprendidos o premiados. Puedo decirles, sin temor a equivocarme, que nunca vi, en todos mis años de experiencia, equipos y personas tan comprometidas como en 10Pines.

### *Auto-organización en 10Pines*

10Pines es una empresa horizontal gestionada por quienes formamos parte de ella. Compartimos la responsabilidad de llevar adelante la empresa. Colaboramos para resolver los problemas.

---

<sup>60</sup> <http://sociocracyconsulting.com/wp-content/uploads/2016/04/CreativeForces-updated2012.pdf>

Me gusta trabajar de esta manera. Siento que a los restantes pinos también. De hecho, como ya vieron, se encuentra dentro de las aristas que valoramos colectivamente. Sentirnos partícipes de lo que hacemos nos motiva y todos nos sentimos partícipes de 10Pines.

No voy a contarles demasiados detalles de cómo nos organizamos. Creo que esto merece un libro por sí mismo. Solo quería comentarles que, al momento de escribirlo, somos más de 80 personas auto-organizadas para gestionar 10Pines. Tenemos equipos auto-conformados que se ocupan de tareas diarias que deben hacerse en todas las empresas, como el *recruiting*, la gestión del hardware o la configuración de la red. Conformamos equipos espontáneos para organizar eventos particulares, como una conferencia o, incluso, la fiesta de fin de año. Nos reunimos mensualmente para compartir los números de la empresa y tomar decisiones importantes como, por ejemplo, en qué proyectos trabajar y quiénes conformarán los equipos.

Por supuesto que la gestión participativa implica un gran desafío a la hora de tomar decisiones. Sin embargo, a pesar de las dificultades e incomodidades que puedan originarse, el efecto positivo consistente en participar de las decisiones y saber que el proceso es justo nos llena de energía y de confianza.

## *Líderes en 10Pines*

Les conté que en 10Pines nos auto-organizamos para gestionar la empresa. También, que no existen jefes. Podrán pensar ustedes que es un ambiente caótico, casi anárquico, pero nada dista más de la realidad. Existe un sistema, conformado por valores, procesos y reglas, que funciona muy bien.

Y existen líderes, que son las personas más experimentadas, las que mejor conocen la empresa. **Ellos sentaron las bases de la cultura. Configuraron el contexto. Fijaron los objetivos, el rumbo y las restricciones.** Los procesos que forman parte de nuestra columna vertebral, como el *recruiting* o las reuniones estratégicas mensuales fueron diseñados por ellos.

Al ser quienes mejor conocen los valores y la visión de 10Pines, se encargan de transmitirlos al resto de los pinos diariamente, en reuniones, eventos y frente a hechos relevantes, que marcan puntos de inflexión en la vida de la empresa. Hablan mucho con los pinos más jóvenes, alinean expectativas y se preocupan por su motivación. Son los “jardineros”, que guían a las personas a través del camino del pino. Aconsejan y brindan sus puntos de vista. En ambientes como el nuestro, sin estructuras formales, los líderes tienen un brillo particular. Son reconocidos por lo que hacen y transmiten, más que por sus títulos.

Claramente los fundadores son los líderes principales. Sin embargo, no son los únicos. Después de casi 10 años de existencia, muchas otras personas han tomado posiciones de liderazgo. Estos “*masters*”, que conocen la empresa íntimamente, participan en áreas vitales de la misma. Sin ellos, 10Pines no funcionaría del mismo modo.

## *Disciplina*

Quizás piensen que los equipos de estas características carecen de disciplina, pero no es cierto. Puedo asegurarles que **en 10Pines los equipos son auto-disciplinados, rigurosos con los procesos y las prácticas y responsables para cumplir los objetivos** teniendo en cuenta las restricciones. Cuando un grupo de personas competentes trabaja de esta manera, tomando responsabilidad colectiva por los resultados, la auto-disciplina (así como también, la auto-organización) sucede naturalmente.

Jim Highsmith expresa que “la auto-disciplina hace posible la libertad y el empoderamiento” y continúa diciendo que “uno de los peligros más grandes de los desarrollos centrados en procesos y de la gestión de proyectos es la reducción de los incentivos para la auto-disciplina. Posteriormente, comenta que “algunos *Project Managers* se preguntan por qué nadie toma la iniciativa y es responsable”.

Un punto que también vale la pena mencionar dentro de este tópico es **la necesidad de trabajar disciplinadamente para lograr excelencia técnica**. Esta disciplina no puede ser impuesta, ya que requiere de una práctica constante por parte de todos los integrantes del equipo. Las prácticas de *eXtreme Programming* permiten crear esa “presión de pares”

que nos lleva a trabajar de ese modo. Hacer *pair programming* y *code reviews* nos impulsa a buscar mejores diseños, a pensar cuidadosamente los nombres, a no dejar deuda técnica, a testear cuidadosamente, evitando atajos.

## Comunicación

*“Algunos de nosotros estamos en el negocio de la tecnología. La mayoría estamos en el negocio de la comunicación humana.”* - Tom de Marco<sup>61</sup>

Reconocemos la importancia de la comunicación. Por eso nos esforzamos en que sea lo más caudalosa y efectiva posible.

Alistair Cockburn define, en *Agile Software Development* (Cockburn, 2006), la comunicación como “la transmisión de una idea desde una persona a otra”. Cuando trabajamos en un equipo, emitimos diariamente conceptos, que se plasman en especificaciones y, en definitiva, en el *codebase* que representa la solución.

Lograr una buena comunicación es difícil. Existen malas interpretaciones, incluso cuando charlamos con nuestras parejas. ¡Imaginen en un equipo, con gente que conocemos menos y tenemos poco contacto!

La comunicación, dentro del enfoque de construcción de software que hemos elegido, el enfoque ágil, es crucial: Un equipo multidisciplinario, motivado y organizado desarrolla pequeños incrementos de funcionalidad para obtener feedback rápido. Este proceso, altamente concurrente, ya que implica la interacción de todos los roles que forman parte del equipo, requiere que fluya un caudal enorme de comunicación entre sus integrantes<sup>62</sup>.

<sup>61</sup> “Few of us are in the high tech business. Most of us are in the human communications business.”

<sup>62</sup> Alistair lo denomina “un **juego de invención y cooperación**”: un equipo de *crafters* colabora entre sí para que la solución emerja, aportando cada uno desde su perspectiva.

En consecuencia, **debemos usar los medios más eficientes y efectivos** a nuestro alcance: aumentar el ancho de banda de comunicación tanto como sea posible, que existan radiadores de información que disparen nuevas conversaciones, que fluya en todas direcciones y que sea directa, es decir, sin intermediarios.

### *¿Cuál es la manera más eficiente de comunicarse?*

Sin duda, estar con la persona o personas con las cuales queremos comunicarnos frente a frente, ya que podemos observar su lenguaje corporal y aumentar la velocidad al notar la comprensión o detenernos en caso de observar dificultades. Desde el punto de vista del receptor, podemos interrumpir, permitiendo que el emisor aclare, para continuar sobre bases sólidas. ¿Cuál es la manera menos eficiente? La comunicación escrita, donde el emisor genera un documento para que otras personas lean e interpreten. No sólo es más lento, sino también más proclive a errores. Entre una y otra, podemos encontrar diversos métodos alternativos como *Google Hangouts*, que permite visualizar a las personas y compartir la pantalla o *Slack*, que posibilita la creación de grupos con los cuales se comparte información masivamente.

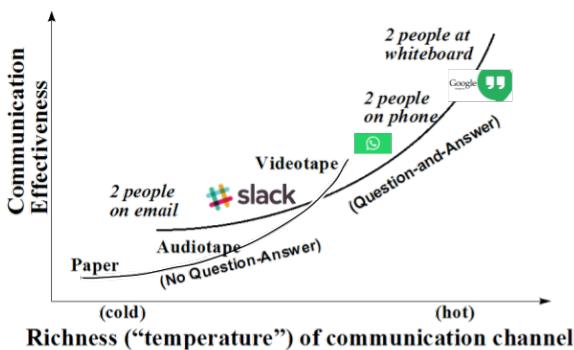


Gráfico original de “Efectividad de la comunicación” (<https://g.co/kgs/RVkx3i>) con nuevas herramientas (la evaluación de su efectividad está basada en un juicio propio).

En 10Pines procuramos que los equipos se comuniquen de modo efectivo: se trabaja en un *open space*, donde los integrantes se sientan juntos de modo de maximizar la comunicación osmótica (Cockburn, 2006). Cuando trabajamos desde casa, nos reunimos virtualmente usando *Hangouts* o *Skype*. Usamos herramientas para chatear y para gestionar el proyecto, como *Jira* o *Pivotal Tracker*. Desarrollamos haciendo *pair programming* y *code review*. Todas estas prácticas y herramientas resultan esenciales para mejorar la comunicación.

Muchos de los equipos en los que trabajamos son distribuidos, conformados con desarrolladores de 10Pines y *product owners*, *testers* y *scrum masters* en otras ubicaciones del globo. En estos casos, analizamos y definimos cómo lograr la comunicación más eficiente posible, reconociendo que, cuando las personas no comparten la misma ubicación, se genera un menor caudal de información. Para lidiar con esta situación, multiplicamos las reuniones y hacemos un uso, aún más, intensivo de las herramientas digitales.



## Conclusión

No buscamos personas para un proyecto en particular, sino personas que deseen desarrollar su vida profesional dentro de 10Pines. Nuestro

proceso de selección evalúa fuertemente los aspectos técnicos sin dejar de lado el aspecto humano.

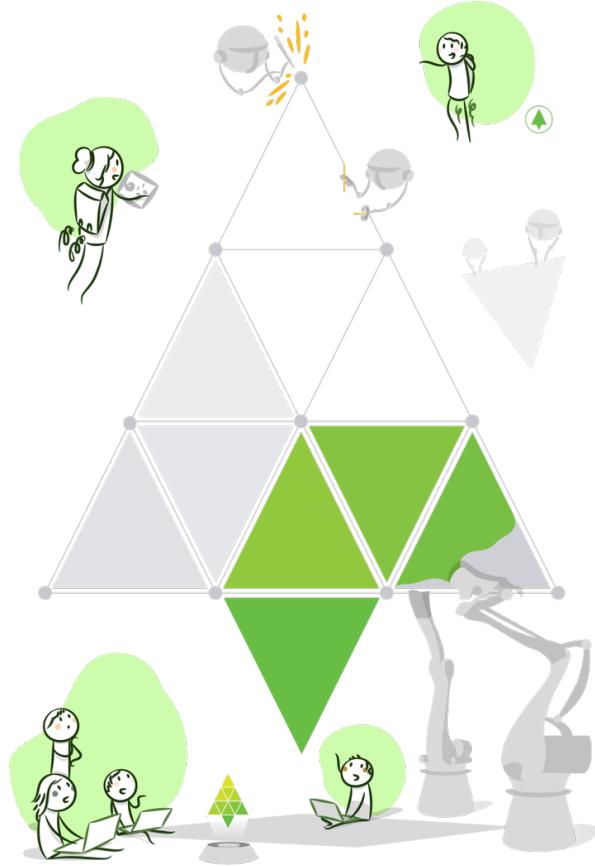
Creemos que es fundamental sentirnos bien, trabajar en un ambiente transparente, honesto y colaborativo porque permite brindarnos al máximo. Continuamente evaluamos cómo nos sentimos y qué podemos mejorar. Poseemos un camino de crecimiento, que llamamos “Camino del Pino”, que facilita nuestro desarrollo. Poder crecer profesionalmente impacta en nuestra felicidad.

Construimos una visión compartida, tanto en 10Pines como en cada uno de los productos que desarrollamos, que nos permite estar alineados y colaborar eficientemente.

El empoderamiento resulta esencial para desarrollar el mejor producto posible y para gestionar una empresa. Además, nos compromete, disciplina, motiva y hace felices.

Finalmente, consideramos a la comunicación vital en el desarrollo de software. Consecuentemente, buscamos maximizar el *bandwidth* continuamente.

# Desarrollando Software, Paso a Paso



## Introducción

En capítulos anteriores, descompusimos nuestro trabajo en incrementos funcionales, que llamamos *User Stories*, con el objetivo principal de involucrar a los usuarios de negocios durante la construcción del producto. En este capítulo describiré el proceso de desarrollo iterativo e incremental: cómo construimos esas *User Stories*, por qué usamos iteraciones de duración fija y, finalmente, por qué trabajar de este modo resulta importante para lograr nuestros objetivos.

## Desarrollo iterativo e incremental

Empezamos nuestro desarrollo por las *User Stories* que priorizamos en nuestro backlog, que iremos construyendo gradualmente para obtener feedback rápido. Algunas de estas *stories* son bosquejos, esqueletos no terminados, partes incompletas de funcionalidad, que necesitan trabajo adicional para ser finalizadas. Otras son incrementos funcionales completos, listos para implementarse en producción. Las primeras corresponden a la estrategia iterativa. Las segundas, a la estrategia incremental. El proceso de desarrollo iterativo e incremental combina ambas estrategias.

Jeff Patton, en un viejo post<sup>63</sup>, las detalla preguntándose cómo haríamos para pintar la Mona Lisa si lo hicieramos de uno u otro modo.

Si lo hicieramos de modo 100% iterativo, empezaríamos por dibujar un bosquejo del cuadro completo (paso 1), que iríamos refinando en versiones posteriores:

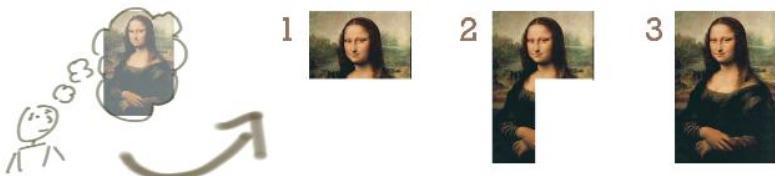


---

<sup>63</sup> [https://www.jpattonassociates.com/dont\\_know\\_what\\_i\\_want/](https://www.jpattonassociates.com/dont_know_what_i_want/)

Al momento de pintar cada una de las versiones intermedias, **necesitaríamos tiempo para iterar sobre ésta**, es decir, cambiarla, mejorarla. Alistair Cockburn describe este proceso como **una estrategia de retrabajo<sup>64</sup>**. En cada una, agendamos tiempo para revisar y mejorar el cuadro.

¿Y si lo hiciéramos de modo 100% incremental? Descompondríamos el cuadro en diferentes partes, que iríamos pintando gradualmente:



Cada una de estas partes estaría completa, sin necesidad de retrabajo. Tendrían calidad de producción. Alistair describe este proceso como **una estrategia de planeamiento**. La alternativa sería hacer un desarrollo completo que contenga toda la funcionalidad.

El proceso de desarrollo iterativo e incremental combina las 2 estrategias. Algunas de nuestras *stories* son bosquejos que necesitan retrabajo. Por ejemplo, podemos hacer una *User Story* para un formulario de alta, sabiendo que, más adelante, incorporaremos las validaciones y el *look & feel*. Otras son incrementos funcionales completos que podrían implementarse en producción una vez desarrollados. Durante el comienzo del proyecto, abordaremos las *stories* correspondientes a la estrategia iterativa, ya que necesitamos construir el esqueleto de la aplicación, su arquitectura. A medida que progresemos con el desarrollo, trabajaremos con más *stories* que correspondan a la estrategia incremental.

---

<sup>64</sup> <http://www.se.rit.edu/~swen-256/resources/UsingBothIncrementalandIterativeDevelopment-AlistairCockburn.pdf>

## Trabajando en iteraciones de duración fija

Con el plan y las *User Stories* que tenemos, podríamos empezar con el desarrollo. Sin embargo, el horizonte apuntado quizás resulte demasiado lejano. ¿Podremos mantener el foco durante esos 3 meses que, estimamos, durará el desarrollo del MVP? Difícil.

Para lidiar con ésto, podemos dividir esos 3 meses en *slots*, de duración fija (por ejemplo 2 semanas), que se denominan iteraciones (*Sprints* en *Scrum*). En el comienzo de cada una de ellas, decidimos un conjunto de funcionalidad que consideramos apropiado para ese periodo. Con un horizonte tan cercano, **nos enfocaremos en terminar este subconjunto de funcionalidad** que nosotros mismos planeamos, sin desconcentrarnos con *features* futuros.

Al finalizar cada iteración “paramos la pelota”, como suele decirse en el ámbito futbolístico, **para medir la capacidad real del equipo** (su *velocity*) y también **para inspeccionar qué funcionó y qué no**. Usamos esta información para establecer acciones de mejora y modificar el plan.

Con el correr de las iteraciones, logramos un ritmo. Nos acostumbraremos a planear, enfocarnos en terminar el trabajo, hacer introspección y comenzar el ciclo nuevamente. Los agilistas llamamos a esto cadencia.

### ¿Siempre usamos iteraciones fijas?

Me gusta trabajar con iteraciones fijas (*timeboxed iterations*), sobre todo, en proyectos como los descritos en este libro. Creo que el equipo gana foco en periodos cortos, mejora el proceso entre las iteraciones y mantiene una cadencia. Sin embargo, tenemos otros clientes, que desarrollan mejoras evolutivas o soporte de producción, donde no las usamos. En estos casos, al suprimir las iteraciones, estamos pasando a un sistema “*pull*” donde, en vez de “empujar” un conjunto de trabajo hacia una iteración, lo “pulleamos” (una *User Story*) cuando tenemos la capacidad. En sistemas Kanban como el previamente descripto, el foco se mantiene a partir del límite de trabajo en progreso, la capacidad se mide a partir del

*leadtime* y el *throughput* y la cadencia se logra a partir de otros mecanismos<sup>65</sup>. Lo esencial, trabajar iterativa e incrementalmente con equipos multidisciplinarios y auto-organizados, se mantiene.

## El trabajo de una iteración

Como ya les comenté en la sección anterior, el equipo se auto-organiza para seleccionar el conjunto mínimo de *User Stories* que puede construir eficientemente. Los desarrolladores somos quienes mejor conocemos el tamaño de éstas y quienes mejor podemos dividir y organizar el trabajo que debe realizarse.

Intentaremos terminar la mayor cantidad de *User Stories*, abordándolas en base a su prioridad. **La heurística que propongo para lograr este objetivo consiste en maximizar el flujo, minimizando el trabajo en progreso.** ¿Qué significa? Trabajar en la menor cantidad de *User Stories*, siempre que sea eficiente. ¿Cuántos desarrolladores pueden trabajar eficientemente en cada *story*? Nuevamente, el equipo es quien mejor sabe cómo responder esta pregunta.

Antes de empezar cada *story*, reunimos a los desarrolladores involucrados y los integrantes de negocios para refinar los últimos detalles. Se pulen los bordes. Se discute sobre los argumentos que vuelven necesaria esa funcionalidad, una vez más. También, cómo se testeará y validará. Entramos así de lleno en el diseño, que hacemos con la colaboración de todos los integrantes del equipo.

Desarrollamos pensando, en primer lugar, en los tests automatizados que validarán la funcionalidad y procurando obtener “calidad de producción”, como dice la guía de *Scrum*. Este es el momento de hacer el producto con calidad. La calidad se “hornea desde el principio” (Poppendieck00, 2003), no puede incorporarse al final.

Una vez que terminamos el desarrollo y testeо de una *User Story*, realizamos un proceso de aceptación formal, donde repasamos, una vez más, el cumplimiento de todos los criterios de aceptación. También

<sup>65</sup> <http://www.djaa.com/kanban-cadences>

podemos repasar nuestra “*definition of done*”<sup>66</sup> para estar completamente seguros de que hemos terminado.

Todo este proceso, altamente concurrente, involucra un grado de comunicación muy intenso. El *Product Owner* y los usuarios de negocios trabajan de la mano con los desarrolladores para construir el producto que desean, comunicando claramente y validando cada incremento. Todo este feedback, brindado a tiempo, se incorpora dentro del producto.

## ¿Qué logramos trabajando de este modo?

El desarrollo iterativo e incremental es uno de los pilares fundamentales de la agilidad. Repasemos lo que logramos, al trabajar de esta manera:

**Crear un plan ágil:** esto es, como ya les comenté en el capítulo correspondiente, un plan liviano, orientado a *features*. Trabajar de este modo nos ahorra análisis detallados, que no aportan valor.

**Recibir feedback temprano:** terminar incrementos funcionales rápidamente permite que el *Product Owner* se involucre en el desarrollo, brindando un feedback útil para moldear el producto de acuerdo con sus necesidades. Iterar amplifica el aprendizaje<sup>67</sup> y maximiza el valor del producto final.

**Aumentar la calidad:** se brinda un feedback rápido acerca de la calidad a partir del testeo de cada incremento, en lugar de hacerlo al final. Incorporar la calidad buscada dentro del proceso de construcción resulta fundamental.

**Adaptarnos al cambio:** uno de los objetivos más importantes buscados por la agilidad es poder adaptarnos al cambio. Cuando trabajamos de este modo, podemos hacerlo eficientemente.

---

<sup>66</sup> <https://www.agilealliance.org/glossary/definition-of-done/>

<sup>67</sup> <http://www.poppendieck.com/pdfs/AmplifyLearning.pdf>

**Hacer releases parciales:** Finalmente, este modo de trabajo permite realizar *releases*, cuando completamos un conjunto de *user stories* que, juzgamos, representan un ciclo completo de funcionalidad.

¿Tiene alguna desventaja trabajar de este modo? La primera que puedo nombrarles es la eficiencia. Iterar sobre una funcionalidad implica retrabajo. Por otra parte, estos incrementos, desarrollados por un equipo multidisciplinario, generan una concurrencia alta con el consecuente costo de comunicación. En contrapartida, el desarrollo en cascada divide el trabajo en fases en las cuales no es necesario que todos trabajen, haciendo que la concurrencia sea menor y que, por ende, aumente la eficiencia. No considero que esta sea una ventaja válida, principalmente porque no redundá en un mejor producto. Además de perderse los beneficios anteriormente enumerados.

## Conclusión

Construimos el software gradualmente, a partir de incrementos funcionales pequeños con el objetivo de recibir feedback temprano. Trabajamos en iteraciones de duración fija para ganar foco, medir nuestra *velocity* y establecer una cadencia. El desarrollo iterativo e incremental es uno de los pilares fundamentales de la agilidad.

# Desarrollando Software con Excelencia Técnica



## Introducción

La excelencia técnica es un principio del manifiesto ágil. En 10Pines, este principio forma parte de nuestro ADN. Constituye uno de nuestros pilares de calidad. No perseguimos un perfeccionismo vacío. Creemos que es lo mejor para nuestros clientes: si desarrollamos código, bien diseñado y testeado, se producirá un ahorro de dinero, facilitándose los incrementos de funcionalidad y los cambios de modo sustentable. Por el contrario, trabajar sobre sistemas *legacy* implica un costo muy alto, además de resultar poco reconfortante para quienes debemos trabajar con él.

Introducimos la excelencia técnica desde el momento mismo en que comenzamos con el desarrollo. No creo que sea factible hacerlo después. El desarrollo de software es un proceso de aprendizaje durante el cual debemos mantener el código limpio.

Pagamos la deuda técnica, refactorizando continuamente. Si no incorporamos el conocimiento que generamos en nuestro *codebase*, este se convertirá en código *legacy* muy rápidamente y ya no podremos seguir incorporando funcionalidad o adaptarnos a los cambios fácilmente.

Todo el código que desarrollamos posee tests automatizados, que certifican la funcionalidad y nos permiten hacer pruebas de regresión eficientemente.

¡Exploremos los aspectos técnicos del desarrollo de software!

## Clean Code

Ron Jeffries dijo que nuestro trabajo consiste en escribir “código limpio que funcione”. ¿Qué es *clean code*? Extraigo esta definición del libro homónimo de Uncle Bob (Martin, 2008): **A nivel alto, es código bien estructurado, que sigue buenos principios de diseño. A nivel bajo, es código cuyas abstracciones tienen nombres relevantes y funciones elegante. Está bien formateado y contiene comentarios útiles** (si todo el resto se cumple, estos últimos no son necesarios). Expresa su intención de modo claro. Es fácil de entender y, por supuesto, no está duplicado.

Posee tests automatizados que certifican que hace lo esperado. El feedback continuo, rápido y barato que estos tests brindan nos permite agregar funcionalidad o refactorizar sin temor a romper nada. De hacerlo, nos enteraríamos rápidamente (con los cambios aún en nuestras cabezas), permitiéndonos corregir el error sin esfuerzo. Tener buenos tests también es parte del código limpio.

## Deuda técnica

Ward Cunningham<sup>68</sup> acuñó este concepto estableciendo una metáfora del mundo financiero, justamente porque debía explicar la necesidad de un *refactor* a un *Product Owner* procedente del mundo de las finanzas.

La deuda técnica es un préstamo que nos permite aprender rápidamente. Todos podemos necesitar o requerir de un préstamo en algún momento puntual de nuestra vida económica o laboral. Sin embargo y al igual que en otros planos, debemos pagar las cuotas pactadas. De lo contrario, los intereses se acumularían, haciendo su cancelación cada vez más difícil.

Como programadores, muchas veces escribimos código para aprender sobre algún concepto. Al hacerlo, debemos empezar a pagar nuestra deuda, refactorizando el código para que refleje el conocimiento adquirido. Si no lo hicieramos, sucedería exactamente lo mismo que con un préstamo monetario, ya que no estaríamos pagando las cuotas: el código sería cada vez más difícil de comprender y modificar y, consecuentemente, se volvería más costoso agregar funcionalidad.

## Costo de cambio

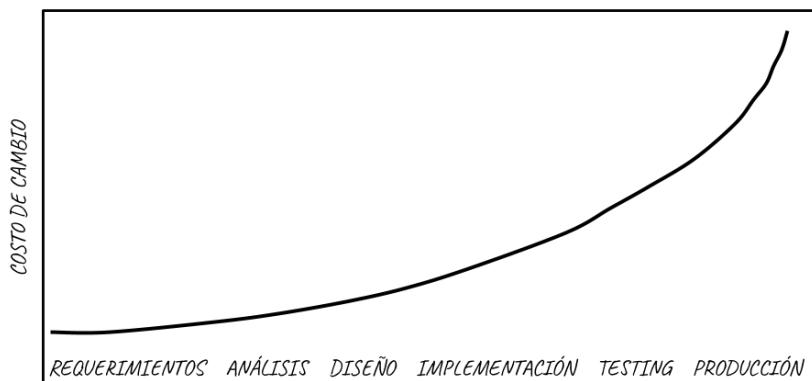
Debo confesar que la primera vez que trabajé usando eXtreme Programming me resultó tedioso. Trabajar en incrementos funcionales pequeños, hacer

---

<sup>68</sup> Ward, un personaje esencial en el nacimiento del movimiento ágil, pero de un perfil muy bajo hizo un video (<https://www.youtube.com/watch?v=pqeJFYwnkjE>) que resume este concepto de modo mucho más claro.

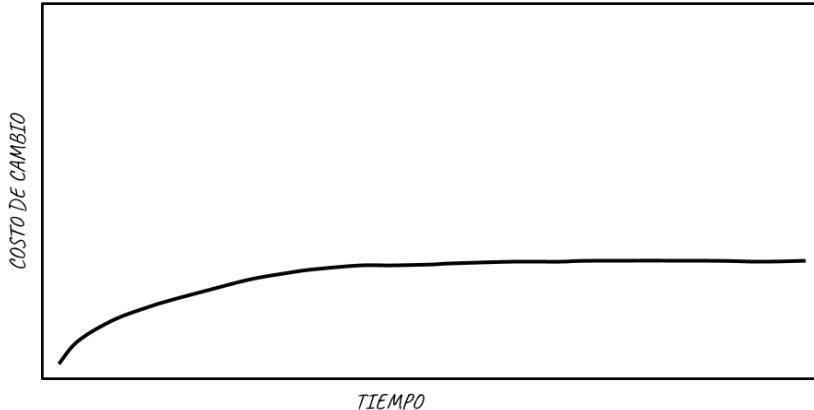
*pair programming*, diseñar, testear, refactorizar y revisar el código tan detallada y minuciosamente no fue sencillo. Con los años, fui reconociendo que otras personas tendrían que “lidiar” con “mi” código durante mucho tiempo. También aprendí acerca de la importancia de mantener el *codebase* en buenas condiciones para poder agregar funcionalidad sustentablemente y cambiar la existente con celeridad y confianza.

Al mismo tiempo, llegó a mis manos un libro de esos que producen un quiebre en la vida profesional: “*Planning eXtreme Programming*” (Beck, Planning eXtreme Programming (The Xp Series), 2000). El concepto que deseaba referir aquí es el de “costo de cambio”. El autor comienza por un estudio publicado por Barry Boehm a comienzos de los 80s donde sugiere que corregir un error se vuelve exponencialmente más costoso mientras más tarde se detecte (de aquí proviene la famosa frase: “un error es 100x más costoso de corregir en producción que al momento de realizar la especificación”).



Versión propia basada en el gráfico de costo de cambio (Beck, Planning eXtreme Programming (The Xp Series), 2000)

Beck sugiere que con “la combinación de tecnologías y prácticas de programación” descritas, la curva puede achatarse de esta manera:



Versión propia basada en el gráfico de Costo de Cambio (Beck, Planning eXtreme Programming (The Xp Series), 2000)

Este concepto fue, para mí, revelador. Quizás Beck pueda haber exagerado un poco (como sugiere Ambler<sup>69</sup>), pero lo crucial es destacar la importancia de mantener el costo de cambio tan bajo como sea posible para poder incorporar funcionalidad sustentablemente, adaptándonos al cambio. Mantener el costo de cambio bajo es, en definitiva, un requisito fundamental si queremos ser Ágiles. Si no pudiéramos hacerlo, el resto de las prácticas perderían sentido. Dicho de otro modo, si incorporar funcionalidad se volviese cada vez más costoso, difícilmente podríamos mantener la agilidad.

## Algunas prácticas esenciales

### *Testing automatizado*

Todo el código que escribimos es acompañado por tests automatizados que se ejecutan continuamente. Permiten tener la mejor *Velocity* posible y desarrollar sustentablemente. Representan la mejor documentación del

---

<sup>69</sup> <http://www.agilemodeling.com/essays/costOfChange.htm>

sistema y adicionalmente soportan el resto de las prácticas de ingeniería que permiten alcanzar la excelencia técnica que buscamos.

Veamos cada una de estas razones con mayor profundidad:

**Velocity.** Los tests automatizados son fundamentales porque nos permite desarrollar incrementos de funcionalidad muy rápidamente, sin temor de introducir *bugs*. Permiten enfocarnos en el diseño que resuelve el incremento de funcionalidad actual, alivianando la inmensa complejidad que deberíamos mantener en nuestras cabezas. Al completar la funcionalidad, el conocimiento adquirido queda plasmado en nuevos tests, que generan un círculo virtuoso. Adicionalmente, recibimos feedback casi inmediato de los cambios. Esto amplifica nuestro aprendizaje, nos permite iterarlo rápidamente y plasmarlo en nuevos tests que seguirán generando conocimiento de un modo eficiente. Cuando cometemos un error, los tests nos alertan instantáneamente, pudiendo corregir el problema sin esfuerzo (porque todavía tenemos los cambios realizados en nuestras cabezas) y evitando la introducción de *bugs* que pueden resultar costosos de reparar en el futuro.

**Sustentabilidad:** Los tests automatizados son fundamentales para poder seguir incrementando la funcionalidad del sistema sustentablemente, es decir, por un periodo de tiempo largo y a un costo razonable. Durante la construcción del MVP, nos permiten completar *User Stories* sin temor de haber introducido *bugs* en la funcionalidad existente, que se incrementa iteración a iteración. Si, por el contrario, el testing de regresión fuera manual, el esfuerzo se incrementaría linealmente. Cada vez tendremos más funcionalidad que testear. Y si decidíramos no testear la funcionalidad existente, no reconoceríamos la introducción de *bugs*. En definitiva, no podríamos asegurar que estamos haciendo progreso. Brooks llamó a este fenómeno “un paso para adelante y dos para atrás”. Pensemos ahora en la sustentabilidad del producto después de la implementación: si tenemos suerte, vivirá muchos meses o años durante los cuales será necesaria la incorporación de nueva funcionalidad o el cambio de la existente. Sin una buena suite de tests de regresión, sería imposible.

**Adaptabilidad:** Para poder ser ágiles, debe ser posible cambiar. Si el costo de cambio fuera grande, sería imposible. Los tests permiten realizar cambios rápidamente, sin temor a equivocarnos. En sistemas *legacy*, que

carecen de una buena suite de tests automatizados, no tenemos esa capacidad. La introducción de cambios en la funcionalidad se torna lenta, riesgosa y costosa, con el consecuente impacto en el negocio.

**Documentación viva:** los tests representan la mejor documentación posible del producto, documentación viva (así la denominó Gojko Adzic en (Adzic, Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing, 2009), que siempre está al día. Si algo cambiara y no modificáramos los tests, estos se romperían. Además, es documentación que continuamente nos brinda feedback, ya que todo el tiempo se ejecuta. Mejor aún, es documentación que podemos *debuggear* para entender el código.

**Soportan otras prácticas:** Finalmente, los tests automatizados nos brindan la seguridad y el feedback inmediato necesarios para realizar un diseño emergente, refactorizar continuamente y para que cada miembro pueda trabajar en cualquier área del código (*collective code ownership*<sup>70</sup>). En definitiva, soportan prácticas fundamentales en nuestro objetivo de alcanzar la excelencia técnica.

Para nosotros, en 10Pines, escribir tests es un hábito. No concebimos el desarrollo de software de otro modo (por esta razón lo evaluamos tan estrictamente durante el proceso de selección). Nos dan el feedback para programar correctamente. Constituyen una práctica esencial para construir el producto con excelencia técnica.

### *Una estrategia para el testing automatizado*

Hace ya muchos años, leí un libro de Testing Ágil (Crispin & Gregory, 2009) que me dejó un concepto que siempre valoré: **es importante tener una “estrategia de testing automatizado” para el proyecto.** Lisa y Janet argumentan que cada uno es diferente (negocio, tecnologías, arquitectura, equipo, etc.) y que, por ende, tendrá una estrategia diferente, que debería definir:

- Tipos de tests (unidad, integración, funcionales) y proporciones (donde pondremos más esfuerzo).

---

<sup>70</sup> <https://martinfowler.com/bliki/CodeOwnership.html>

- Tecnologías (*frameworks*, herramientas)
- Proceso (cómo afectará nuestra *definition of done*)

Una vez definida la estrategia, deberíamos poder contestar la pregunta: ¿Cómo incrementar la funcionalidad de manera sustentable y con una buena *Velocity*?

Desde su lectura, han pasado muchos años y han cambiado muchas cosas. Muchos de nuestros clientes trabajan ahora con equipos distribuidos en diferentes países, haciendo *continuous delivery*. Charlamos mucho de la importancia de la estrategia con Matías Fernández, un colega, líder en uno de estos equipos. Concluimos que, para estas situaciones, la estrategia necesita responder algunas preguntas más:

- Cómo incrementar la funcionalidad sin que el tiempo del *build* se dispare (en un producto grande con miles y miles de tests, este tiempo puede volverse extenso, repercutiendo negativamente en la productividad).
- Cómo lograr que todo el equipo comparta la filosofía de testing, tanto en lo que respecta a la proporción de tipos de tests como en la manera de escribirlos.
- Cómo testear el *deployment*. Cómo asegurar que, con cada nuevo deploy, no se introduzca ningún problema, teniendo en cuenta que se hacen múltiples *deploys* diarios. Una anécdota que me llamó mucho la atención fue el descubrimiento de un *bug* pequeño, casi imperceptible, pero de un alto impacto económico. ¿Por qué? Porque se producía exactamente en el momento de *checkout*. No se trataba de un error de esos que dejan el servicio fuera de operación, pero provocó pérdidas mucho mayores. Es importante que la estrategia permita sentirnos confiados de que estas cosas no van a ocurrir.
- Qué hacer cuando se reporta un error en producción. Cómo determinar la gravedad y el mejor momento para corregirlo (existe errores que deben corregirse inmediatamente porque imposibilitan el uso del producto y otros que no vale la pena siquiera corregir).

Matías y el resto del equipo hicieron un trabajo excelente, estandarizando la proporción de tests, unificando los enfoques de testing (a través de *pair*

*programing y code reviews*), asegurándose que el tiempo del *build* se mantuviera razonable (a través de la optimización del servidor de CI y del tiempo de los tests, etc.) y también a partir del diseño e implementación de un proceso para filtrar errores de producción y agendar su corrección. La estandarización de todos estos puntos en una “estrategia de testing automatizado” del producto produjo una mejora sensible en el proceso de desarrollo que repercutió en la agilidad del negocio.

### *Proporción de tipos de tests*

Uno de los puntos a definir en la estrategia es qué tipos de tests construiremos y en qué proporción. Mike Cohn publicó una heurística, que ahora conocemos como la pirámide de testing de Cohn. La pirámide está compuesta por 3 tipos de tests: unitarios, integración y GUI. Cada uno de ellos ocupa un área de la pirámide, que metaforiza la proporción recomendada de ese tipo de tests:



Pueden observar que la heurística sugiere que la proporción de tests unitarios debería ser la mayor, ya que son rápidos y robustos. Luego, seguirían los de integración, en menor cantidad, pues sólo deberían testear el funcionamiento de las integraciones entre componentes ya testeados unitariamente. Finalmente, en la punta y con el área más pequeña los tests de *UI*, cuya proporción debería ser la menor: son los más lentos, ya que debemos levantar todo el *stack* tecnológico para ejecutarlos y los menos robustos, pues se rompen con cambios en cualquiera de las capas (en mi experiencia, cuando fallan, resulta difícil la detección del problema, pues son difíciles de *debuggear* y el logueo del error y de la ejecución son menos descriptivos).

En 2011, Gojko Adzic publicó un libro llamado “Especificaciones con Ejemplos” (Adzic, Specification by Example: How Successful Teams Deliver the Right Software, 1st Edition, 2011), que generó un gran movimiento en la comunidad. La idea es que estos ejemplos, escritos por el *Product Owner*, se conviertan en los tests automatizados que guíen el desarrollo (usando herramientas como *cucumber*<sup>71</sup> y *capybara*<sup>72</sup>). El beneficio es claro: se involucra en la creación de los ejemplos, que posteriormente se convierten en documentación viva, que está siempre actualizada y que ejercita el código constantemente. Sin embargo, en mi experiencia personal, nunca pude ver un equipo donde estos tests realmente guíen el desarrollo. Los problemas que observé son:

- El *Product Owner* no se involucraba por completo en la creación de los ejemplos, que terminaban escritos por otros integrantes del equipo (como el tester o el analista). En consecuencia, ¿tiene realmente sentido que el test esté escrito en lenguaje natural? En estos casos, lo mejor es que el equipo técnico decida como escribir estos tests.
- Se escriben demasiados tests de este tipo, contradiciendo la pirámide previamente descrita. Según expertos consultados, esto no debería pasar porque los ejemplos deberían representar los casos más relevantes.

<sup>71</sup> <https://cucumber.io/>

<sup>72</sup> <https://github.com/teamcapybara/capybara>

Por todo esto, prefiero construir los tests funcionales para los ejemplos más relevantes una vez que la funcionalidad esté desarrollada.

## *Test Driven Development - por Hernán Wilkinson*

¿Qué es *Test Driven Development*? O para hacerlo más simple, TDD. ¿Cuándo hay que usarlo y cuándo no? ¿Para qué sirve y para qué no? ¿Se puede usar siempre, en cualquier sistema, en cualquier contexto? Si desarrollamos con TDD, ¿es necesario tener QA? Estas y muchas preguntas más son las que recibimos cada vez que hablamos o damos un curso de TDD. Voy a tratar de responderlas en esta sección. No será fácil. Seguramente algunas preguntas queden sin responder y se generen otras. Está bien que así suceda: es parte de todo crecimiento intelectual la generación de dudas. Sin ellas no se aprende, no se crece. En esta introducción les dejo mi primer consejo: ante la duda, practiquen, prueben y jueguen con el tema. Aprendan haciendo.

Nuestra querida profesión está bendecida y maldecida al mismo tiempo por una característica esencial, como diría Fred Brooks en "*No Silver Bullet*": la "maleabilidad" del producto que generamos, la facilidad para cambiarlo, para experimentar con él, y, como decía Marvin Minsky en su paper "*Why Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas*", para probar prácticamente las teorías de conocimiento que tenemos sobre un problema.

La práctica, en nuestra profesión, resulta esencial para aprender. Espero generar muchas dudas en ustedes, pero sobre todo que puedan responderlas ustedes mismos programando. Es la mejor manera de aprender.

Me gustaría empezar con un poco de historia. Conocerla ayuda a entender cómo surgió, en este caso TDD, y qué podemos esperar del futuro. Podríamos marcar el comienzo "formal" de TDD en octubre del año 1994 cuando Kent Beck publica en la *Smalltalk Report* (una revista dedicada al lenguaje *Smalltalk* de la década del 90. Sí, en los 90 había revistas de lenguajes de programación) el artículo "*Simple Smalltalk Testing*", donde describe un *framework* de "*unit testing*" que estaba utilizando para verificar que el código escrito funcionase como esperaba. Kent Beck

centra su atención en "*testing*", o sea en escribir tests luego de que el "código" estuviese desarrollado. Más allá de eso, fue la semilla que terminó evolucionando en lo que conocemos como TDD. Este *framework* de *testing* es el que luego evolucionó como *SUnit* y del cual derivan aquellos correspondientes a otros lenguajes de programación como JUnit para Java, NUnit para .Net, etc.

Beck comenta que en el proceso de “germinación” de TDD influyeron mucho las enseñanzas o vivencias de su padre quien fue desarrollador de software y quien le inculcó la importancia de la verificación de lo que uno hace.

Recuerdo que, allá por el año 1998, mientras trabajaba como consultor de la AFIP, otorgué a una desarrolladora de mi equipo la responsabilidad de implementar un "*framework de testing*" en Java, idea que había sacado, claro está, del artículo mencionado. Ya se empezaba a vislumbrar, a sentir, por entonces la importancia del testing automatizado. Dicha programadora, cuyo nombre lamentablemente no recuerdo, realizó un excelente trabajo que no prosperó en el resto del grupo de trabajo por dos razones: 1) la aversión de los programadores a testear, ¿cómo un programador iba a "perder tiempo" testeando? Para eso estaba QA, para algo se les pagaba a ellos. 2) Dejé de trabajar en la AFIP :-)

Kent Beck terminó de dar forma a la idea de TDD y aplicarla en ese famoso proyecto C3 de la empresa *Chrysler*, desarrollando en Smalltalk<sup>73</sup>

<sup>73</sup> No puedo dejar de hacer un comentario relacionado a mi "amor" declarado por Smalltalk. Puede ser poco objetivo, pero no podemos negar la cantidad de cosas interesantes que se generaron a partir de desarrolladores relacionados a la cultura "Smalltalkera". En el caso particular del proyecto C3, no solamente usaban Smalltalk como lenguaje de programación, sino GemStone, una base de objetos IMPRESIONANTE. Tuve la suerte, la dicha, de desarrollar software con esa tecnología y puedo decir, sin pudor, que nunca me sentí tan cómodo ni tan productivo (pese a haber utilizado muchas tecnologías y lenguajes, como assembler, C, Pascal, C++, Java, C#, Python, Ruby y muchos más).

(Beck, Test Driven Development: By Example, 2002)

un sistema de *payroll* del cual surge no sólo la idea de TDD sino también de XP (*eXtreme Programming*).

En 1999, Beck edita "*eXtreme Programming Explained*" donde, entre otras cosas, "eleva" la importancia del testing automatizado, hablando principalmente de "*unit testing*" (tests unitarios). En mi caso, que venía siguiendo a Beck en todas sus publicaciones, no dudé en comprar el libro y leerlo inmediatamente. Quedé maravillado con lo expuesto y lo implementé rápidamente. No me tuvo que convencer, lo tratado por el texto era "lo natural". Por fin alguien se animaba a decirlo y rompía con los mandatos "taylorianos" y "cascadosos" a los que nos tenían acostumbrados los supuestos gurúes de la "ingeniería de software".

Sin embargo, la técnica no quedará formalizada hasta el año 2000, con la edición de "*TDD By Example*" (Beck, Test Driven Development: By Example, 2002). Beck estipula 3 pasos para hacer TDD:

1. *Red Step*: Escribir un test simple que falle al ser corrido.
2. *Green Step*: Implementar sólo lo necesario **y nada más que lo necesario** (el destacado es mío) para que el test pase, cometiendo todos los "pecados" de programación y diseño que se nos ocurran.
3. *Refactor Step*: ¡Confesarse! Beck no lo plantea con estas palabras, sino que propone eliminar la duplicación creada en el paso 2. En definitiva, "sacarse los pecados de encima" o, en términos más técnicos, hacer un buen diseño.

Como se puede ver la técnica no es compleja. Consta de tres pasos simples, fáciles de recordar y explicar, que, sin embargo, resultan difíciles de seguir y de aplicar (veremos más adelante por qué).

Estos tres pasos no explican qué es TDD, sino que definen cómo se hace. ¿Qué es TDD entonces? ¿Cuál es su esencia? ¿Qué hace que TDD sea lo que es y no sea otra cosa? TDD es una técnica de desarrollo de software, iterativa e incremental, con feedback inmediato. En pocas palabras, una técnica que favorece el proceso de aprendizaje que todo ser humano lleva adelante cuando debe entender, explicar y formalizar qué es y cómo funciona algo. En definitiva, cuando debe programar.

Para comprender bien TDD y poder aplicarlo correctamente, no podemos perder de vista que debemos desarrollar de manera iterativa,

agregando en cada iteración un nuevo incremento a nuestra solución para obtener feedback rápido que valide que estoy en el camino correcto.

Veamos esto con un ejemplo. En la introducción comenté la importancia de practicar para aprender y quiero ser consistente con lo que dije (aunque a veces la inconsistencia me supera). Por ello, para explicar cada paso y sus causas, utilizaré un ejemplo sencillo, el de *Conways Game Of Life*<sup>74</sup>. Básicamente consiste en un tablero bidimensional compuesto por celdas vivas o muertas, donde se van creando nuevas generaciones a partir de las siguientes reglas:

1. Una celda viva con menos de dos vecinos vivos muere (subpoblación).
2. Una celda viva con dos o tres vecinos vivos sigue viva.
3. Una celda viva con más de tres vecinos vivos muere (sobre población).
4. Una celda muerta con exactamente tres vecinos vivos resucita (reproducción).

Dado este problema, ¿cómo empezamos? La respuesta es clara: hay que escribir el test más simple que se nos ocurra. ¿Por qué debe ser el test más simple? Porque la idea es no caer en análisis-parálisis: tener algo funcionando lo más rápido posible para entrar en un flujo de trabajo virtuoso y sostenible. Además, para poder resolver un problema grande debemos particionarlo. Es conveniente comenzar por la parte más chica y simple de todas ya que nos asegura una solución rápida. Estos motivos nos brindan pistas acerca de lo que debemos medir cuando estamos haciendo TDD, para saber si lo estamos haciendo correctamente.

Sí, TDD nos da un método de trabajo medible, predecible y que, por lo tanto, nos permite reflexionar y mejorar a partir de dichas mediciones. Nótese que esto no es así con la técnica "clásica" de desarrollo donde, de manera poco clara y a veces caótica, se empieza por la implementación de una clase, luego se pasa a otra y posteriormente se retorna a la anterior sin orden ni guía, más allá de la idea mental acerca de la solución que deseamos generar. Con TDD esto no sucede: existen pasos bien

<sup>74</sup> [https://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway's_Game_of_Life)

estipulados por lo que podemos reflexionar qué tan bien los estamos haciendo y actuar en consecuencia.

¿Cuál es la medición más importante que debemos realizar cuando hacemos TDD? El tiempo que nos lleva cada paso. Recuerden, queremos feedback inmediato. Por lo tanto, cuanto más rápido hagamos cada paso, mejor. Demorar mucho tiempo en escribir un test es un indicio de que algo no estamos haciendo bien. Posiblemente no estemos encarando el test más simple. En tal caso, deberíamos repensar qué testear. O quizás es el más sencillo, pero estoy tardando porque escribirlo resulta muy complejo debido a que el diseño del sistema no me ayuda. Por lo tanto, debería hacer el paso 3) primero (o sea refactorizar<sup>75</sup> el diseño para hacerlo más testeable).

Volviendo a nuestro ejemplo, ¿cuál sería el primer test que podríamos escribir? Veamos estas opciones que aparecen generalmente en el público de los cursos que dicto:

1. Verificar que el juego se cree correctamente.
2. Verificar que, si creo un juego sin celdas vivas, ninguna celda esté viva.
3. Verificar que una celda viva con menos de dos vecinos vivos muera en la próxima generación.

---

<sup>75</sup> Recordar que el significado original de refactorizar es "modificar el diseño sin modificar la ejecución". Es una actividad que tiene por objetivo mejorar la vida del programador, hacer que entienda más fácilmente el programa, volver más declarativa la solución. A una computadora no le importa cómo se llama un objeto o un mensaje. A una persona, sí. La diferencia entre un buen nombre y uno malo puede ser determinante en la comprensión de una solución. Hago esta aclaración sobre el significado de esta palabra porque en nuestra profesión resulta muy común mutar los significados, cambiarlos. Últimamente, he notado una confusión entre "cambiar" y "refactorizar". Se utiliza la palabra refactorizar para indicar cambio, que no implica el mismo resultado de ejecución. Cambiar un programa puede incluir cambio a nivel ejecución. Refactorizar, no. Es una diferencia muy importante.

4. Verificar que una celda viva con dos o tres vecinos vivos sobreviva.
5. Verificar que todas las reglas se cumplan
6. ... y varias ideas más que omito por un tema de espacio y tiempo.

¿Cuál de todas ellas es la más simple? Descartemos de a poco. La 5) no pareciera ser la más simple porque para eso debemos tener las 3) y 4) más otras cosas implementadas. La 1) pareciera ser la más simple, ¡sólo tengo que crear un juego! Es verdad, sólo tengo que crear un juego. Pero, ¿qué verificamos de esa acción? En pocas palabras, ¿qué "asserto"? Veamos:

#### testGamesCreatedCorrectly

```
gameOfLife := GameOfLife new.  
self assert: gameOfLife ???
```

¿Qué ponemos en el ??? ? Algunos suelen decir “hay que verificar que no es *nil* o *null*”. ¿Realmente? ¿Cómo podría suceder que *gameOfLife* sea *nil*? Sólo podría ocurrir si nos quedáramos sin memoria, pero en ese caso ¿qué estaríamos testeando?, ¿el *GameOfLife* o cómo se comporta el lenguaje de programación respecto de qué hace cuando no hay memoria? Claramente lo segundo, que carece de importancia. Veamos la opción 2). Podríamos escribir el siguiente test:

#### testGameCreatedWithoutLiveCellsHasNoLiveCells

```
gameOfLife := GameOfLife withAliveCells: {}.  
self assert: gameOfLife aliveCellsIsEmpty
```

¿Qué les parece? Podría ser una opción. Pero, ¿cuánto avanzamos con este test? ¿Agrega algo a mi manera de entender el problema? La respuesta es completamente discutible y diferentes personas la responderían de distinto modo. Desde mi punto de vista, este test no agrega ningún valor. No dice nada sobre “qué hace” el *GameOfLife*, sobre la parte dinámica del juego, que es lo más importante de todo modelo computable (o sea, programa).

Por otro lado, básicamente estamos testeando un "getter". Podrán argumentar que no es un *getter*, sino un *test method*. Sin embargo, es casi lo mismo ya que podría haber escrito el test de la siguiente manera:

```
testGameCreatedWithoutLiveCellsHasNoLiveCells
```

```
gameOfLife := GameOfLife withAliveCells: {}.  
self assert: 0 equals: gameOfLife aliveCells size.
```

En este caso queda bien explícito que estoy testeando un *getter* y testear *getters* o *setters* no agrega ningún valor real a la solución.

Pregunta de diseño: ¿Por qué escribí el test sin usar un *getter*? ¿Cuál de los dos últimos tests es mejor desde el punto de vista de diseño de la solución? La respuesta es: el primero. ¿Por qué? Porque no estoy rompiendo el encapsulamiento de GameOfLife, mientras que sí lo hago en el segundo test. Romper el encapsulamiento es equivalente a sacarle responsabilidades a un objeto, por lo que esa responsabilidad se termina implementando en otros lugares de manera dispersa y seguramente repetida.

Nos quedan los tests 3) y 4). ¿Cuál es más simple? A mi manera de entender, el 3). El 4) implica que hay que testear si son 2 o 3 vecinos vivos, que resulta más complejo. Veamos cómo sería este test:

```
testAliveCellWithLessThanTwoAliveCellsDies
```

```
gameOfLife := GameOfLife withAliveCells: {1@1}.\ngameOfLife calculateNextGeneration.\nself assert: (gameOfLife isDead: 1@1)
```

Nótese que con este test ya tomé varias decisiones de diseño:

1. Crearé GameOfLife como una colección de celdas vivas, en este caso {1@1}.
2. Identificaré las celdas mediante puntos. En este caso 1@1 será la celda con posición x=1 e y=1, que además definiré como viva.

3. Para pasar de una generación a otra, enviaremos el mensaje `#calculateNextGeneration` a instancias de `GameOfLife`.

Si corremos este test, va a fallar (suponiendo que tenemos creada la clase y los métodos relacionados) ya que no hemos implementado ninguna lógica. Por lo tanto, ya cumplimos con el paso 1 de TDD: escribir un test, correrlo y que falle. En el paso 2, implementamos lo mínimo, y necesario, para que el caso que acabamos de testear pase. La implementación más simple que podemos hacer es:

```
GameOfLife class>>withAliveCells: aCollectionOfCells
  ^self new initializeWithAliveCells: aCollectionOfCells
GameOfLife>>initializeWithAliveCells: aCollectionOfCells
  "No hago nada"
GameOfLife>>calculateNextGeneration
  "No hago nada"
GameOfLife>>isDead: aCell
  ^true
```

Lo único que tengo que hacer para que el test pase es que `#isDead:` devuelva `true`. El primer método, `#withAliveCells:` está implementado como método de clase de `GameOfLife`, o sea que `#withAliveCells` es el mensaje que recibe la clase para crear una instancia de ella. Por eso la implementación primero envía el mensaje `#new` a `self` para obtener la nueva instancia y después envía el mensaje `#initializeWithAliveCells: a` la misma. Los mensajes `#initializeWithAliveCells:, #calculateNextGeneration` y `#isDead:` son mensajes que saben responder las instancias de `GameOfLife`.

Por supuesto que muchos argumentarán que esto parece un chiste, que no implementé nada, pero no es así. Para mi "estado de conocimiento", para el incremento del problema que estoy atacando, o sea el caso de "celdas vivas con menos de 2 vecinos vivos muere", el problema está resuelto. Por supuesto que hay otros problemas con esta implementación, pero recordemos que estamos haciendo un desarrollo iterativo-incremental, por lo tanto, debo preocuparme únicamente por los incrementos implementados, no por lo que me falta por implementar.

Ya cumplimos con el paso 2, ahora hay que hacer el paso 3, refactorizar. ¿Hay algo para refactorizar en esta implementación? La verdad, no. Es tan simple, tan concreta, que no es necesario refactorizar nada. Es conveniente hacer una aclaración sobre el paso 3: es el único de los pasos de TDD cuya aplicación es contextual. Siempre hay que hacer el paso 3 pero puede suceder que en algunos contextos no convenga refactorizar nada porque estoy aún en un proceso de aprendizaje del problema muy temprano en el que conviene seguir avanzando antes de crear abstracciones.

Como no tenemos nada para refactorizar, podemos empezar con el ciclo nuevamente: escribir el próximo test. La gran pregunta, que suele aparecer en este momento, es ¿por dónde sigo? ¿qué test me conviene escribir? Lo interesante de la implementación que hicimos es que "hardcodeamos" el retorno de *true* en el método `#isDead`: Por lo tanto, esto me da la pauta del próximo test: uno que me permita sacar ese *true* hardcodeado, en otras palabras, un test que cuando le pregunte al `gameOfLife` si una celda está muerta me devuelva *false*. Claramente ese test verificará que se cumpla la segunda regla del juego para el caso de dos vecinos vivos (el de tres vecinos vivos lo veremos en otro test).

```
testAliveCellWithTwoAliveNeighborsSurvives
```

```
gameOfLife := GameOfLife withAliveCells: {1@1. 1@2. 2@1}.

gameOfLife calculateNextGeneration.

self deny: (gameOfLife isDead: 1@1).
```

Podemos ver en este test cómo cambia el *setup* del juego: la celda 1@1 tiene dos celdas vecinas vivas, la 1@2 y la 2@1. Nótese que la aserción es que sea falso (`#deny:`) que la celda 1@1 esté muerta, o sea, que esté viva.

Si corremos el test, fallará. Por lo tanto, estamos haciendo TDD correctamente. Ahora, en el paso 2, tenemos que escribir lo mínimo necesario para que el test pase. Una solución que a veces me proponen es implementar `#isDead`: de tal manera que devuelva *false*. Sin embargo, no es una buena idea porque si hicieramos eso el primer test no pasaría. En este caso vamos a tener que hacer algo un poco más interesante de lo que hicimos en el paso 2 anterior. Vamos a tener que crear una nueva

colección de celdas vivas a partir de las que estén vivas y deban sobrevivir. La implementación quedaría así:

```
GameOfLife>>initializeWithAliveCells: aCollectionOfCells
    aliveCells := aCollectionOfCells
GameOfLife>>calculateNextGeneration
    aliveCells := aliveCells select: [ :aCell |
        (aCell eightNeighbors count: [ :aNeighbor | (self isDead: aNeighbor)
            not ]) = 2 ]
GameOfLife>>isDead: aCell
    ^(aliveCells includes: aCell) not
```

Si corremos los tests, veremos que pasan. Ahora hay mucho para hacer en el paso 3, ¿no les parece? Seguramente muchos entenderán el código que acabo de escribir para este paso, pero seguramente también con un poco de dificultad. La solución no es lo suficientemente declarativa. Debemos pensar bastante para entender QUÉ hace, ya que el CÓMO lo hace impera.

Primero veamos la descripción en lenguaje natural de qué hace: selecciona (#select:) las celdas vivas que al contar (#count:) sus vecinas (aCell eightNeighbors) que no están muertas ((self isDead: aNeighbor) not) sea igual a 2 (= 2). No sé qué les pasa a ustedes cuando ven código así. A mí me duele la cabeza porque me hace gastar energía tener que entender qué pasa. Analicemos qué se puede mejorar:

La doble negación no es una "buena práctica". Nunca decimos: "no verifiques que no está muerto", decimos "verifica que está vivo".

En lenguaje natural explicaríamos la regla diciendo que deseamos quedarnos con las celdas que tienen dos vecinas vivas. Entonces, ¿por qué no "decimos" lo mismo en el código.

Hagamos el primer refactor, tenemos que cambiar el test de la siguiente manera:

```
testAliveCellWithTwoAliveNeighborsSurvives
```

```
gameOfLife := GameOfLife withAliveCells: {1@1. 1@2. 2@1}.
gameOfLife calculateNextGeneration.
self assert: (gameOfLife isAlive: 1@1).
```

Para hacer pasar el test, hay que implementar `#isAlive:`. Lo hacemos de la siguiente manera:

```
GameOfLife>>isAlive: aCell
  ^aliveCells includes: aCell
```

Si corremos los tests, todos seguirán pasando. Sin embargo, ahora tenemos código repetido: tanto `#isDead:` como `#isAlive` envían el mensaje `#includes: a aliveCells`, por lo que vamos a refactorizar nuevamente para implementar `#isDead:` en base a `#isAlive:` ya que son mutuamente excluyentes:

```
GameOfLife>>isDead: aCell
  ^(self isAlive: aCell) not
```

Corremos todos los tests y verificamos que siguen corriendo satisfactoriamente. Ahora podemos hacer el otro refactor, un *extract method* que permita darle semántica a la condición del `#select:`

```
GameOfLife>>calculateNextGeneration
  aliveCells := aliveCells select: [ :aCell | (self
    numberOfAliveNeighborsOf: aCell) = 2 ]
GameOfLife>>numberOfAliveNeighborsOf: aCell
  ^aCell eightNeighbors count: [ :aNeighbor | (self isDead: aNeighbor)
    not ]
```

Si corremos los tests, deberían pasar todos, por lo que finalmente podemos cambiar la condición del `#count:`

```
GameOfLife>>numberOfAliveNeighborsOf: aCell
```

```
^aCell eightNeighbors count: [ :aNeighbor | self isAlive: aNeighbor ]
```

Hay un refactor más que podemos hacer. A no muchos se les ocurrirá hacerlo ahora, por lo que es cuestionable. Lo haremos, de todos modos, porque refleja que estamos programando el QUÉ y no el CÓMO.

```
GameOfLife>>calculateNextGeneration
```

```
aliveCells := aliveCells select: [ :aCell | self shouldSurvive: aCell ]
```

```
GameOfLife>>shouldSurvive: aCell
```

```
^(self numberOfAliveNeighborsOf: aCell) = 2
```

Fíjense que encapsulamos en el mensaje `#shouldSurvive:` el hecho de que sean 2 las vecinas vivas y, al hacerlo, nos despreocupamos de ese detalle (del **cómo**) haciendo explícito el **qué**. Las nuevas celdas vivas son aquellas que deben sobrevivir. Como dice el dicho: “más claro echale agua”.

Es importante recalcar que si no hubiésemos hecho estos *refactors* no estaríamos haciendo bien TDD. Por otro lado, debemos aceptarlo: no a cualquiera se le ocurrirían. Esta es justamente la diferencia entre un “diseñador” y un “buen diseñador”. Un buen diseñador reconoce cuando el código no es lo suficientemente explícito. Sabe que usar doble negación no es bueno. Sabe que es bueno encapsular los ciclos para hacer explícita la semántica de los mismos (en este caso el `#select:`). Además, no tiene miedo del impacto en la performance porque sabe que en la actualidad programamos para las personas, no para las máquinas como sucedía en las décadas del 60, 70, 80 y podríamos decir hasta del 90, cuando hacer un envío de mensaje adicional o una “llamada a una función más” implicaba un costo en performance. En la actualidad, con la arquitectura de *pipeline* de los microprocesadores actuales, los problemas de performance pasan por otro lado. Una conclusión muy importante: “TDD no implica buen diseño”. Los buenos diseños los hacen los buenos diseñadores. Ser buen diseñador no cuesta tanto: hay que priorizar declaratividad sobre performance (inicialmente) y seguir ciertas heurísticas.

Pero volvamos a nuestro problema, tenemos que hacer el paso 1 nuevamente, escribir un test simple y que falle. Aún no hemos terminado con la segunda regla del juego ya que sólo hemos testeado por la existencia de 2 vecinos vivos. A continuación, tenemos que testear por 3 vecinos vivos. Noten que mi elección de qué testear se relaciona con la regla actual, no con la siguiente. Esa es una buena heurística sobre qué test escribir: tratar siempre de finalizar con la funcionalidad que estamos testeando antes de pasar a otra. En este caso, terminar con la segunda regla del juego antes de pasar a la tercera. El test sería entonces:

```
testAliveCellWithThreeAliveNeighborsSurvives
```

```
gameOfLife := GameOfLife withAliveCells: {1@1. 1@2. 2@1. 2@2}.
```

```
gameOfLife calculateNextGeneration.
```

```
self assert: (gameOfLife isAlive: 1@1).
```

Si corremos este test, veremos que falla. En consecuencia, estamos haciendo bien TDD. Ahora debemos hacer la implementación más simple que pase:

```
GameOfLife>>shouldSurvive: aCell  
^(self numberOfAliveNeighborsOf: aCell) between: 2 and: 3
```

Simplemente tenemos que asegurarnos que la cantidad de vecinos vivos son 2 o 3 y el test pasa.

Luego de hacer pasar este test (o incluso a veces luego de haber escrito o hecho pasar el test anterior) varios preguntan por qué sólo verifíco que la celda 1@1 esté viva y no las otras que también deberían estar vivas, como en este caso pasa con todas sus vecinas.

La respuesta es que el hacerlo no agrega valor. Si agregan esas aserciones verán que los tests siguen funcionando lo que significa (para este ejemplo) que no son necesarias porque no están funcionalmente probando un caso distinto. Si hacemos un test de cobertura con la aserción sobre 1@1 solamente y luego agregando las aserciones sobre las otras celdas, veremos que el resultado será el mismo. Esto demuestra que no estamos verificando nada nuevo.

Hice hincapié en "para este ejemplo" porque puede suceder que hayamos olvidado aserciones, o sea que el test no esté bien escrito. Claramente en dichas situaciones, agregarlas resulta correcto.

Retomando el ejemplo, tenemos que revisar si se debe refactorizar algo (paso 3). Si no es necesario, estamos en condiciones de escribir nuestro próximo test: el que prueba la tercera regla:

```
testAliveCellWithMoreThanThreeAliveNeighborsDies
    gameOfLife := GameOfLife withAliveCells: {1@1. 1@2. 2@1. 2@2.
1@3}.
    gameOfLife calculateNextGeneration.
    self assert: (gameOfLife isDead: 2@2).
```

Ahora estamos testeando por la celda 2@2, que es la que posee 4 vecinos vivos en el juego que acabamos de configurar. Al correr este test, observamos que funciona en el primer intento. Esto indica que ya no estamos haciendo TDD, porque no cumplimos con el hecho de que falle el test recién escrito. Por lo tanto, tenemos que reflexionar qué pasó, por qué no falló.

Cuando un test recién escrito pasa sin inconvenientes, puede haberse presentado una de las siguientes situaciones:

1. El test está repetido. El mismo caso ya se testeó en otro lugar.
2. Cuando hicimos el paso 2 en una iteración anterior, no implementamos lo mínimo necesario para que pasen los tests existentes hasta ese momento. En otros términos, nos adelantamos en el alcance de la solución.
3. 1) y 2) al mismo tiempo.

¿Qué sucedió en este caso? ¿Hicimos una implementación que abarca más casos en una iteración anterior o el test está repetido? Podemos inclinarnos a pensar que está ocurriendo 2), pero si analizamos el código veremos que no hicimos nada de más realmente. Lo que sucede es que este caso ya está testeado. Esto resulta difícil de creer porque se trata de una regla distinta del juego. Por lo menos, así está presentada en la descripción del mismo. Sin embargo, si nos ponemos a analizar las reglas desde el punto de vista lógico, ¡la regla 2 incluye la 3! Por lo tanto, el

problema surge porque estamos verificando un caso ya implementado debido a que la especificación es redundante. En pocas palabras, "hay un problema" en la especificación. Aquí abro un paréntesis: seguramente nadie lo detectó al momento de leer las reglas. Esto es completamente lógico y natural, porque constituye una regla de un juego ya probado y jugado mil veces, ¡no puede estar mal! De hecho, no está mal la regla. Simplemente es redundante y por eso resulta previsible que no podamos reconocerla. Como seres humanos, hablamos y pensamos mediante un lenguaje natural, ambiguo y contextual. Por lo tanto, algunas "redundancias" no molestan. Por el contrario, entregan seguridad.

Pero nosotros estamos haciendo un modelo computable formal. En este proceso de formalización de conocimiento, que es el desarrollo de software, nos encontraremos frente a definiciones ambiguas y contextuales. Es parte de nuestro trabajo reconocerlas y actuar en consecuencia.

¿Qué hacemos con este test, entonces? Lo borramos, porque no ayuda desde el punto de vista formal. Podemos ir más allá aún, borrar también la tercera regla del juego. Seguimos ahora por el paso 1: tenemos que pensar en otro test, el que verifique la regla número cuatro:

```
testDeadCellWithThreeAliveNeighborsBecomesAlive
```

```
gameOfLife := GameOfLife withAliveCells: {1@1. 1@2. 2@1}.
gameOfLife calculateNextGeneration.
self assert: (gameOfLife isAlive: 2@2).
```

Nótese que seguimos verificando el estado de la celda 2@2, pero que la misma no está dentro de las celdas vivas por lo que no queda otra opción: está muerta. Si corremos el test, fallará. Esto indica que volvimos a hacer TDD. Ahora debemos hacer la implementación que haga que este test pase. ¿Cómo tenemos que modificar #calculateNextGeneration? Buscando los vecinos vivos de una celda muerta. Pero enfrentamos un problema: ¡no sabemos cuáles son las celdas muertas! Sólo estamos guardando las celdas vivas. Esto nos trae un dilema a nivel diseño: ¿cómo informarle al juego qué celdas están muertas?

Hay varias soluciones:

1. Cuando creamos el juego, no sólo pasarle las celdas vivas, sino también las muertas. Esto implica cambiar bastante el modo de implementación, ya que deberíamos usar una matriz donde cada elemento represente si la celda está viva o muerta (¿con un boolean, quizás?). También podríamos usar un diccionario cuya clave sea el punto y el valor si está viva. Esto parece, en primera instancia, bastante malo desde el punto de vista implementativo.
2. Pasarle al juego una colección adicional, la de las celdas muertas. Esto parecería no ser tan disruptivo respecto del diseño que venimos haciendo. Tendríamos que asegurarnos que la unión de ambas colecciones contenga todas las celdas del juego y que no existan celdas vivas y muertas al mismo tiempo.
3. Pasarle al juego, además de la colección de celdas vivas, la dimensión del tablero. Esto nos permitiría iterar sobre todas las celdas. Aquellas que no se hallan dentro de la colección de celdas vivas están muertas. Esta opción parece ser la más simple: no tenemos que verificar que no haya errores de construcción como en 2) y tampoco modificar tanto el diseño respecto de lo implementado, manteniendo la simplicidad de tener que configurar solamente las celdas vivas ya que las otras, por omisión, están muertas.
4. Posiblemente otras opciones que no se me ocurren.

Utilizaremos la opción 3). Tenemos que agregar un parámetro al mensaje de creación de instancia (constructor en algunos lenguajes) sin romper los tests. Como lo que vamos a hacer es un *refactor*, los tests deben estar pasando. Recuerden que sólo es posible hacer el paso 3) (refactorizar) luego de hacer el paso 2) del cual se sale solamente si no falla ningún test. Por lo tanto, vamos a sacar el último test que hicimos, renombrándolo de `#testDeadCellWithThreeAliveNeighborsBecomesAlive` a `#no_testDeadCellWithThreeAliveNeighborsBecomesAlive`. Luego volvemos a correr los tests para asegurarnos que refactorizar es posible.

Una vez que estamos seguros, agregamos un nuevo parámetro al mensaje `#withAliveCells:`. Lo podemos hacer a mano o, si el IDE que usamos tiene *refactors* automatizados, podemos hacer un “*add parameter*” o un “*change signature*” dependiendo del lenguaje/IDE utilizado. Más allá del caso, lo que vamos a indicarle es el uso del objeto `3@3` como valor inicial

para este nuevo parámetro. En el caso de un lenguaje con *keywords* explícitos, como *Smalltalk*, nombraremos dicho *keyword* como `#ofSize:`.

Esto implicaría que todos los métodos que envían el mensaje `#withAliveCells:` deberían quedar así:

```
...
GameOfLife withAliveCells: { ... } ofSize: 3@3.
...
```

Y la implementación del mensaje quedaría así:

```
GameOfLife class>>withAliveCells: aCollectionOfCells ofSize:
aBoardSize
^self new initializeWithAliveCells: aCollectionOfCells
```

Ahora tenemos que agregarle el parámetro a `#initializeWithAliveCells:` de la misma manera:

```
GameOfLife class>>withAliveCells: aCollectionOfCells ofSize:
aBoardSize
^self new initializeWithAliveCells: aCollectionOfCells ofSize:
aBoardSize
GameOfLife>>initializeWithAliveCells: aCollectionOfCells ofSize:
aBoardSize
    aliveCells := aCollectionOfCells
```

Si corremos los tests, deberían seguir funcionando porque acabamos de hacer un refactor automatizado. Terminados los pasos 3) y 1), continuamos con el 2). Para que el test pase, debemos quedarnos con la dimensión del tablero:

```
GameOfLife>>initializeWithAliveCells: aCollectionOfCells ofSize:
aBoardSize
    aliveCells := aCollectionOfCells.
    boardSize := aBoardSize
```

Y modificar `#calculateNextGeneration` para que itere sobre las celdas del mismo. Lamentablemente, no es tan sencillo puesto que hay que implementar la iteración a mano:

```
GameOfLife>>calculateNextGeneration
| newAliveCells |
newAliveCells := OrderedCollection new.
1 to: boardSize x do: [ :x |
  1 to: boardSize y do: [ :y | | cell |
    cell := x@y.
    ((self shouldSurvive: cell) or: [ self shouldBecomeAlive: cell ])
      ifTrue: [ newAliveCells add: cell ]].
  aliveCells := newAliveCells.
GameOfLife>>shouldBecomeAlive: aCell
  ^(self isDead: aCell) and: [ (self numberOfAliveNeighborsOf: aCell) =
  3 ]
```

Con estos cambios, los tests pasan. Si volvemos a habilitar el test `#testDeadCellWithThreeAliveNeighborsBecomesAlive`, veremos que también pasa. Esto significa que ya hicimos el paso 2) para esta nueva funcionalidad y estamos en condiciones de hacer el paso 3) para hacer más declarativa la implementación de `#calculateNextGeneration`.

La iteración sobre todos los puntos del tablero genera ruido al QUÉ. Encapsularemos la iteración en un mensaje que recibirá un *closure*, que será evaluado para cada punto:

```
calculateNextGeneration
| newAliveCells |
newAliveCells := OrderedCollection new.
self boardCellsDo: [ :aCell |
((self shouldSurvive: aCell) or: [ self shouldBecomeAlive: aCell ])
ifTrue: [ newAliveCells add: aCell ]].
aliveCells := newAliveCells.
boardCellsDo: aClosure
1 to: boardSize x do: [ :x |
1 to: boardSize y do: [ :y | aClosure value: x@y ]]
```

Si corremos los tests, deberían pasar. Ahora lo dejaremos como estaba antes, con un #select: fácil de leer:

```
calculateNextGeneration
aliveCells := self boardCellsSelect: [ :aCell |
(self shouldSurvive: aCell) or: [ self shouldBecomeAlive: aCell ]].
boardCellsSelect: aCondition
| selectedCells |
selectedCells := OrderedCollection new.
self boardCellsDo: [ :aCell | (aCondition value: aCell)
ifTrue: [ selectedCells add: aCell ]].
^selectedCells
```

¡Ahora, sí! Resulta claro. Nótese que la implementación no genera colecciones adicionales. No se crea la colección de celdas del tablero y luego se hace el #select:, sino que se itera sobre las celdas, quedándose sólo con aquellas que cumplen con la condición del #select:.

Volvamos a la funcionalidad. En el último cambio funcional realizado, se introdujo un error que puede pasar desapercibido si no hacemos un buen análisis de problema. La regla 4) dice que celdas muertas con 3 vecinos vivos reviven, por lo que hay que asegurarse que celdas muertas con otra

cantidad de vecinos no sobrevivan. Nosotros sólo testeamos "el caso feliz" en el último test, pero no el caso "negativo". ¿Cómo escribimos el test para celdas muertas con vecinos vivos distinto a 3? El problema es que puede haber muchos vecinos vivos, por lo menos de 0 a 2 y de 4 a 8, ya que pueden haber de 0 a 8 vecinos vivos. Podríamos hacer un test por cada una de las posibilidades (9 tests) o un test mediante el que iterar las distintas configuraciones. Más allá de eso estaríamos probando el mismo caso funcional y por lo tanto es medio molesto tener que hacerlo para cada "dato de prueba" (configuración). De esta situación se desprenden varios puntos a tener en cuenta:

1. Diferencia entre "dato de prueba" y "caso de prueba": Los datos de prueba son aquellos que se utilizan para probar un caso de prueba. Un caso de prueba puede estar compuesto por un número finito o infinito de datos de prueba.
2. Debido a que puede haber infinitos datos de prueba es necesario buscar una manera de testear un "caso" de la manera más completa posible con un conjunto finito de datos de prueba. Ese conjunto finito se obtiene por medio de técnicas donde se buscan "datos testigos", que identifican situaciones límites o representan un conjunto de datos de prueba.
3. Debido a 2) es que a) Dijkstra "odiaba el testing" argumentaba que no representaba una técnica formal de verificación. O sea, el testing solo te asegura que funciona o no lo que se está testeando. Nada dice sobre lo no testeado. b) Personalmente, opino que debemos hacer "inducción incorrecta", tomando la técnica de inducción de Álgebra.

¿Cómo se hace inducción en álgebra? Se toma una tesis y se demuestra que funciona para 1, que también funciona para N y entonces, por inducción, se demuestra que funciona para N+1 (o sea, para el resto). ¿Qué implica hacer inducción incorrecta? Se demuestra que funciona para 1 y se dice "funciona para N" lo que constituye un error lógico pero necesario al realizar testing. Eso es lo que vamos a hacer con este caso. Tomaremos casos testigos y probaremos con ellos que celdas muertas reviven solo si tienen 3 vecinos vivos. Esos casos testigos serán el 2 y el 4. En realidad, podría ser sólo el 2, ya que en otras reglas se compara cantidad de vecinos vivos con 2, pero vamos a tomar el 4 también y

asegurar que fuera del 3 (por lo menos para el anterior y posterior) el juego se comporta correctamente.

Escribamos el test:

```
testDeadCellAliveNeighborsDifferentToThreeKeepsDead
```

```
    gameOfLife := GameOfLife withAliveCells: {1@2. 2@2. 3@2. 1@3}
ofSize: 3@3.
    gameOfLife calculateNextGeneration.
    self assert: (gameOfLife isDead: 1@1).
    self assert: (gameOfLife isDead: 2@3).
```

Este test falla para el caso de 2 vecinos vivos ya que habíamos escrito `#shouldSurvive:` cuando sólo recorriámos celdas vivas. Ahora que recorremos todo el tablero, no estamos seguros que cuando comparamos por 2 o 3 vecinos vivos sea solo para celdas vivas. Debemos modificar `#shouldSurvive:` de la siguiente manera:

```
GameOfLife>>shouldSurvive: aCell
  ^(self isAlive: aCell)
  and: [ (self numberOfAliveNeighborsOf: aCell) between: 2 and: 3 ]
```

Ahora sí: los tests pasan todos. A simple vista no podemos mejorar el diseño, pero si nos fijamos en la implementación existen un par de mejoras que podemos hacer. La primera es no tener que buscar los vecinos vivos más de una vez, algo que actualmente sucede porque se buscan en `#shouldSurvive:` y `#shouldBecomeAlive::`. La segunda mejora es un poco más sutil y se relaciona con el hecho de que no importa si una celda está viva o muerta. Si tiene tres vecinos vivos, debe estar viva en la próxima generación. Nuevamente vemos una opción de mejora en la definición del juego por haber formalizado el mismo.

Para realizar este refactor habría que hacer un "*inline*" de los métodos `#shouldSurvive:` y `#shouldBecomeAlive::`. Hacer un "*inline method*" es lo opuesto a un "*extract method*": reemplaza todos los *senders* del mensaje

relacionado a ese método por las colaboraciones del mismo. El método `#calculateNextGeneration` quedaría así:

```
GameOfLife>>calculateNextGeneration
aliveCells := self boardCellsSelect: [ :aCell |
  ((self isAlive: aCell) and: [ (self numberOfAliveNeighborsOf: aCell) between: 2 and: 3 ]) or: [ (self isDead: aCell) and: [ (self numberOfAliveNeighborsOf: aCell) = 3 ]]].
```

El código volvió a ser ilegible, pero puede observarse la búsqueda de los vecinos vivos 2 veces. Acá podría argumentarse, y con razón, que haber hecho los *refactors* en `#shouldSurvive`: y `#shouldBecomeAlive`: no fue una buena idea. Podríamos haber pasado por alto esta situación. Es por ello que a veces conviene esperar, quizás hasta finalizada la solución, para empezar a refactorizar y crear abstracciones. Por otro lado, no hacerlo puede volver muy difícil la comprensión del código que estamos escribiendo. Este es un *trade off* que debemos hacer continuamente. Es completamente contextual.

Volviendo al refactor, primero calcularemos los vecinos vivos y los guardaremos en una variable. Para ello, usaremos un refactor llamado “*extract to variable*”:

```
GameOfLife>>calculateNextGeneration
aliveCells := self boardCellsSelect: [ :aCell |
  numberOfAliveNeighbors | 
  numberOfAliveNeighbors := self numberOfAliveNeighborsOf: aCell.
  ((self isAlive: aCell) and: [ numberOfAliveNeighbors between: 2 and: 3 ]) or: [ (self isDead: aCell) and: [ numberOfAliveNeighbors = 3 ]]].
```

Si corremos los tests, deberían funcionar todos. Ahora podemos simplificar la comparación de cantidad de vecinos vivos a 3, no importa si está viva o muerta.

```
GameOfLife>>calculateNextGeneration
  aliveCells := self boardCellsSelect: [ :aCell | |
    numberOfAliveNeighbors | |
      numberOfAliveNeighbors := self numberOfAliveNeighborsOf:
        aCell.
      numberOfAliveNeighbors = 3
      or: [ (self isAlive: aCell) and: [ numberOfAliveNeighbors = 2 ]]].
```

Si corremos los tests, deberían funcionar todos. Daremos nuevamente semántica a la condición del #select:

```
GameOfLife>>calculateNextGeneration
  aliveCells := self boardCellsSelect: [ :aCell | |
    self shouldBeAliveOnNextGeneration: aCell ]
  shouldBeAliveOnNextGeneration: aCell
  | numberOfAliveNeighbors |
  numberOfAliveNeighbors := self numberOfAliveNeighborsOf: aCell.
  ^numberOfAliveNeighbors = 3
  or: [ (self isAlive: aCell) and: [ numberOfAliveNeighbors = 2 ]]
```

Daremos semántica a la última condición:

```

shouldBeAliveOnNextGeneration: aCell
| numberOfAliveNeighbors |
numberOfAliveNeighours := self numberOfAliveNeighborsOf: aCell.
^numberOfAliveNeighours = 3
or: [ self shouldSurvive: aCell with: numberOfAliveNeighbors ]

shouldSurvive: aCell with: aNumberOfAliveNeighbors
^(self isAlive: aCell) and: [ aNumberOfAliveNeighbors = 2 ]

```

Finalmente tenemos la solución completa, con un diseño claro y con la seguridad, gracias a los tests, de que funciona correctamente. Nótese, además, que no tuvimos grandes problemas de diseño o implementación. El desarrollo fue "suave" debido justamente al modo iterativo e incremental utilizado. Espero que hayan podido observar y sentir estas características y que hayan seguido el ejemplo en sus máquinas programando de forma simultánea a la lectura del capítulo. Si no lo hicieron, ¡no esperen más! ¡Agarren una computadora con su lenguaje de programación preferido e implementen este juego!

Queda mucho por hablar de TDD, mucho por decir, ya no sobre qué es o cómo se lo practica sino preguntas de carácter organizativo e implementativo en un grupo de trabajo. Trataré de cubrir algunas a continuación:

### **1. ¿Se puede hacer TDD en todo el sistema?**

No, no se puede. TDD aplica cuando se está "desarrollando software" (por desarrollar me refiero a "crear"), cuando se crean nuevas abstracciones o implementan nuevos algoritmos. No aplica cuando se está "configurando" software. ¿A qué me refiero con ésto? A instanciar *frameworks* básicamente. Por ejemplo: cuando configuramos un *ORM* (*Hibernate*, etc) para mapear objetos a base de datos relacionales, configuramos un *framework* visual para crear ventanas, un *framework* para traducir un formato de objeto a otro, etc. Básicamente todo lo que implique configuración o únicamente reutilización de una solución existente no se puede desarrollar con TDD. Para esos casos hay que hacer testing o ser testeado indirectamente por tests que se hagan por medio de TDD cuando se desarrolle software que lo utilice.

## 2. ¿Qué cobertura de código debemos tener cuando se hace TDD?

Si aplicamos TDD correctamente, la cobertura de lo que se está desarrollando debería ser del 100%, pero subrayo "correctamente", lo cual es muy difícil de lograr y más aún cuando recién estamos empezando a practicarlo. Hago esta aclaración porque es muy común encontrarse con los "talibanes de siempre", con los dogmáticos que ocupan posiciones de decisión o poder y que luego de leer que con TDD se puede lograr 100% de cobertura, lo exijan a los equipos de desarrollo apenas empiezan. Esto es una locura total y termina siendo contraproducente porque lo único que genera es odio de los programadores hacia la técnica que les está "complicando la vida". Terminan escribiendo cualquier tipo de test sólo para tener cobertura. Por otro lado, recordemos que la misma sólo debe ser la de los objetos programados, no la de los utilizados. Ese límite es muy difícil de trazar si la herramienta de *coverage* no es buena. Por último, recuerden que un sistema con 100% de cobertura resulta muy "rígido" lo que puede jugar en contra del mantenimiento y la evolución del mismo.

## 3. Si hacemos TDD, ¿se necesita tener un equipo de QA? ¡POR SUPUESTO!

Hacer TDD no reemplaza QA ya que hay ciertas partes del sistema que no se pueden testear haciendo TDD (ver pregunta 1), porque hay ciertos tests que deben ser automatizados por herramientas de alto nivel funcional y siempre es necesario que al menos un ser humano use el sistema. Lo que tiene de interesante hacer TDD es que el equipo de QA recibirá un sistema con menos errores, lo que permitirá concentrarse en hacer tests de alto nivel funcional en vez de probar temas simples.

## 4. ¿Cómo hago TDD del *user interface*? ¡NO SE PUEDE!

O sí, dependiendo de la UI que estemos haciendo. Los tests son muy frágiles. Veamos dos casos: 1) Si estoy haciendo una UI ventana entonces la construcción de la UI se realizará por medio de instanciar un *framework* y, como ya vimos en la pregunta 1, no tiene sentido hacer TDD cuando se instancian *frameworks*. Por otro lado, es muy difícil controlar la interacción con la UI de manera automatizada, principalmente porque la misma corre en su propio *thread*, generando problemas de sincronismo entre el test y la UI y finalmente porque cambios en lo que se muestra y cómo se muestra pueden hacer fallar los tests. 2) Si estamos haciendo una UI web, se mantiene el mismo problema de la fragilidad. Cualquier cambio en el

HTML generado puede hacer que el test falle (más allá de las complicaciones de escribir aserciones sobre HTML). Conclusión: Si queremos automatizar la verificación de la UI, hay que hacer testing y usar herramientas especializadas para dicho fin.

**5. ¿Cuánto tiempo se necesita para dominar esta técnica?** Es difícil de responder. Depende del conocimiento de la persona que está aprendiendo, de la experiencia en desarrollo de software, etc. Algo que he notado es que cuanto más senior es un programador más tiempo le cuesta aprender TDD porque debe hacer un cambio muy fuerte en su manera de pensar y en años de experiencia de programar de otro modo. Sin embargo, escriben mejores tests justamente por los años de experiencia y los golpes recibidos. Los programadores con menos experiencia que aceptan la técnica como forma de programar aprenden más rápido, pero les cuesta más escribir buenos tests o todos los tests necesarios debido a no haber "sufrido" tanto lo errores del desarrollo de software.

**6. ¿Se puede hacer TDD en un sistema ya desarrollado?** No, no se puede hacer TDD en un sistema ya existente a menos que lo que se quiera desarrollar no tenga ninguna relación con lo hecho, en definitiva, algo completamente nuevo. ¿Por qué? Porque dicho sistema tiene, a nivel diseño, una entropía muy fuerte y, por lo tanto, mucho acoplamiento y poca cohesión. Además, seguramente fue desarrollado con malas técnicas de diseño (clases anémicas y servicios que representan más a un paradigma estructurado que de objetos, etc.) o está acoplado a la base de datos. Esto atenta contra la posibilidad de hacer TDD y, más aún, contra la posibilidad de hacer testing automatizado. Si el sistema tiene mucho acoplamiento, será difícil testear situaciones que impliquen usar datos de prueba por fuera de los dados por ese acoplamiento. Por ejemplo, si el día se obtiene en cualquier parte del sistema haciendo algo parecido a "Date today" o "new Date()", entonces no podremos correr tests en fechas que no sean del día de hoy. En pocas palabras, no podremos simular qué día es hoy. Si el sistema está acoplado con la base de datos, entonces lamentablemente los tests serán lentos. El problema con los tests lentos es que dejamos de correrlos. Todo esto atentará contra escribir tests. La única manera de resolver este problema es cambiar el diseño del sistema de a poco, por medio de refactors automatizados que garanticen que los cambios realizados no rompan la ejecución del

sistema, llevando el diseño a una situación que permita empezar a escribir tests. En dicha situación, se hará testing, no TDD: o sea se escribirán tests de código ya escrito. De a poco, a medida que se escriban más tests, el diseño irá mejorando. Esto permitirá que, llegado cierto momento, se empiece a utilizar TDD. Como se puede observar, este es un proceso largo y meticuloso que generalmente sobrevive al tiempo de rotación de las personas en un proyecto. Por eso es tan difícil de hacer exitosamente.

**7. ¿Cuándo debo usar *mock objects*? (objetos simuladores)** Los objetos simuladores, *Test Doubles*, también lamentablemente llamados mocks, son objetos que simulan ser algo cuando en realidad son otra cosa. En definitiva, son objetos polimórficos con el objeto simulado que se usan para poder testear situaciones difíciles de reproducir con los objetos reales o donde el test no está en control. Ejemplos concretos a simular son sistemas externos que, por su carácter, los tests no controlan, volviendo más lenta su ejecución. El tópico de objetos simuladores es un tema en sí mismo. Tratarlo llevaría casi la misma dimensión que este capítulo, por lo que no me extenderé mucho más. Sólo les diré una regla de oro: Simular sólo lo que no desarrollo, nunca simular objetos que forman parte de lo que estoy desarrollando.

**8. ¿Qué hago si mis tests tardan mucho en ejecutarse?** Debo buscar el modo de reducir el tiempo de ejecución. Los tests que deben ejecutarse cuando hago un cambio o implemento algo con TDD no deberían tardar en correr más de 2, 3 o 5 segundos, como mucho 20 o, a lo sumo, 1 minuto. Cuando pasamos de segundos a minutos en el tiempo de ejecución de los tests estamos en problemas. Aclaro que cuando digo los "tests que deben ejecutarse" no me refiero a los tests de todo el sistema, sino a los de la funcionalidad que estoy cambiando o agregando. Un sistema grande seguramente esté compuesto por módulos o subsistemas. Por lo tanto, al modificar un subsistema, sólo debo correr los tests correspondientes al mismo, que deben ejecutarse en segundos. Luego de hacer la modificación y ver que todo funcione bien a ese nivel, deben correrse los tests del sistema. En dicho caso, puedo pasar a hablar de minutos, aunque no demasiados. Cuantos menos, mejor. Pasar los diez minutos para correr todos los tests del sistema sería un problema. Tener tests que satisfagan estos tiempos de ejecución no es fácil, pero tampoco imposible. Se logra por medio de un buen diseño y uso de objetos simuladores. ¿Qué puede hacer que los tests tarden mucho? Un motivo

puede ser que el sistema esté acoplado a la base de datos. Primero hay que simular la base de datos con objetos *fake* como base de datos en memoria. Aún así, seguramente tardarán mucho en correr, por lo que hay que tener como objetivo desacoplar completamente de la base de datos. Este objetivo, en algunos contextos, será imposible de lograr. Por ejemplo, si utilizamos *Ruby on Rails*, debido a que se trata de un *framework* de caja blanca en lo que respecta a persistencia (todo lo que se persiste debe subclasicar *ActiveRecord*), resulta imposible desacoplar la base de datos. La única forma de hacer que los tests corran rápido en este contexto consiste en sacar *Rails* del sistema, pero eso es imposible si el sistema ya está desarrollado, a menos que se haga uno completamente nuevo. Lo que podemos hacer, si nos interesa testear nuestro código, aplicar TDD y tener un sistema mantible es no escribir aplicaciones web con *Ruby on Rails* y usar otro *framework* que esté preparado para hacer TDD. Otro motivo por el cual los tests pueden tardar mucho es que se encuentren repetidos, mal escritos o que el sistema esté mal desarrollado. En la mayoría de los casos, este problema se debe al uso de recursos externos (archivos, microservicios, base de datos, etc.).

### *Conclusiones*

TDD es una técnica de desarrollo basada en un proceso iterativo, incremental y con feedback inmediato. Permite resolver problemas complejos dando pequeños pasos y permite, al ser un proceso predefinido, reflexionar cómo estamos programando. Cuando hacemos TDD, el tiempo es uno de los factores cruciales a medir. El tiempo que tardamos en escribir un test, el que tardamos en hacer pasar un test, el que invertimos refactorizando y el de ejecución de los tests. Esta medida me permite saber si estoy aplicando correctamente TDD.

No se puede hacer TDD en todo el sistema, no reemplaza QA y no puede hacerse en un sistema existente, por lo menos, hasta no tener un diseño que permita testearlo.

Hay mucho más para hablar sobre TDD, como la estructura que deben tener los tests, cómo se los debe nombrar, cómo se los puede organizar, qué datos de prueba usar, etc., pero hacerlo excedería la intención de esta sección. Espero, sin embargo, que el mismo les haya servido para entender la técnica. Para terminar, volveré al principio: practiquen todo lo

que leyeron acá y luego, ¡vuelvan a practicarlo! Es la única manera de asegurar el aprendizaje y la correcta ejecución de la técnica.

## *Refactor Continuo*

Escribir código es similar a escribir un libro. Me doy cuenta ahora que estoy aprendiendo esto último. No escribo bien y como deseo “en una pasada”. Escribo una primera versión: leo, pienso, pido feedback, corrojo y refino (o re-escribo si no me gusta). Cuando escribo código, a pesar de haberlo hecho durante muchos más años, me pasa lo mismo. Primero entiendo el problema, bosquejo la posible solución y escribo una primera versión de ella. Luego leo el código y pienso cómo podría mejorarlo. Siempre se me ocurren formas de hacerlo. Voy “refactorizando” (ejecutando los tests para estar seguro de que todo sigue funcionando) hasta obtener un resultado satisfactorio. Este proceso de mejora del código no termina cuando completo la *User Story*. Continúa infinitamente mientras trabajo en un proyecto. Es una búsqueda constante del mejor diseño, de lograr la mejor versión del código que funcione.

Esta práctica, descrita en el párrafo anterior, se llama *Continuous Refactoring* y también la popularizó Beck, a partir de *eXtreme Programming*. Martin Fowler, autor del famoso libro “*Refactoring: improving the design of existing code*” (Fowler, 199), que recientemente sacó a la luz su 2<sup>da</sup> edición, define un “refactor” como “un cambio en la estructura interna del software que lo hace más fácil de entender y más barato de evolucionar, sin que se modifique el comportamiento observable”.

En 10Pines, tenemos casi una obsesión por refactorizar. Lo hacemos todo el tiempo: si no nos gusta el nombre elegido para una variable, si notamos que cierta área del código se volvió demasiado compleja y, por supuesto, si detectamos duplicación en el código. Podemos hacerlo, de forma rápida y confiada, porque existen tests que nos brindan feedback instantáneo y nos alertan ante una falla. Como ya mencioné, estas prácticas funcionan mejor en combinación.

Refactorizar continuamente nos permite mejorar “el diseño del código existente” de modo de mantenernos capaces de incorporar nueva funcionalidad sustentablemente. En otras palabras, nos permite mantener

el costo de cambio bajo. Más allá de que nos guste el código elegante, la razón principal es económica.

## *Code Review*

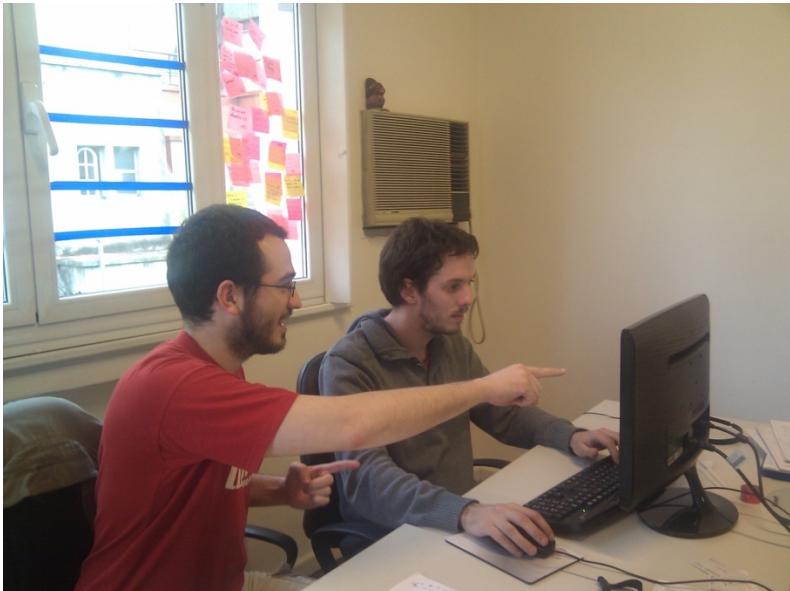
Antes de incorporar nuestros cambios en el *branch* de desarrollo, es bueno que otro par de ojos le den una mirada. Esta revisión, que llamamos *code review*, brinda observaciones que permiten pulir y mejorar el código antes de “*mergearlo*”. Por ejemplo, nos alerta de un nombre que no se entiende, de cambios que pueden afectar algún área que no tuvimos en cuenta o de algún problema de performance.

El proceso que se ha estandarizado para hacer esta *review*, a partir de la popularización de Git y Github, es la creación de un *pull request*, que se revisa y *mergea* cuando se aprueba. Debemos realizar el *code review* tan pronto como sea posible, de manera de minimizar el trabajo en progreso.

Esta revisión podría ser “*online*”, es decir, el revisor mira el código junto con las personas que lo desarrollaron o asincrónica, dejando comentarios y sugerencias. El primer caso es más eficiente porque, mientras se revisa el código, pueden hacerse preguntas y debatir las decisiones tomadas. Sin embargo, esto implica que los involucrados deban coordinar un horario, con la consecuente incomodidad para el revisor que debe interrumpir sus tareas. Las revisiones asíncronas también funcionan correctamente. Incluso, se pueden categorizar los comentarios u observaciones: algunos serán bloqueantes y otros, simples sugerencias.

Esta práctica funciona bien en ambientes sanos y colaborativos donde cualquiera puede hablar con seguridad y funciona mal en ambientes con poca colaboración donde la gente se vuelve muy cautelosa. La confianza es un requisito para este tipo de prácticas.

## *Pair Programming*



Agustín y Dario, haciendo *pair-programming* en las 1<sup>ras</sup> oficinas de 10Pines.  
Foto extraída de Wikipedia

Si el *code review* es bueno, llevémoslo al extremo, dice Beck en *eXtreme Programming* al explicar la práctica de *pair programming*. Escribamos el código de a dos, tomando las decisiones juntos, debatiendo ideas y revisando el código escrito.

Usamos mucho esta práctica para desarrollar *User Stories* difíciles y para transmitir conocimiento. La calidad del código producido es siempre mayor a la que se puede alcanzar trabajando individualmente. Las decisiones se debaten. Las personas combinan sus esfuerzos en procura de un código legible. Suman sus ideas: se debaten, se refinan y se mejoran. Además, las probabilidades de introducir un *bug* se ven reducidas. Nos sentimos bien trabajando de esta manera. Nos divertimos y nos levanta la moral. También nos ayuda a mantener la concentración: no chequeamos el *whatsapp* cuando estamos haciendo *pair programming* con

un compañero. Algunos equipos establecen sesiones con una duración de, por ejemplo, una hora, permitiendo pausas necesarias para realizar tareas sociales.

El año pasado usamos esta práctica en una transformación digital de un cliente de un modo muy efectivo. En una retrospectiva surgió la idea de desarrollar las *stories* haciendo *pair-programming* con un programador del cliente y otro de 10Pines. La combinación de los conocimientos del dominio con los conocimientos prácticos hizo que la productividad y la calidad mejoraran. Además, repercutió positivamente en las relaciones entre los miembros del equipo.

Mucho tiempo atrás, se cuestionó la pérdida de productividad que esta práctica podría provocar. Sin embargo, estas hipótesis se rebatieron oportunamente. En la actualidad, nadie cuestiona la productividad que muchos consideran indispensable.

Al igual que con la práctica de *code review*, pueden surgir problemas en ambientes poco colaborativos. También con programadores acostumbrados a trabajar individualmente durante toda su vida. Como antes afirmamos, la confianza también es un prerequisito para obtener los mejores resultados a partir de esta práctica.

## *Integración Continua*

Cuando trabajamos en una *User Story*, creamos un *branch* para poder trabajar enfocados en nuestra funcionalidad. Al completarla, debemos integrar los cambios que se hicieron en el *branch* principal para crear una versión unificada. Este proceso puede resultar complejo, dependiendo del tiempo que nos haya tomado. Cuanto mayor sea, más complejo será, pues habrá más cambios en el *branch* principal.

Para decrementar esta dificultad y, sobre todo, el riesgo que conlleva la integración, procuramos que los cambios sean pequeños, para realizarlos rápidamente. En eso consiste, justamente, la práctica de integración continua: trabajar en cambios pequeños que se integran al *branch* principal frecuentemente. De esta manera, ganamos en sencillez y reducimos el riesgo de introducción de *bugs*.

Usamos un servidor de integración continua para reforzar esta práctica y volverla más efectiva. El *workflow* es el siguiente:

- Al comenzar una *User Story* creamos un *branch*. El *build* verde nos indica que partimos de una base que funciona correctamente.
- Cuando completamos la funcionalidad, *mergeamos* nuestros cambios con los existentes en el *branch* principal. Personalmente, prefiero hacer un *rebase* para dejar los *commits* ordenados.
- Una vez combinados los cambios, corremos los tests. Generalmente, es más cómodo y rápido *pushear* los cambios y dejar que el servidor de integración continua lo haga.
- Con el *build* verde, estamos listos para crear el *pull request* y *mergear* los cambios (momento indicado para pedir *code review*). Si, por el contrario, falla algún test, debemos corregirlo. No podemos *mergear* hasta no hacerlo.
- Nuestro trabajo no termina aquí. Una vez *mergeado*<sup>76</sup> nuestro *branch* al *branch* principal, verificamos que el *build* esté verde. De existir un problema, resultará fundamental su inmediata corrección a fin de evitar el bloqueo del resto del equipo.

Integrar continuamente hace nuestro trabajo sustancialmente más sencillo. Nos sentimos más seguros, ya que estamos trabajando en una versión reciente del código. Cuando rompemos algo, tenemos los cambios en la cabeza, facilitando enormemente su corrección, lo que reduce la probabilidad de introducción de *bugs*, difíciles de detectar y corregir.

En conclusión, la práctica de integración continua también resulta fundamental en nuestra búsqueda por alcanzar la excelencia técnica.

---

<sup>76</sup> Anglicismo muy utilizado, equivalente al verbo fusionar de nuestra lengua.

## La pared de *Scrum*

*“No podemos cumplir nuestros compromisos, no podemos hacer releases, nuestros clientes se frustran y se enojan. Parece que Scrum estuviera roto.”<sup>77</sup>*

¿Qué pasa cuando hacemos *Scrum*, pero no utilizamos las prácticas previamente descritas? Pues nos chocamos con la pared de *Scrum*<sup>78</sup>. La metáfora es muy visual. No es posible hacer desarrollo de software sin usar las prácticas mencionadas en las secciones anteriores.

## Manifiesto de *Software Craftsmanship*

¿El Manifiesto Ágil debería haber prestado mayor atención a los aspectos técnicos? Escuché esta pregunta en reiteradas ocasiones. Marick, uno de los signatarios, en su presentación “7 años después: lo que el manifiesto dejó afuera”<sup>79</sup>, contesta esta pregunta.

Posteriormente, surgió un movimiento que, esta vez sí, puso la carga fundamental en el aspecto técnico. Se redactó un manifiesto de *software craftsmanship*<sup>80</sup> (en la actualidad, los *craftsmen* se volvieron *crafters* gracias al movimiento feminista) como contraposición a su precedente ágil, muy popular a esta altura, donde se expuso la valoración del software “bien hecho” así como la necesidad de una comunidad de profesionales capacitados para realizar esta labor. Los creadores de este manifiesto nos hicieron comprometer con el respeto por la profesión, abrazando un sentimiento de orgullo por lo que hacemos. Finalmente, establecieron un

<sup>77</sup> *You can't keep your commitments, you can't release software, your customers get annoyed and angry, it looks like Scrum is broken.*

<sup>78</sup> <https://www.allankellyassociates.co.uk/archives/869/scrum-wall-another-agile-failure-mode/>

<sup>79</sup> *Seven Years Later: What the Agile Manifesto Left Out -* [https://www.stickyminds.com/sites/default/files/presentation/file/2013/08BADPR\\_WK1.pdf](https://www.stickyminds.com/sites/default/files/presentation/file/2013/08BADPR_WK1.pdf)

<sup>80</sup> <http://manifesto.softwarecraftsmanship.org/>

camino para llegar a la maestría, tomando como modelo los oficios tradicionales donde se comienza siendo un aprendiz que copia y repite al *master* hasta volverse uno con el correr del tiempo.

## Sobre prácticas técnicas y prácticas de gestión - por Nicolás Paez

Me acerqué a *Agile* allá por 2004 aproximadamente. Lo hice desde *eXtreme Programming (XP)*: un enfoque ágil que es muy explícito respecto de la excelencia técnica y que, en términos generales, incluye las prácticas mencionadas en las secciones precedentes. Durante un tiempo, hasta 2005 / 2006, *XP* fue el método ágil más popular. Luego fue desplazado por *Scrum*, el enfoque ágil más utilizado en la actualidad. Paralelamente, también se popularizaron algunas implementaciones de *Scrum*, que en 2009 fueron bautizadas por Martin Fowler como *Flaccid Scrum*<sup>81</sup>. Estas implementaciones se caracterizan por la ausencia de prácticas técnicas lo cual, en términos de desarrollo ágil de software, implica una contradicción. Con el correr del tiempo, surgieron distintos movimientos que, a veces intencionalmente y otras no, fueron disociando las prácticas ágiles de índole técnica de aquellas ligadas a la gestión. Estas últimas empezaron a denominarse “prácticas ágiles” a secas, mientras que las primeras, “prácticas de ingeniería”. Constituye una evidencia de ésto el reporte anual sobre el estado de Agile publicado por Version One<sup>82</sup>.

Creo que este fenómeno de la “falta de excelencia técnica” se debe en parte a la expansión de *Agile* hacia otros contextos por fuera del desarrollo de software donde las denominadas prácticas técnicas como TDD y *pair-programming* no tienen aplicación. Alguien podrá argumentar que si mi contexto es relativo a la confección de prendas entonces en lugar de *pair-programming* podría hacer “*pair-bordado*”, pero precisamente

---

<sup>81</sup> <https://martinfowler.com/bliki/FlaccidScrum.html>

<sup>82</sup> <https://stateofagile.versionone.com/> - Algunos referentes de la comunidad ágil como Joshua Kerievsky han manifestado explícitamente su oposición a esta tendencia. <https://www.linkedin.com/pulse/stop-calling-them-technical-practices-joshua-kerievsky/>

no sería *pair-programming* sino algo análogo que podría o no traer un beneficio.

En mi ejercicio profesional también fui testigo de esta “agilidad sin excelencia técnica”. He visto muchos equipos diciendo que trabajaban “a lo *Agile*” pero con una completa ausencia de excelencia técnica (más aún, en algunos casos no tenían excelencia de ningún tipo). En un intento de no hacer generalizaciones prejuiciosas e incorrectas, decidí buscar sustento formal para mis percepciones. Así fue como me sumé a un grupo de investigación en la Universidad Nacional de Tres de Febrero.

Con este grupo nos propusimos, como primer objetivo, entender cómo era el uso de prácticas ágiles en América Latina, analizándolas desde una perspectiva dual de prácticas técnicas y de gestión. Como metodología de investigación decidimos usar encuestas. Establecimos el conjunto de prácticas a estudiar y definimos un criterio para categorizarlas:

- **Prácticas de gestión:** son aquellas que nos ayudan a organizar el proyecto y que tienen aplicación incluso en proyectos que no sean de desarrollo de software.
- **Prácticas técnicas:** son aquellas exclusivas al desarrollo de software y cuya aplicación no es posible (o carece de sentido) en proyectos que no sean propios del área.

Partiendo de esta definición, elegimos un total de seis prácticas: tres de gestión y tres técnicas y armamos un cuestionario para entender el nivel de uso de éstas. Decidimos realizar la encuesta en una conferencia de *Agile* y, para tal fin, consultamos a los participantes del *Agile Open Camp* Bariloche 2016. Los resultados, junto con nuestro análisis y conclusiones consecuentes, fueron publicados en el Congreso Nacional de Ingeniería Informática y Sistemas de Información CONAIISI 2016. Repetimos la encuesta en dos ocasiones. En el *Agile Open Camp Chile* 2017, estudiando en este caso ocho prácticas: las seis que constituyan el estudio anterior más dos nuevas. Finalmente, extendimos nuestro estudio a diez prácticas, encuestando a los participantes de la Conferencia Latinoamericana de Métodos Ágiles 2017.

El listado final de prácticas de nuestro estudio incluyó:

Prácticas de gestión:

- Proceso iterativo
- *Release* frecuente
- Retrospectivas
- Auto-organización
- Propiedad Colectiva

Prácticas técnicas:

- Automatización de pruebas
- Integración Continua
- *Test-Driven Development*
- Diseño Emergente
- *Pair-Programming*

La siguiente tabla muestra comparativamente distintas características de las tres conferencias donde realizamos nuestras encuestas:

Característica	<i>Agile Open Camp 2016</i>	<i>Agile Open Camp 2017</i>	Ágiles 2017
Cantidad de encuestas luego de depuración	44	49	107
Participantes de la conferencia	98	79	~800
Cantidad de prácticas estudiadas	6	8	10

En las 3 etapas/encuestas de nuestro estudio llegamos a las mismas conclusiones:

Las prácticas de gestión tienen un mayor grado de adopción que las prácticas técnicas. Más concretamente y de acuerdo con nuestra última encuesta, las de gestión tienen un nivel de adopción que supera el 63%, mientras que el nivel de adopción de las técnicas no supera el 60%.

La cantidad de prácticas utilizadas aumenta junto con la experiencia de la organización en el uso de métodos ágiles, o sea: a mayor experiencia, mayor adopción de prácticas.

La diferencia de adopción entre prácticas técnicas y de gestión disminuye a medida que aumenta la experiencia de la organización en el uso de métodos ágiles. Organizaciones con poca experiencia en *Agile* usan pocas prácticas (~4), en general de gestión. Organizaciones con más experiencia usan más prácticas (~7) y la mezcla entre aquellas propias de la gestión y las técnicas es más pareja.

El trabajo final de nuestro estudio que agrupa las 3 encuestas fue publicado en la *International Conference on Agile Software Development* 2018. Quienes gusten ver los detalles del estudio pueden consultarla en: [https://link.springer.com/chapter/10.1007%2F978-3-319-91602-6\\_10](https://link.springer.com/chapter/10.1007%2F978-3-319-91602-6_10).

Una posible implicancia de estas conclusiones es que para muchas organizaciones el camino de adopción de *Agile* comienza por las prácticas de gestión e incorpora prácticas técnicas en una segunda instancia. Personalmente he visto esta situación en reiteradas ocasiones y en muchos casos esa segunda instancia nunca llega. Al mismo tiempo, trabajando como consultor, ayudando en la adopción de *Agile*, introduce las prácticas técnicas desde el inicio, pero tampoco funcionó. Inicialmente el equipo las incorporó, pero cuando el zapato apretó, sin prácticas de gestión establecidas ni disciplina para cumplir los acuerdos de trabajo, las dejó de lado, retornando “al lado oscuro”, donde no hay excelencia técnica.

Esto me lleva a pensar que el camino de adopción de *Agile* debe ser holístico y orgánico, incorporando simultáneamente prácticas de gestión y prácticas ágiles, pues no hay agilidad posible sin excelencia técnica.

## Conclusión

Buscamos la excelencia técnica continuamente. Para alcanzarla, trabajamos de forma disciplinada usando las prácticas ingenieriles descritas originalmente en *eXtreme Programming*. Queremos desarrollar código “limpio”, bien diseñado y bien testeado. Es la manera más eficiente de lograr el valor que buscamos aportar a nuestros clientes y la más efectiva para desarrollar software sustentablemente, con la calidad esperada.

# Conclusión

Desarrollamos software buscando maximizar el flujo de valor generado para nuestros clientes. Deseamos que sea sustentable, frente a las modificaciones del contexto. Trabajamos en un ambiente de gran calidad humana, empoderados para colaborar del mejor modo.

Nuestra Metodología, nuestra manera de organizarnos y de trabajar, responde a esta visión. Está centrada en generar valor, es liviana, basada en las personas y busca maximizar la colaboración con nuestros clientes, el feedback y la calidad.

Recapitulemos:

Empezamos por un *Product Discovery*, que permite entender la propuesta de valor a la vez que sienta las bases de una relación de colaboración con nuestros clientes. Participamos en él algunas de las personas ligadas al proyecto, reduciendo así el desperdicio. Las herramientas que usamos son livianas, permitiendo que todo el equipo colabore eficientemente.

A partir del *Product Discovery*, escribimos *User Stories*, pequeños incrementos funcionales que iremos mostrando al *Product Owner* para maximizar el feedback. Las almacenamos en el *Backlog*, un artefacto liviano que nos permite consensuar las prioridades y que refinamos y modificamos a medida que incorporamos aprendizaje.

Usando este *Backlog* inicial, hacemos una estimación superficial para entender las dimensiones del producto y pensar el equipo que formará parte del proyecto, teniendo en cuenta las restricciones de tiempo y dinero existentes.

Los artefactos previamente descriptos constituyen el plan inicial, que también se modificará cuando aprendamos sobre el producto, el proceso y nosotros mismos.

Conformamos los equipos y co-creamos un ambiente donde trabajar de la mejor manera. Compartimos la visión establecida durante el *Product Discovery*. Estamos empoderados porque sabemos que se obtienen

mejores resultados de este modo. Esto nos hace sentir motivados y energizados. Maximizamos el *bandwidth* de comunicación, porque reconocemos su importancia.

Desarrollamos software iterativa e incrementalmente. Involucrar a nuestros clientes durante el proceso de construcción resulta fundamental para obtener el producto que buscamos con la calidad esperada.

Buscamos la excelencia técnica continuamente. Sabemos que es fundamental para alcanzar la calidad necesaria, para ser sustentables en nuestro proceso de desarrollo y para poder adaptarnos ante los cambios externos. Lograr esto requiere mantener el código limpio: con el mejor diseño posible y bien testeado. Prácticas como *pair-programming*, *test driven development* y *refactor* continuo son claves en esta búsqueda.



# Bibliografía

- Adzic, G. (2009). *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*.
- Adzic, G. (2011). *Specification by Example: How Successful Teams Deliver the Right Software, 1st Edition*.
- Anderson, D. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*.
- Appelo, J. (2010). *Management 3.0: Leading Agile Developers, Developing Agile Leaders*. Addison-Wesley Signature Series.
- Beck, K. (2000). *Planning eXtreme Programming (The Xp Series)*.
- Beck, K. (2002). *Test Driven Development: By Example*.
- Brooks, F. J. (1995). *The Mythical Man Month*.
- Cagan, M. (2008). *Inspired: How To Create Products Customers Love*.
- Cockburn, A. (2006). - *Agile Software Development: The Cooperative Game (2nd Edition)*.
- Cohn, M. (2004). *User Stories Applied: For Agile Software Development*.
- Cohn, M. (2005). *Agile Estimating and Planning*.
- Crispin, L., & Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*.
- De Marco, T. (1999). *Peopleware: Productive Projects and Teams*.
- Fowler, M. (199). *Refactoring: Improving the Design of Existing Code*.
- Highsmith, J. (2009). *Agile Project Management: Creating Innovative Products, 2nd Edition*. Agile Software Development Series.
- Kniberg, H. (2011). *Lean from the Trenches: Managing Large-Scale Projects with Kanban*.

- Lencioni, P. (2002). *The Five Dysfunctions of a Team: A Leadership Fable*.
- Martin, R. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*.
- Patton, J. (2014). *User Story Mapping: Discover the Whole Story, Build the Right Product* -.
- Poppoedieck00, M. &. (2003). *Lean Software Development: An Agile Toolkit*.
- Ries, E. (2011). *The Lean Startup : How Constant Innovation Creates Radically Successful Businesses*.
- Rothman, J. (2007). *Manage It!: Your Guide to Modern, Pragmatic Project Management*.
- Rothman, J. (2009). *Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects*. Pragmatic Programmers.

# Sobre el autor y los invitados

Federico Zuppa: Federico se licenció en Ciencias de la Computación en la Universidad Nacional de San Luis en 2000. Comenzó luego una Maestría en Ingeniería del Software (que no finalizó), donde tuvo su primer contacto con las Metodologías Ágiles. Ya en Buenos Aires, trabajó en Electronic Data Systems y en Globant donde aplicó Scrum por 1<sup>ra</sup> vez y, también, donde conoció a Emilio Gutter y Jorge Silva, quienes posteriormente lo invitarían a formar parte de 10Pines. En el 2008 co-organizó las 1<sup>ras</sup> jornadas de Metodologías Ágiles Latinoamericanas. Los últimos 10 años trabajó en 10Pines, haciendo desarrollo, dictando cursos de agilidad y coacheando equipos.

Jorge Silva: es ingeniero en informática de la UTN trabajando en la industria del software por mas de 15 años. También programador, docente, orador, padre y jugador de basket aficionado. Es uno de los fundadores de 10Pines, empresa argentina de software con una gestión horizontal que desafía cada aspecto del management tradicional. Actualmente se dedica a ayudar a convertir ambientes de trabajo en lugares mas sanos.

Juan Manuel Carraro: *Strategic Program Sr. Manager* en Accenture. Profesional de *Customer & User Experience*. Coach de *Design Thinking & Service Design*. Autor y Expositor. Juan Manuel es pionero en el campo de experiencia de usuario y diseño centrado en las personas en Latinoamérica. Desde el año 2000 trabaja para algunas de las principales compañías globales como consultor y director de *Customer Experience* y *User Experience*. A través de su actividad como autor, docente, mentor y speaker ayuda a innumerables compañías, profesionales y equipos a formarse y lograr procesos de cambio organizacional para centrarse en el cliente.

Germán Gaitan: Soy Licenciado en Análisis de sistemas de la Universidad de Buenos Aires, con 25 años de experiencia en desarrollo de software. Luego de una fuerte experiencia en BI y gestión de proyectos, en los últimos años me enfoqué en Metodología de IT, impulsando la transformación hacia la agilidad a través de la adopción de Scrum y otras prácticas, que tuvieron un fuerte impacto en la entrega de producto de calidad y en la relación con nuestros clientes internos.

Gisela Decuzzi: estudió Ingeniería en Sistema de Información en la UTN (FRBA), desde 2007 es docente universitaria de materias relacionadas con programación orientada a objetos. Trabaja en la industria desde 2008 donde tuvo sus primeros acercamientos a prácticas ágiles pero no se replantearía lo aprendido sino hasta 2012 donde empezó a trabajar en 10Pines aprendiendo y aplicando diariamente técnicas ágiles no solo para el desarrollo de software sino también para la gestión de la empresa.

Nicolás Paez: es Ingeniero en Informática de la Universidad de Buenos Aires. Tiene más de 15 años de experiencia trabajando en la industria de software. Reparte su tiempo entre la industria y la academia realizando tareas de desarrollo, docencia e investigación. Es profesor en UBA y UNTreF. Ha realizado estudios de posgrado en Tecnología Aplicada a la Educación y ha publicado varios artículos y libros en el área de Ingeniería de Software.

Hernán Wilkinson: Programador apasionado. Lic. en Ciencias de la Computación de la UBA. Fundador de 10Pines y de FAST (Fundación Argentina de Smalltalk). Profesor de la FCEyN de la UBA. Enseña el Paradigma de Objetos y técnicas Agiles en la universidad y la industria. *Key Note Speaker* de congresos nacionales e internacionales. Contribuye en proyectos opensource de Smalltalk. Promueve las organizaciones horizontales y las metodologías ágiles. Lo pueden seguir en @hernanwilkinson

# Agradecimientos

Mi primer agradecimiento es para 10Pines: a todas y todos por apoyarme y permitirme que cumpla este sueño. Particularmente, a mis socios y amigos: Alejandra, Emilio, Hernán, Jorge y Darío por acompañarme en todo este camino.

A Cris, mi amigo y editor, por las horas dedicadas, la paciencia para enseñarme la lengua española y por aguantar todos los anglicismos que usamos los desarrolladores. He aprendido muchísimo durante estos dos años.

A Emma, por darle vida a cada uno de los capítulos con sus increíbles ilustraciones. También a Gaby por su gran colaboración en todo este proceso ¡Sin ustedes no sería lo mismo!

A Jorge, Juan, Germán, Gise, Nico y Hernán por sus valiosos aportes, que enriquecen enormemente el libro.

A Belisa, por ser la 2<sup>da</sup> lectora de cada una de estas páginas (Cris me dijo que el primero era yo). Ella dice haber aprendido mucho de agilidad, ¿Será verdad?

Al resto de mi familia: Valen, Carlo, Mirta y Lucía.