



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Scrum: Breve Guía y Reseña

Proyecto Personal - 2017

Resumen

Integrante	LU	Correo electrónico
Ambroa, Nicolás	229/13	ambroanicolas@hotmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - Pabellón I

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Argentina

Tel/Fax: (54 11) 4576-3359

<http://exactas.uba.ar>

Índice

1. <i>Sprints</i>	10
1.1. <i>Timeboxed</i>	10
1.1.1. Establece Límite de <i>WIP</i>	10
1.1.2. Prioriza el Trabajo sin Perfeccionismo Innecesario	10
1.1.3. Motiva el Cierre	10
1.1.4. Demuestra Progreso	10
1.2. Corta Duración (15 a 30 días)	10
1.2.1. Facil Planificación y Rápida Validación	10
1.2.2. Mejor Retorno de Inversión	11
1.2.3. Entusiasmo Rejuvenecido	11
1.3. Duración Consistente	11
1.3.1. Mantiene el Ritmo	11
1.3.2. Simplifica la Planificación	11
1.4. Sin Cambios Drásticos de Objetivos	12
1.4.1. Mutuo Compromiso	12
1.4.2. Mejora Concentración	12
1.4.3. Controla el Nivel de Cambio	12
1.4.4. Terminación Anormal del <i>Sprint</i>	12
1.5. <i>Definition of Done</i> y PSP (<i>Potentially Shippable Product</i>)	13
1.5.1. Definición Evolutiva	13
2. Requerimientos & User Stories	14
2.1. Introducción	14
2.2. Conversaciones Como Herramienta	14
2.3. Refinamiento Iterativo	14
2.4. User Stories	14
2.4.1. Card	14
2.4.2. Conversation	15
2.4.3. Confirmation	15
2.5. Nivel de Detalle	15
2.5.1. Epic	15
2.5.2. Themes	15
2.5.3. Stories	16
2.5.4. Tasks	16
2.6. Criterio INVEST	16
2.6.1. Independent	16
2.6.2. Negotiable	16
2.6.3. Valuable	16

2.6.4.	Estimable	17
2.6.5.	Sized Appropriately	17
2.6.6.	Testable	17
2.7.	Requerimientos No Funcionales	17
2.8.	KAS: <i>Knowledge Acquisition Stories</i>	17
2.9.	Generación de <i>Stories: User-Story</i> Writing Workshop	17
2.10.	Generación de <i>Stories: Story Mapping</i>	18
3.	Product Backlog	18
3.1.	DEEP: Construyendo el Backlog	18
3.1.1.	Detailed Appropriately	19
3.1.2.	Emergent	19
3.1.3.	Estimated	19
3.1.4.	Prioritized	19
3.2.	Grooming	19
3.2.1.	Quién Realiza el <i>Grooming</i> ?	20
3.2.2.	Cuándo se Realiza el <i>Grooming</i> ?	20
3.3.	Definition of Ready	21
3.4.	Manejo de Flujo	21
3.4.1.	Manejo de Flujo de Lanzamiento	21
3.4.2.	Manejo de Flujo de <i>Sprint</i>	22
3.5.	Cuáles y Cuántos <i>Product Backlogs</i> Usar	22
3.5.1.	Qué es un Producto?	22
3.5.2.	<i>Backlogs Jerárquicos</i> para Grandes Productos	22
3.5.3.	Múltiples Productos para un Equipo	22
4.	Estimación y Velocidad	23
4.1.	Qué y Cuándo Estimar	23
4.1.1.	Estimaciones para el <i>Portfolio</i>	23
4.1.2.	Estimaciones para el <i>Product Backlog</i>	23
4.1.3.	Estimaciones para las Tareas de un <i>PBI</i>	24
4.2.	Conceptos de Estimación para <i>PBI</i>	24
4.2.1.	Estimar como Equipo	24
4.2.2.	La Estimación no es un Compromiso	24
4.2.3.	Exactitud vs Precisión	24
4.2.4.	Utilizar Tamaños Relativos	24
4.3.	Unidades de Estimación para <i>PBI</i>	25
4.3.1.	Story Points	25
4.3.2.	Ideal Days	25
4.4.	Planning Poker	25

4.4.1.	Cómo Jugar	25
4.4.2.	Beneficios	25
4.5.	Velocidad	26
4.5.1.	Calculando un Rango de Velocidad	26
4.5.2.	Prediciendo Velocidad	26
4.5.3.	Afectando la Velocidad	27
4.5.4.	La Velocidad Mal Utilizada	27
5.	Deuda Técnica	28
5.1.	Consecuencias de la Deuda Técnica	28
5.1.1.	Punto de Inflexión Impredecible	28
5.1.2.	Tiempo de Entrega en Alza	28
5.1.3.	Número Significativo de Defectos	28
5.1.4.	Costos de Desarrollo y Soporte Creciente	29
5.1.5.	Consecuencias Humanas	29
5.2.	Causas de la Deuda Técnica	29
5.2.1.	Presión para Alcanzar Deadlines	29
5.2.2.	Mito: Menor Testeo Incrementa Velocidad	29
5.2.3.	Deuda Construída sobre Deuda	29
5.3.	Como Controlar la Deuda Técnica	29
5.3.1.	Buenas Prácticas Técnicas	30
5.3.2.	<i>Definition of Done</i>	30
5.3.3.	Entendimiento de la Economía detrás de la Deuda Técnica	30
5.4.	Visibilidad de la Deuda Técnica	31
5.4.1.	Visibilidad a Nivel Negocio	31
5.4.2.	Visibilidad a Nivel Técnico	31
5.5.	Pagando la Deuda Técnica	31
5.5.1.	No Toda Deuda Debería Ser Pagada	31
5.5.2.	Boy Scout Rule: Pagar la Deuda Cuando se Encuentre	32
5.5.3.	Pagar la Deuda de Forma Incremental (en cada <i>Sprint</i>)	32
5.5.4.	Pagar la Deuda más Importante Primero	32
5.5.5.	Pagar la Deuda y Entregar Trabajo de Valor	32
6.	Product Owner	33
6.1.	Responsabilidades Principales	33
6.1.1.	Manejo de la Economía	33
6.1.2.	Participación en la Planificación	33
6.1.3.	<i>Grooming</i>	33
6.1.4.	Definir Criterios de Aceptación (y asegurarse que se cumplan)	33
6.1.5.	Colaborar con el Equipo de Desarrollo	33
6.2.	Características y Habilidades	34

6.2.1.	Conocimiento del Dominio	34
6.2.2.	Habilidades Sociales	34
6.2.3.	Decisiones y Responsabilidad	34
6.3.	Ejemplo de un Día en la Vida del <i>Product Owner</i>	34
6.4.	Quién Debería ser <i>Product Owner</i> ?	35
6.4.1.	Desarrollo Interno	35
6.4.2.	Desarrollo Comercial	36
6.4.3.	Desarrollo con <i>Outsourcing</i>	36
6.4.4.	Desarrollo con Componentes	36
6.5.	<i>PO</i> Combinado con Otros Roles	36
6.5.1.	<i>Product Owner</i> Team	36
6.5.2.	Product Owner Proxy	36
6.5.3.	Chief Product Owner	37
7.	<i>ScrumMaster</i>	37
7.1.	Responsabilidades Principales	37
7.1.1.	Coaching	37
7.1.2.	Líder en Servicio	38
7.1.3.	Autoridad del Proceso	38
7.1.4.	Removedor de Impedimentos	38
7.2.	Características y Habilidades	38
7.2.1.	Experto en Scrum	38
7.2.2.	Cuestionador Paciente	38
7.2.3.	Colaborativo y Protector	38
7.2.4.	Transparente	39
7.3.	Un día en la vida del <i>ScrumMaster</i>	39
7.4.	Cumpliendo el Rol	39
7.4.1.	Quién Debería ser <i>ScrumMaster</i> ?	39
7.4.2.	Trabajo de Tiempo Completo?	40
7.4.3.	<i>ScrumMaster</i> Combinado con Otros Roles	40
8.	<i>Development Team</i>	41
8.1.	Responsabilidades Principales	41
8.1.1.	Ejecución del <i>Sprint</i> y Tareas de Planificación	41
8.2.	Características y Habilidades	41
8.2.1.	Auto-Organizable	41
8.2.2.	Diversamente Multifuncional	41
8.2.3.	Habilidades en Forma de T	42
8.2.4.	Actitud de Mosquetero	42
8.2.5.	Comunicación Eficiente	43
8.2.6.	Tamaño Apropiado	43

8.2.7. Concentración y Compromiso	43
8.2.8. Trabajo a un Ritmo Sustentable	44
9. Estructuras de Equipo de <i>Scrum</i>	44
9.1. Equipos de <i>Feautres</i> vs Equipos de Componentes	44
9.2. Coordinación entre Múltiples Equipos	45
9.2.1. <i>Scrum of Scrums</i>	46
9.2.2. <i>Release Train</i>	46
10. Managers en <i>Scrum</i>	48
10.1. Responsabilidades de un <i>Functional Manager</i>	48
10.1.1. Construir Equipos	48
10.1.2. Nutrir Equipos	49
10.1.3. Alinear y Adaptar el Entorno para Adoptar <i>Agile</i>	49
10.1.4. Manejar el Flujo de Creación de Valor	49
10.2. Project Managers	50
10.3. Reteniendo el rol de <i>Project Manager</i>	51
11. Principios de Planificación en <i>Scrum</i>	52
11.1. No se Puede Obtener la Planificación Correcta por Adelantado	52
11.2. La Planificación por Adelantado Debería Ayudar sin ser Excesiva	52
11.3. Foco en Adaptación y Replanificación sobre el Plan	52
11.4. Manejar Correctamente el Inventario de Planificación	52
11.5. Favorecer <i>Releases</i> Pequeños y mas Frecuentes	53
12. Planificación Multinivel	54
13. <i>Portfolio</i> Planning	54
13.1. Resumen	55
13.1.1. Timing	55
13.1.2. Participantes	55
13.1.3. Proceso	55
13.2. Estrategias de Planificación	55
13.2.1. Optimizar para las Ganancias del Ciclo de Vida de un Producto	55
13.2.2. Calcular el Costo del Retraso	56
13.2.3. Estimar para Precisión, no Exactitud	57
13.3. Estrategias para el <i>Inflow</i> (Flujo de Entrada)	57
13.3.1. Aplicando el Filtro Económico de la Empresa	57
13.3.2. Balancear las Fechas de Entrada y Salida	57
13.3.3. Aprovechar Oportunidades Emergentes Rápidamente	58
13.3.4. Planear <i>Releases</i> más Pequeños y Frecuentes	58
13.4. Estrategias para el <i>Outflow</i>	58

13.4.1. Disminu��r el Trabajo Ocioso en vez de Trabajadores Ociosos	58
13.4.2. Establecer L��mite para el <i>WIP</i>	58
13.5. Estrategias Dentro del Proceso	58
13.5.1. Usar Econom��as Marginalistas	59
14.Product Planning	60
14.1. Resumen	60
14.1.1. Timing	60
14.1.2. Participantes	60
14.1.3. Proceso	60
14.2. Ejemplo Empresa ReviewEverything: <i>SmartReviewForYou</i> (SR4U)	60
14.2.1. Visi��n	61
14.2.2. Creaci��n de un <i>Product Backlog</i> de Alto Nivel	62
14.2.3. Definici��n del <i>Roadmap</i> del Producto	62
14.2.4. Otras Actividades	63
14.3. <i>Envisioning</i> Econ��micamente Sensible	63
14.3.1. Apuntar a un Umbral de Confianza Realista	63
14.3.2. Concentrarse en Horizontes Cortos, Actuando R��pido	64
14.3.3. Invertir en Aprendizaje Validado	64
14.3.4. Usar Financiaci��n Provisoria/Incremental	64
15.Release Planning	65
15.1. Resumen	65
15.1.1. Timing y Participantes	65
15.1.2. Proceso	65
15.2. Restricciones del <i>Release</i>	66
15.2.1. Fixed Everything	66
15.2.2. Fixed Scope & Date	66
15.2.3. Fixed Scope	67
15.2.4. Fixed Date	67
15.3. Tareas de <i>Grooming</i>	67
15.4. Refinando <i>MRF's</i>	67
15.5. <i>Sprint Mapping</i> (�� <i>PBI Slotting</i>)	67
15.6. <i>Release Planning</i> para Fixed Date	68
15.7. <i>Release Planning</i> para Fixed Scope	69
15.8. Calculando Costos del <i>Release</i>	70
15.9. Comunicando el Progreso	70
15.9.1. <i>Fixed Scope-Release Burndown Chart</i>	70
15.9.2. <i>Fixed Scope-Release Burnup Chart</i>	70
15.9.3. Comunicando Progreso en <i>Releases Fixed-Date</i>	71

16. <i>Sprint Planning</i>	71
16.1. Resumen	72
16.1.1. Timing y Participantes	72
16.1.2. Proceso	72
16.2. Enfoques para la <i>Sprint Planning</i>	72
16.2.1. Separación en Dos Partes	72
16.2.2. Sin Separación	73
16.3. Determinando la Capacidad	73
16.3.1. Qué es?	73
16.3.2. Capacidad en <i>story points</i>	73
16.3.3. Capacidad en <i>effort hours</i>	73
16.4. Seleccionando <i>PBI's</i>	74
16.5. Adquiriendo Confianza	74
17. Ejecución del <i>Sprint</i>	75
17.1. Planificación de la Ejecución del <i>Sprint</i>	75
17.2. Manejo de Flujo de Trabajo	75
17.2.1. Trabajo en Paralelo	75
17.2.2. Qué Trabajo Comenzar y Cómo Organizar Tareas	76
17.3. <i>Daily Scrum</i>	76
17.4. Prácticas Técnicas para el Rendimiento	76
17.5. Comunicación	77
17.5.1. <i>Task Board</i>	77
17.5.2. <i>Sprint Burndown y Burnup Chart</i>	77
18. <i>Sprint Review</i>	78
18.1. Trabajo previo a la <i>Sprint Review</i>	79
18.1.1. Determinar a Quién Invitar y Programar la Actividad	79
18.1.2. Confirmar que el Trabajo del <i>Sprint</i> Esta Terminado	79
18.1.3. Preparar la Demo y Determinar Participantes	79
18.2. Enfoque	80
18.2.1. Resumir y Demostrar	80
18.2.2. Discutir y Adaptar	80
18.3. Problemas de la <i>Review</i>	81
18.3.1. Aprobaciones	81
18.3.2. Asistencia Esporádica	81
18.3.3. Grandes Grupos de Desarrollo	81
19. <i>Sprint Retrospective</i>	81
19.1. Participantes	82
19.2. Trabajo Previo	82

19.2.1. Definir el Foco de la Retrospectiva	82
19.2.2. Seleccionar Ejercicios y Recolectar Información	82
19.2.3. Estructurar la Retrospectiva	83
19.3. Enfoque	83
19.3.1. Crear la Atmósfera	83
19.3.2. Compartir Contexto	83
19.3.3. Identificar Percepciones, Mejoras y Conocimiento	84
19.3.4. Determinar Acciones	85
19.4. Seguimiento sobre el <i>Sprint</i>	86
19.5. Problemas de la Retrospectiva	86

20. <i>Fuentes</i>	87
---------------------------	----

1. *Sprints*

Los *Sprints* son la estructura organizacional central de la planificación en *Scrum*. Si bien cada organización realiza su propia implementación de ellos, a continuación veremos características que deberían ser comunes a todo *Sprint* y todo equipo.

1.1. *Timeboxed*

Cada *Sprint* sucede dentro de un *time-frame*, un intervalo de tiempo con fechas predefinidas para el comienzo y el final. Si bien la predicción de éstas fechas puede comenzar siendo poco precisa, la idea es que en cada *Sprint Retrospective* se vaya ajustando hacia lo ideal.

1.1.1. Establece Límite de *WIP*

Como en cada *Sprint* se trabaja en funcionalidades que se deberían terminar dentro del mismo, uno naturalmente no puede dejar una gran cantidad de trabajo empezado pero sin terminar, reduciendo así el impacto económico negativo del inventario *WIP*. Si uno tuviese mucho *WIP*, no podría implementar nuevas funcionalidades en un *Sprint*, haciendo que el mismo pierda valor.

1.1.2. Prioriza el Trabajo sin Perfeccionismo Innecesario

Como el tiempo es limitado y hay un *backlog* por terminar, se priorizan el trabajo más sencillo que más importante sea, generando urgencia (pero no presión negativa sobre el equipo de trabajo). A la vez, ésto acaba el perfeccionismo innecesario (aquellas funcionalidades que intentamos hacer de forma perfecta cuando en realidad con hacer un buen trabajo era suficiente). Al tener que acabar el *Sprint* en una fecha determinada, uno no puede ser profundamente perfeccionista, porque sino el *Sprint* corre riesgo de retraso o de recorte de funcionalidad.

1.1.3. Motiva el Cierre

El *timeboxing* motiva el cierre de las funcionalidades por el conocimiento de una fecha de terminación (a primer momento inamovible). Sin ésta fecha, no se tiene la urgencia de tener que entregar funcionalidades en una *deadline*, y el trabajo tiende a completarse de forma más lenta. Por ejemplo, sin una fecha fija de cierre podríamos decidir hacer un refactor de código útil, pero no necesario según los objetivos del *Sprint*. El refactor tranquilamente podría haberse hecho más adelante, lo cual permite entregar las funcionalidades actuales de forma más eficiente y alineada con las necesidades del cliente.

1.1.4. Demuestra Progreso

Otro beneficio del *timeboxing* es la rápida validación del trabajo realizado hasta el momento (en el final del *Sprint*). Permite avanzar con mayor seguridad sobre features de todos los tipos, como también informarles con alta periodicidad a los *stakeholders* sobre el trabajo terminado y restante. En otras formas de reporte, solo se obtiene validación al final, aumentando fuertemente el coste de los errores.

1.2. Corta Duración (15 a 30 días)

1.2.1. Facil Planificación y Rápida Validación

Ésto se desprende del simple hecho que es más fácil planificar las dos semanas o mes siguientes que el próximo semestre (también requiere mucho menos tiempo). Además, al ser cortos los *Sprints*, podemos obtener información valiosa sobre cada funcionalidad de manera casi inmediata. En un proyecto con *Waterfall*,

sabemos que tendremos *checkpoints* luego de cada fase (Análisis Inicial, Diseño, Código, etc). Con *Scrum*, éstos *checkpoints* ocurren al final en cada *Sprint Review*. Ésto nos permite descartar caminos u opciones de desarrollo que tendrían un enorme costo económico de ser descubiertas mucho tiempo después.

Al descartar caminos inútiles de forma rápida, la corta duración del *Sprint* ayuda a acotar el error posible, minimizando su impacto. En el peor caso posible, solo se pierden 2 semanas de trabajo, frente a los potenciales meses perdidos dentro de una estructura más tradicional.

1.2.2. Mejor Retorno de Inversión

Al tener que tener un *PSP* (Potentially Shippable Product) listo luego de cada *Sprint*, significa que nuestro producto puede salir antes al mercado (o generar actualizaciones más rápido), lo cual se traduce en ingresos más rápidos y frecuentes.

1.2.3. Entusiasmo Rejuvenecido

Trabajando en proyectos con ciclos gigantes, es natural que la gente comience a perder el entusiasmo en terminar su trabajo, pues parece que el mismo no tiene fin alcanzable. Con *Sprints* de corta duración, ésto no suele ocurrir, pues el entusiasmo se puede renovar en cada iteración (al mejorar o implementar nuevas funcionalidades y verlas en acción).

1.3. Duración Consistente

La duración dentro de un *Sprint* debe ser consistente. Se puede cambiar la duración si se quiere refinar el proceso de *feedback* por ejemplo (pasar de 4 semanas a 2 para obtener valoraciones más rápidas). Alargar el *Sprint* porque queda trabajo en el *backlog* es más un síntoma de refinamiento necesario en el proceso de la organización y una oportunidad para mejorarlo que razón para cambiar el *Sprint*. Otra buena razón para alterar la duración son las circunstancias accidentales, como una repentina expansión del equipo de desarrollo (lo cual consume un *overhead* hasta que entren dentro del contexto del producto).

1.3.1. Mantiene el Ritmo

Una duración consistente le permite al equipo de desarrollo entrar en un “ritmo de *Sprint*”, lo cual habitúa ciertas actividades de *Scrum* de forma automática -pues siempre ocurren con la misma frecuencia la misma cantidad de veces- y les permite concentrarse en entregar trabajo de valor.

Éste ritmo también nivela la intensidad de los *Sprints*. En las metodologías más tradicionales, es muy común encontrar un marcado incremento en la intensidad en las últimas fases del proyecto. Ésta energía extra requerida en las fases finales tiene costos humanos altos, extenuando al equipo de desarrollo y requiriendo tiempo futuro de recuperación (aunque éste no sea explicitado por la organización, la velocidad del trabajo será menor). Bajo *Scrum*, cada *Sprint* tiene un perfil de intensidad que es similar al de los otros *Sprints*, lo cual nos permite trabajar de forma consistente y eficaz, mejorando el nivel humano del desarrollo.

1.3.2. Simplifica la Planificación

La duración consistente ayuda mucho a la hora de planificar. Conforme avanza el proceso de *Scrum* en un equipo, los integrantes van mejorando cada vez más las estimaciones de la planificación (elementos como *cuanto* puedo hacer dentro de un *Sprint*, entre otros). Dejando la duración fija, es más sencillo predecir correctamente la cantidad correcta de trabajo a realizar, pues uno está muy acostumbrado a trabajar sobre el mismo formato. Además, todo ésto permite consumir mucho menos tiempo planificando, pues ya se tiene una buena idea y posibles datos históricos (si no es el primer *Sprint*) bajo el mismo formato.

1.4. Sin Cambios Drásticos de Objetivos

Cada *Sprint* puede ser resumido por uno o más objetivos principales. Una vez que éstos sean definidos entre el equipo de desarrollo y el *Product Owner*, no deberían sufrir mayores cambios. Estos objetivos deberían describir el propósito a nivel negocio y el valor del *Sprint*, por ejemplo **“Soportar la Generación Inicial de Reportes”** ó **“Cargar y Optimizar los Datos del Mapa Americano”**.

Durante la planificación del *Sprint* el equipo de desarrollo debería utilizar el objetivo para determinar la prioridad de los ítems en el *Backlog*, con el objetivo final de obtener un conjunto de funcionalidades para entregar al final del mismo.

1.4.1. Mutuo Compromiso

El objetivo del *Sprint* representa el compromiso del equipo de desarrollo de cumplirlo, y el compromiso del *Product Owner* de no cambiarlo. Ésta responsabilidad sumada a la corta duración representa el balance del *Sprint* entre mantener la adaptabilidad al cambio alta frente a las necesidades del negocio, y permitirle al equipo de desarrollo crear funcionalidades de forma eficaz gracias a un objetivo claro y bien definido.

1.4.2. Mejora Concentración

Tener un objetivo inmutable ayuda a mantener el foco de los desarrolladores en tareas que ya se sabe (por el *Sprint Planning*) pueden realizarse de manera eficiente y rápida en una iteración. Cambiar o agregar objetivos altera la estructura del *Sprint*, lo cual suele venir junto con retrasos en la planificación. Ésto es diferente de la clarificación de objetivos. Un objetivo puede clarificarse para reducir la subjetividad del lenguaje natural, y eso no sólo está permitido, sino que es beneficioso.

Concretamente hablando, si tuviéramos que desarrollar una herramienta de filtros en un *Sprint*, sería contraproducente agregar algo como filtrado de reconocimiento por imagen (si no estaba ya planificado). Una clarificación sería, en el caso de querer ordenar una lista de personas, hablar con el *Product Owner* sobre el ordenamiento y que nos aclare que debe ser por orden alfabético.

1.4.3. Controla el Nivel de Cambio

Scrum es una metodología que adopta el cambio, pero en una forma que tenga sentido económico. El desperdicio de recursos generado por el cambio de objetivos aumenta cuando más avanzado se encuentre el *Sprint* (sentido común, revertir una funcionalidad cuesta más cuando está en desarrollo o terminada). Ésto afecta la planificación también, pues nunca se podría hacer una buena estimación del trabajo a realizar, ya que deberíamos considerar un overhead desconocido de cambios constantes.

Habiendo dicho todo ésto, se debe ser pragmático ante cualquier situación. Cambiar radicalmente la estructura del *Sprint* no es un pecado, solo debe hacerse sabiendo sus consecuencias, únicamente cuando tenga sentido económico (por ejemplo, si un competidor lanza un producto con mejor valor y funcionalidad, sería una buena idea revisar y alterar el plan de acción).

1.4.4. Terminación Anormal del *Sprint*

Bajo circunstancias excepcionales, el objetivo del *Sprint* puede volverse invalido y el equipo de *Scrum* puede aconsejarle al *Product Owner* terminar anormalmente el *Sprint*. Ésto implica realizar una *Retrospective* para analizar lo aprendido y las nuevas circunstancias, para luego reunirse con el *PO* para planear el nuevo *Sprint*, con diferentes objetivos y tareas.

El nuevo *Sprint* puede tener una duración menor a la tradicional (por ejemplo una semana si es sólo ajustes críticos), o una duración equivalente a lo restante del *Sprint* anterior sumado a la duración tradicional. Si faltaba 1 semana y el *Sprint* se corta, el nuevo podría durar 3 semanas si contiene funcionalidades pesadas. Claramente que también está el caso de aplicar la duración tradicional al nuevo *Sprint*. Todas las opciones tienen en común el concepto de “conservar el ritmo tradicional o volver a éste lo antes posible”.

En general, debido a la corta duración del *Sprint*, cancelarlo no suele ser una opción económica

viable frente a sus alternativas. Si sucede un error crítico de máxima prioridad en producción, suele ser mejor quitar una *feature* del *Sprint* para poder concentrarse en eso que cancelarlo completamente (para no potencialmente perder el trabajo hecho).

1.5. *Definition of Done* y PSP (*Potentially Shippable Product*)

Al final de cada *Sprint* se debe producir un *PSP*. Esto no quiere decir que el producto o la revisión vayan a lanzarse (eso es más una decisión de negocios), más bien significa que el proceso de integración (desde código a testeo completo) de la funcionalidad al producto está terminado. “Está terminado” siempre se refiere a un criterio predefinido por el grupo de desarrollo, llamado comúnmente ***Definition of Done***.

Un ejemplo de ***Definition of Done*** podría ser:

- Código completado (con formato estándar, refactorizado, con comentarios).
- Código revisado en etapa de *Code Review*.
- Código testado con Tests de Unidad.
- Código testado con Tests de Aceptación (criterios encontrados en cada *user story*).
- Documentación de Usuario terminada.
- Sin defectos (*bugs*) conocidos.

Claramente, el criterio es completamente variable y dependiente de cada organización en particular. Sin embargo, el objetivo principal debe mantenerse: proveer un alto nivel de confianza sobre la calidad del trabajo y su potencial de ser exportado a etapa de *producción*.

Si, para el final del *Sprint*, no podemos cumplir nuestra *Definition of Done* (tal vez por un error crítico en la última funcionalidad), se pasa el backlog sin terminar del *Sprint* actual al siguiente, dándole la prioridad correspondiente (si es un error crítico, la prioridad será máxima).

1.5.1. Definición Evolutiva

Este criterio de “trabajo terminado” suele evolucionar con el tiempo de forma iterativa, mientras que cambia el entorno exterior (dependencias con otros productos o *insourcing* de empresas) y el entendimiento sobre el dominio del producto. Por ejemplo, puede ocurrir que por circunstancias ajenas al equipo de desarrollo, no se pueda testear completamente la funcionalidad porque aún se está a la espera del hardware necesario.

2. Requerimientos & User Stories

2.1. Introducción

En contraste con un desarrollo de proyecto secuencial, en *Scrum* los requerimientos van siendo negociados en conversaciones y desarrollados **justo a tiempo**, con la especificación mínima para que el equipo pueda crear su funcionalidad. Se opera bajo la noción que uno no puede detallar todos los requerimientos de entrada, y que nuevos requerimientos irán surgiendo (o viejos serán descartados), en base a diferentes razones de negocio (como puede ser la relación costo/beneficio actualizada de un requerimiento).

Bajo *Scrum* los requerimientos se denominan *Product Backlog Items* (PBI), generalmente como *User Stories* (US). Inicialmente, éstos serán amplios y con un gran *valor de negocio* (BV). A medida que se aprende del dominio del problema y se refina el producto, ciertos *PBI* se irán especificando cada vez más hasta llegar a un tamaño adecuado para entrar en un *Sprint*, donde serán implementados.

2.2. Conversaciones Como Herramienta

En planificaciones más tradicionales, los requerimientos son establecidos completamente de entrada, lo cual incrementa exponencialmente el costo de cambios en el proyecto (en relación a la etapa de maduración del mismo). Una mejor manera de asegurarse que todas las funcionalidades deseadas sean implementadas es tener conversaciones frecuentes con aquellos que poseen la visión central del producto, a medida que se va construyendo.

Bajo *Scrum* utilizamos las conversaciones como herramienta principal de comunicación. Es un medio barato, que provee *feedback* inmediato, y bidireccional, lo cual permite debates que ayudan a identificar futuros problemas o nuevas funcionalidades.

2.3. Refinamiento Iterativo

Los requerimientos bajo *Scrum* no necesitan todos el mismo nivel de detalle. Al no tener la obligación de entregar un extenso y costoso documento de requerimientos para el desarrollo y diseño, podemos generarlos justo cuando son necesarios. El resultado es un *Product Backlog* que contiene requerimientos refinados bajo conversaciones, y requerimientos sin mucho detalle -aquellos sin importancia o nivel *epic*- ahorrándose así importantes recursos.

2.4. User Stories

Las *User Stories* representan un formato para asignar *BV* a ciertas funcionalidades o mejoras del producto. Son escritas bajo el modelo de una conversación, lo cual las hace fácilmente entendibles al público de negocios y adaptables al proceso iterativo. Se pueden resumir como **CCC: Card, Conversation & Confirmation**.

2.4.1. Card

Las cards representan el formato donde se escribe la *User Story*. Deben ser pequeñas (sólo capturan la esencia del requerimiento) y suelen seguir un formato similar a: “*Como <Rol> quiero <Objetivo> para <Beneficio>*”

Find Reviews Near Address
As a typical user I want to see unbiased reviews of a restaurant near an address so that I can decide where to go for dinner.

Figura 1: Ejemplo de *User Story*.

2.4.2. Conversation

Las *US* se van obteniendo de las conversaciones entre el *Product Owner*, el equipo de desarrollo y los stakeholders. La conversación ocurre a lo largo del proyecto, no una única vez (habrá una inicial y varias mientras el producto continúe). Cada conversación aumenta el entendimiento del dominio que posee el equipo de desarrollo, lo cual les permite llevar la *US* a un nivel de detalle mayor.

2.4.3. Confirmation

Una *US* contiene información de confirmación en forma de *condiciones de satisfacción*. Éstas representan criterios decididos con el *PO* para medir cuando una *User Story* está correctamente implementada. Pueden venir de varias formas, como por ejemplo tests de aceptación.

Upload File	Conditions of Satisfaction
As a wiki user I want to upload a file to the wiki so that I can share it with my colleagues.	Verify with .txt and .doc files
	Verify with .jpg, .gif, and .png files
	Verify with .mp4 files <= 1 GB
	Verify no DRM-restricted files

Figura 2: Ejemplo de *User Story* con tests de aceptación.

2.5. Nivel de Detalle

Como mencionamos anteriormente, tener todas las *stories* de un mismo tamaño es contraproducente. Si hiciéramos todas *stories* pequeñas (como para entrar en un *Sprint*), no podríamos capturar objetivos generales del producto. El problema inverso existe si sólo tenemos *stories* grandes. El punto ideal es tener *stories* de varios niveles de granularidad.

2.5.1. Epic

Una *epic* representa un objetivo a alcanzar en uno o múltiples lanzamientos del producto. Jamás pondríamos una *epic* en un *Sprint*, pues su duración puede ser desde meses en adelante. Son excelentes placeholders para lo que terminarán siendo decenas o cientos de *stories*.

2.5.2. Themes

Una *theme* representa una colección de *stories* que pertenecen a una misma área del producto (engloban funcionalidad). Se diferencian de *epics* pues su scope no debería ser mayor a unas semanas. Algunas se mapean directamente con el concepto de *feature*.

2.5.3. Stories

Representa parte de una *feature* o *theme* en una card lo suficientemente pequeña para entrar en un *Sprint*.

2.5.4. Tasks

Son la capa que le sigue a *stories*, y se concentran en el **cómo** de la implementación más que en el *qué*. Requieren horas para completarse y se suele tener varias tareas por cada *story*.

2.6. Criterio INVEST

Aplicar **INVEST** en las *User Stories* permite determinar la calidad de lo especificado en la misma. INVEST es un acrónimo, que se traduce de la siguiente forma:

2.6.1. Independent

Las *US* deberían ser independientes entre sí. Tener sólo un par de dependencias no trae mayores problemas, pero cuando muchas *stories* dependen de otras muchas *stories*, planificar y priorizar el *Sprint* se vuelve más complejo. Sin embargo, en la práctica tener *User Stories* totalmente independientes es casi imposible, o lograrlo no vale la pena el tiempo invertido. Éste punto hace referencia a desglosar todas las dependencias innecesarias de todas las *User Stories* del *Sprint*. Por ejemplo, unas *User Stories* podrían ser:

- Como comprador, quiero poder pagar con VISA para obtener mis entradas.
- Como comprador, quiero poder pagar con Discover para obtener mis entradas.
- Como comprador, quiero poder pagar con AMEX para obtener mis entradas.

Sin embargo, previamente debemos implementar el sistema de procesamiento pago con tarjetas de crédito. Podríamos decidir, de forma completamente arbitraria, anidar éste requerimiento a una de las cards anteriores. Ésto funciona pero atenta mucho contra la claridad de una *User Story*, además de que genera dependencia. Una mejor solución sería desglosar ese requerimiento en otra card “Como comprador, quiero poder pagar con tarjeta de crédito para poder más rápidamente pasar por caja”. Seguimos teniendo la dependencia, pero ganamos la ventaja de no tener requerimientos escondidos en ninguna card previa que agregan peso oculto.

2.6.2. Negotiable

La mayoría de las *US* deben ser negociables entre los stakeholders, el *PO* y el equipo de desarrollo, para así capturar qué funcionalidad se requiere (y porqué). Tener *US* no negociables sacrifica innovación y mejoras en los requerimientos, lo cual nos acerca más a un proyecto secuencial.

2.6.3. Valuable

Todas las *stories* deben tener valor para el usuario o para el cliente, sin excepción (caso contrario, no debería estar en el *Product Backlog*). En el caso de necesidades técnicas (como refactoring grande o migración de bases de datos), es más conveniente incluirlo como *task* dentro de la *story* apropiada. En caso de tener que comunicarlo con el cliente, se lo puede relacionar con la funcionalidad del valor para el mismo, y así ubicarlo en un contexto adecuado dentro de la conversación.

2.6.4. Estimable

Las *US* deben ser estimables por el equipo encargado de implementarlas. La estimación nos permite diferenciar cargas dentro de las *stories*. Éste conocimiento nos permite saber cuando una *story* debe ser refactorizada en otras más pequeñas o distribuida en otras ya existentes.

2.6.5. Sized Appropriately

A la hora de ubicar *stories* para un *Sprint*, éstas deben tener el tamaño apropiado (pequeño). Tener dos *stories* de 1 semana para un *Sprint* de 2 semanas magnifica mucho el riesgo de no terminar a tiempo. Dividiendo las *stories* en órden de días aseguramos un progreso más fluído y menor riesgo de errores.

2.6.6. Testable

La mayor parte de las *stories* deberían ser testeables bajo diferentes criterios, como tests de aceptación, tests de unidad, etc. Habiendo dicho ésto, hay *US* que no son testeables (por ejemplo *epics* o requerimientos no funcionales del sistema).

2.7. Requerimientos No Funcionales

Los requerimientos no funcionales representan ataduras a nivel del sistema. Podría ser algo como: “*Como Usuario quiero que el Sistema tenga 99.9999 % de uptime*”. Si bien tiene el formato de *US*, puede resultar difícil o imposible testearla. Lo mejor para éstos requerimientos es intentar agregarlos a la *definition of done* del equipo. De ésta forma, una *story* que agrega funcionalidad no podría considerarse terminada si tiene impacto negativo sobre el uptime del sistema (cumpliendo así el requerimiento).

2.8. KAS: Knowledge Acquisition Stories

Dentro de un producto innovador es probable tener que realizar tareas de exploración del dominio (como pueden ser investigar protocolos o arquitecturas) antes de trabajar sobre funcionalidad. Al realizar una *story* de éste tipo, el foco cae en la estimación y la relación costo-beneficio.

Supongamos que la estimación del trabajo de exploración nos cuesta \$10000. Entonces, tendría sentido si el costo de no explorar (es decir, realizar funcionalidad sin mucho conocimiento del dominio) es alto (por ejemplo, \$40000). Si el costo de no explorar y simplemente ir por un camino en base a la experiencia del equipo es bajo (por ejemplo, \$12000), la tarea de exploración pierde sentido.

2.9. Generación de Stories: User-Story Writing Workshop

Para obtener *user stories*, preguntarle a los usuarios del producto “Que quieren” no suele ser una buena idea. Muchas veces, al inicio del ciclo de desarrollo, ni ellos tienen claro exactamente todo lo que buscan. Es por eso que se creo un workshop con el objetivo de hacer *brainstorming* de placeholders para *user stories*, con su indicado *BV*.

En el workshop están todos presentes, desde el *PO*, stakeholders y Scrum Master, hasta el equipo de desarrollo. Pueden durar desde horas hasta días, y suelen dividirse en funcionalidad. Por ejemplo, es común comenzar con un workshop para definir los roles dentro del producto, que luego serán usados en las *US*. Los roles deberían ser más específicos que abstractos. “Usuario del Sistema” no es un gran rol en todos los casos, pero “*Lily*”, donde *Lily* representa un sector de la audiencia del producto (por ejemplo una chica de 7 a 9 que utilizará la aplicación), sí lo es.

No existe un consenso a la hora de definir el mejor criterio de armado para *stories*. Algunos lo hacen *top-down*, comenzando por *epics* y subdividiendo, mientras que otros lo hacen *bottom-up* (al revés).

2.10. Generación de Stories: *Story Mapping*

Story Mapping es una técnica para generar *stories* centrada en las actividades principales del usuario en el producto. Utiliza un desarrollo *top-down*, definiendo primero tareas de alto nivel, y dividiendo en sub-tareas prioritarias de distinto nivel. Veamos con un ejemplo como funcionaría.

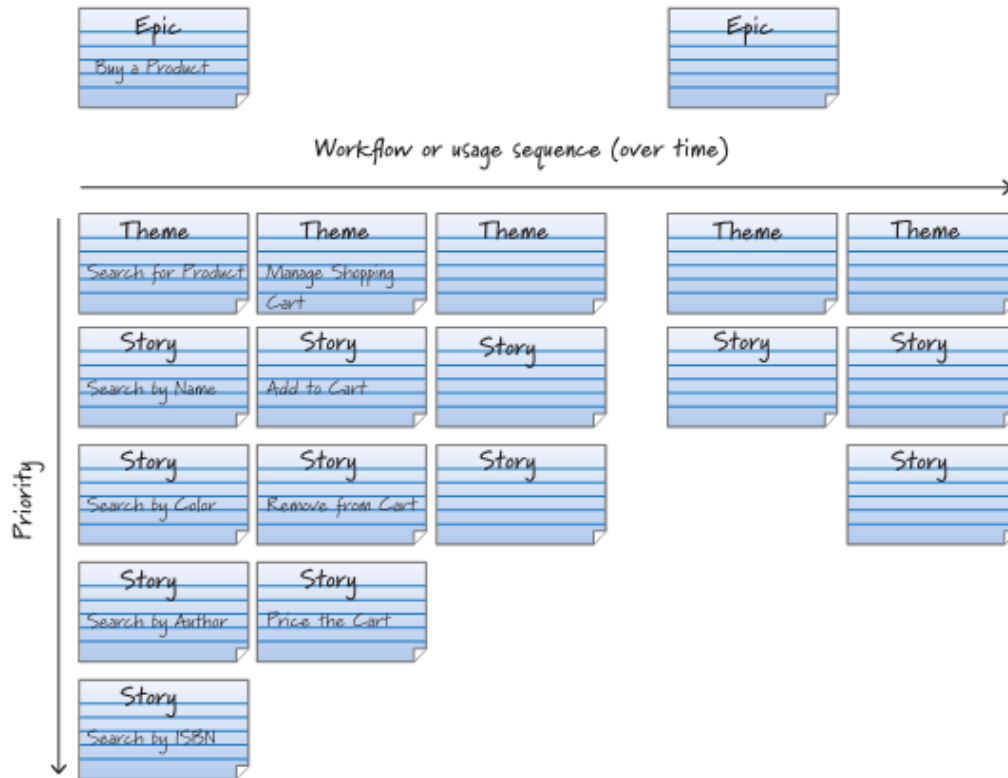


Figura 3: Ejemplo de *story map*.

Los niveles más altos representan tareas del usuario (como "Comprar un Producto"), los cuales se van dividiendo en sub-tareas llamadas *Themes* (como "Buscar el Producto"). A su vez, cada *Theme* contiene dentro varias *stories* de la misma categoría, las cuales podrían entrar en un *Sprint*. Las stories se ordenan por prioridad (arriba las más importantes). Un *story map* bien hecho muestra un flujo de acciones del usuario, dando contexto a las *user stories* y ayudándonos a entender su valor para el usuario final.

3. Product Backlog

El *Product Backlog* es una lista prioritaria de elementos conocidos como PBI's (*Product Backlog Items*). Ejemplos de éstos son funcionalidades a agregar, reparaciones sobre el producto, tareas de exploración, etc. Es la unidad central de trabajo en Scrum y debe ser visible a todos los miembros del equipo.

3.1. DEEP: Construyendo el Backlog

Como el criterio INVEST para *stories*, existe el criterio DEEP para *PB*, que nos ayuda a evaluar el estado de nuestra implementación. Veamos cada subcriterio por separado.

3.1.1. Detailed Appropriately

Los *PBI* sobre los que pensamos trabajar pronto deberían ser prioritarios (tope del *PB*) y lo suficientemente pequeños y detallados como para poder hacerse en un *Sprint*. Aquellos otros *PBI* más a futuro deberían ser más grandes y menos detallados, similar a una *epic*. A medida que se acerca la hora de implementarlos, se los refina, dándoles más detalle y subdividiéndolos en *stories* manejables en un *Sprint*.

3.1.2. Emergent

La estructura interna de nuestro *PB* debe ser emergente por naturaleza. Las necesidades de los clientes cambian, las reglas de negocio se alteran, todo implica un constante flujo de entrada de *PBI* y potencial reprioritización de los *PBI* actuales (generalmente no se mantienen fijos en el tiempo).

3.1.3. Estimated

Cada *PBI* debe estar estimado con un valor correspondiente al esfuerzo que toma realizar dicho ítem. Ésta estimación (generalmente en *story points* o días) es uno de los factores que determinará la prioridad del *Product Backlog Item* en el *PB*.

3.1.4. Prioritized

El nivel de priorización de cada ítem está relacionado con su cercanía a ser realizado. Si tenemos un producto y estructuramos nuestro trabajo en diferentes *Releases*, se esperaría que la mayoría de los *PBI* para el *Release* 1 estén priorizados con detalle.

Sin embargo, los *PBI* correspondientes a futuros *Releases* no deben estarlo. *Scrum* es un framework que adopta el cambio, por lo cual debemos ser conscientes que muchas de las condiciones para los futuros *Releases* probablemente cambien. En consecuencia, se alterarán, agregarán y quitarán los *PBI* correspondientes, por lo que habremos perdido tiempo priorizando algo que potencialmente ya no existe (ó no es cierto).

3.2. Grooming

El acto de *Grooming* se puede resumir en 3 puntos principales: Crear y refinar, estimar y priorizar *PBI*. Como dijimos antes, los cambios en las necesidades del cliente pueden llevar a nuevas priorizaciones para *PBI*, agregado o hasta eliminación de elementos existentes. La refinación ocurre en otros casos también, como cuando deseamos trasladar un ítem dentro de un *Sprint*.

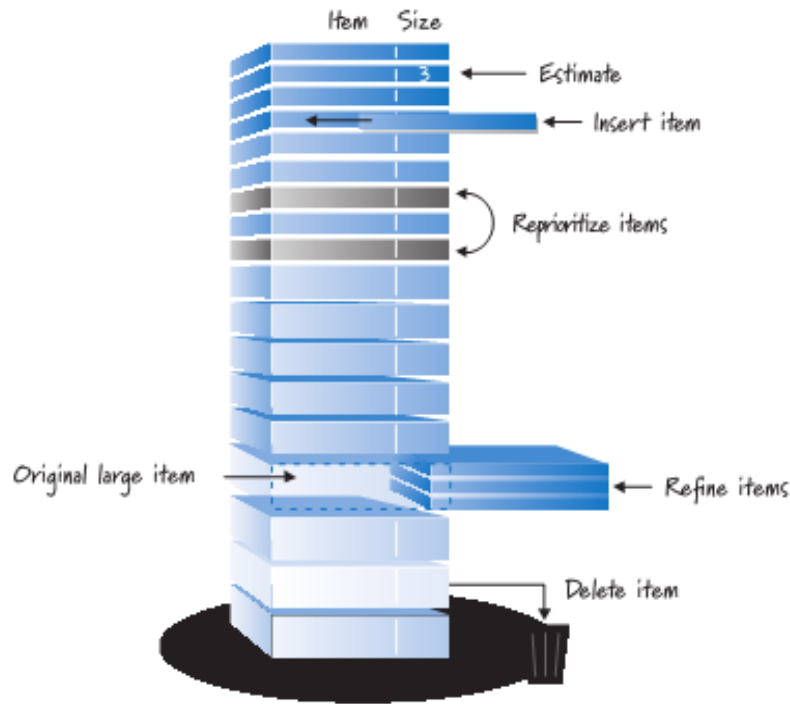


Figura 4: *Grooming* reforma el *Product Backlog*.

3.2.1. Quién Realiza el *Grooming*?

El proceso de *Grooming* tiene como pieza central al *Product Owner*. Sin embargo, es un proceso completamente colaborativo entre todos los miembros del equipo. Realizarlo de ésta forma asegura que todos posean un conocimiento en común sobre lo que se trabaja, minimizando el tiempo de errores de entendimiento o comunicación. Además, permite que cada miembro aporte su visión, ayudando a refinar o estimar de una forma que no era posible sin la misma.

En líneas generales, los miembros del equipo deberían dedicar entre 5 y 10 % de su tiempo de *Sprint* a ayudar al *PO* con las tareas de *Grooming*.

3.2.2. Cuándo se Realiza el *Grooming*?

La estructura de *Scrum* nos garantiza que debemos hacer *Grooming*, pero no especifica exactamente cuando. Aún así, hay ciertos momentos ideales para realizarlo, dependiendo de cada equipo.

- La mayoría realiza el *Grooming* en la *Sprint Review*. Al reunirse los stakeholders, el *PO* y el equipo, se obtiene validación sobre lo construido y se informa sobre la visión y funcionalidades deseadas para el futuro. Todo esto naturalmente ocasiona que nuevos *PBI* sean agregados (o removidos), como también genera una repriorización de *WIP* o *PBI* existentes.
- Además de lo primero, algunos equipos realizan *Grooming* una vez más por *Sprint* durante su ejecución. El concepto subyacente es que, a lo largo del *Sprint* se puede ganar más conocimiento sobre las tareas realizadas, o más entendimiento sobre las prioridades del cliente. Entonces una repriorización de los *PBI* ya existentes (para no cambiar el objetivo del *Sprint*) puede tomar lugar.
- La última opción es realizar el *Grooming* de manera incremental y de a partes, por ejemplo dividir y priorizar un *PBI* luego del *Daily Scrum*.

3.3. Definition of Ready

Realizar *Grooming* debería asegurarnos que los *PBI* al tope de nuestro *Backlog* están listos para ser trasladados dentro de un *Sprint* y completados adecuadamente. La formalización y generalización del estado “Listo para el Sprint” se conoce como **Definition of Ready**. Un ejemplo de definición podría ser:

- *BV* y *SP* correctamente definidos dentro de la *Story*.
- Criterios de aceptación establecidos y testeables.
- Subdivisión correcta en tareas terminada.

Se puede pensar en la *Definition of Ready* y la *Definition of Done* como dos estados por los que deben pasar todos los *PBI* antes de la **Sprint Review**.

3.4. Manejo de Flujo

Desarrollando un producto con total certeza sobre su futuro es algo casi imposible. El *Product Backlog* nos ayuda a organizar el constante flujo de información entrante de forma tal que podamos entregar resultados con alto valor para el cliente de forma frecuente. Veamos como se incorpora ésto a los distintos tipos de flujo.

3.4.1. Manejo de Flujo de Lanzamiento

Se suele practicar *Grooming* de tal forma que el *PB* soporte una estructura de múltiples niveles de lanzamiento del producto.

Para soportar esa estructura, es bueno dividir en *Backlog* en 3 capas. La primer capa (superior) contiene todas las *stories* que obligatoriamente estarán en el lanzamiento más cercano al producto. La segunda capa contiene *PBI* que intentaremos incluir, pero no son obligatorias (podemos no realizarlas y seguiremos teniendo un lanzamiento valioso). La tercer capa (inferior) contiene *PBI* de futuros lanzamientos, no importantes en el presente.

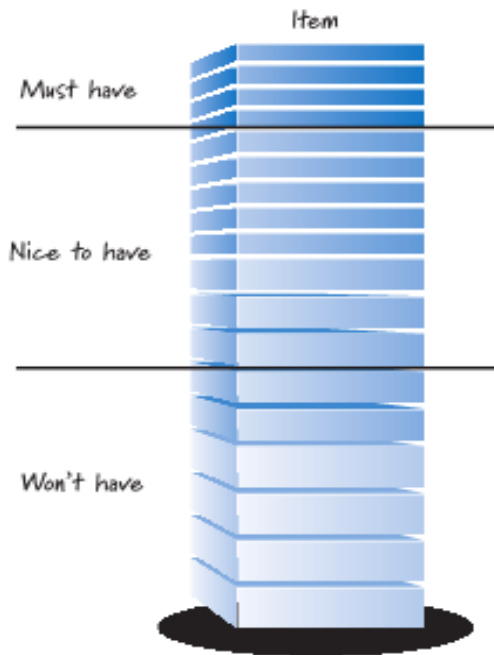


Figura 5: Múltiples capas dentro del *Product Backlog*.

3.4.2. Manejo de Flujo de *Sprint*

Para el manejo de *Sprint*, podemos ofrecer una visión diferente del *PB*, ésta vez como si fuera un pipeline de requerimientos con salida a un *Sprint*.

Cuando entran los *PBI* al *Backlog*, éstos se encuentran con poco nivel de detalle y son generalmente grandes. Al ir pasando por el pipeline, van adquiriendo el tamaño y nivel de detalle apropiado. Al llegar al final del pipeline, los ítems del *Backlog* pasan por el estado *Ready*, acorde con su definición, para terminar dentro de un *Sprint*.

El pipeline no se puede saturar o quedar vacío siempre que haya un balance entre los *PBI* que entran y los que salen. Como buena práctica, es conveniente tener suficientes *PBI* listos para un *Sprint* como para poder realizar 2 ó 3 *Sprints*. Ésto asegura que nunca tendremos el pipeline vacío, suponiendo una constante entrada de información.

3.5. Cuáles y Cuántos *Product Backlogs* Usar

En líneas generales, tener un *Product Backlog* por cada producto es una buena idea. Sin embargo, ésta regla no es siempre aplicable (por ejemplo si un producto es inmenso o no se sabe bien que constituye al producto).

3.5.1. Qué es un Producto?

No siempre queda claro que constituye al producto como un ente en sí. Es *Microsoft Word* un producto, o es una parte del producto *Microsoft Office*? Para éstos casos, es conveniente considerar al producto como un elemento de valor que un cliente estaría dispuesto a pagar y nosotros dispuestos a vender.

Usar ésta regla suele funcionar, excepto que tengamos *component teams*. Imaginemos que tenemos un GPS, y varios equipos designados a elaborar los sistemas internos del mismo. Es claro que el producto es el GPS, pero usar un único *PB* para el mismo podría resultar confuso. En éstos casos, conviene un *Product Backlog* por *component*, pero tratando de minimizar la cantidad de subdivisiones del producto (para no terminar con infinitos *backlogs*).

3.5.2. *Backlogs* Jerárquicos para Grandes Productos

En caso de productos más grandes, como por ejemplo un celular, es común tener múltiples equipos de trabajo. En general, varios equipos realizan funcionalidades en relación a un área del producto, como puede ser el reproductor audiovisual del celular. Una división de *Backlogs* es necesaria para mantener la información limpia y consistente entre equipos, por lo que se suele jerarquizar los diferentes *PB* utilizados.

Por ejemplo, para un producto con 11 áreas, se crean un total de 12 *Product Backlogs*. Cada área tiene su *backlog* específico, y hay uno central que describe y prioriza las funcionalidades a gran escala (nivel *epic*) del producto. Cada *epic* del *backlog* central se corresponde con decenas o cientos de *stories* dentro de un *backlog* de área.

3.5.3. Múltiples Productos para un Equipo

Si una empresa posee múltiples productos, tendrá múltiples *Product Backlogs*. La mejor manera de controlar ésta situación es asignar uno o más equipos a trabajar exclusivamente en un *PB*. Aún así, por razones particulares a la empresa puede ocurrir que un equipo esté asignado a más de un proyecto.

Teniendo eso en cuenta, el objetivo siempre debe ser minimizar la cantidad de equipos multi-proyecto dentro de la organización (en la medida de lo posible). Ésto es porque se pierde tiempo haciendo que el equipo realice un constante *context-switch* de mentalidad mientras completa tareas en un proyecto y otro. Ese tiempo perdido podría ser utilizado mejor en más tareas de un proyecto singular. Existen otras desventajas, como por ejemplo la necesidad de mantener un conocimiento extenso de múltiples dominios de diferentes problemas (lo cual termina no ocurriendo tan bien como debería).

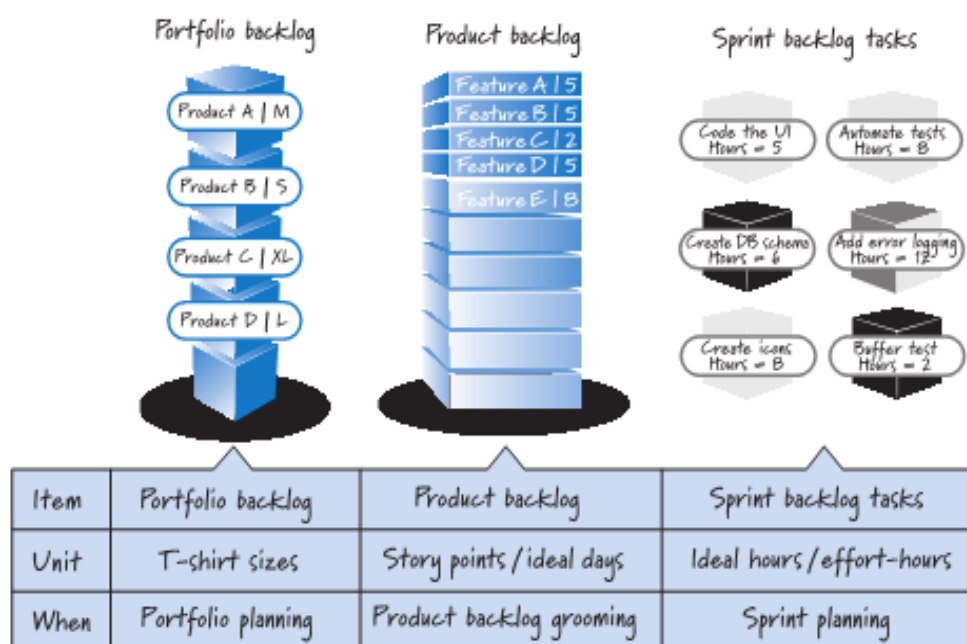
4. Estimación y Velocidad

Una vez que tenemos un *Release* planeado, tenemos que estimar aproximadamente cuánto nos va a llevar completarlo (para informar duración y costo). Para éste cálculo utilizamos la estimación del *Sprint* en *Story Points* o otra medida, dividido por la *velocidad* del equipo (es decir, cuantos *SP* puede completar por *Sprint*, número que se obtiene de promediar *Sprints* anteriores con un rango de velocidad). Ésta cuenta da como resultado la cantidad de *Sprints* necesarios para el *Release*.

Es vital notar que, si bien la estimación es importante, lo realmente crucial es la información ganada en las reuniones de estimación. Estimar **fuerza** al equipo a ganar más conocimiento de las tareas a realizar, de tal forma que promueve una adquisición iterativa de conocimiento sobre el producto y su estructura.

4.1. Qué y Cuándo Estimar

Se suelen hacer estimaciones diferentes en 3 áreas: una a nivel *macro* de la empresa (dentro de un *Backlog* de productos llamado *Portfolio*), a nivel stories dentro del *Backlog* de un producto, y a nivel horas para tareas.



4.1.1. Estimaciones para el *Portfolio*

Dentro de éste se hallan los productos a construir de la empresa. Lo más probable es que no tengamos un completo set de requerimientos en ésta etapa inicial de producción, sino que más tendremos una especificación poco clara del producto. Debido a ésto las estimaciones son de caracter bastante grueso. Una unidad comúnmente utilizada es la de tamaño de una remera para cada producto(desde *S* hasta *XL*).

4.1.2. Estimaciones para el *Product Backlog*

Una vez que un proyecto es aprobado y comienza, empezamos a adquirir más detalles sobre el dominio del mismo y las tareas a realizar. Por ende, debemos empezar a estimar de forma diferente. La estimación de los *PBI* es parte de la actividad de *Grooming*, y se suele usar *story points* o *días ideales* como medidas para estimar.

En general la primera reunión de estimación coincide con la planificación inicial para el *Sprint*, pero

puede haber posteriores reuniones si hacen falta (alguna *story* puede requerir más exploración o validación con el *PO*).

4.1.3. Estimaciones para las Tareas de un *PBI*

Esta actividad suele ocurrir en la planificación del *Sprint*, para asegurarse que el compromiso de funcionalidades a entregar es razonable. Las tareas se miden en *ideal hours* (también llamado *man-hours* ó *person-hours*). Una *man-hour* no es una hora de tiempo, sino que se refiere a la cantidad de trabajo completado por un trabajador promedio de la empresa en una hora de tiempo.

4.2. Conceptos de Estimación para *PBI*

4.2.1. Estimar como Equipo

En *frameworks* más tradicionales de proyecto, suele ocurrir que el manager del mismo realiza las estimaciones iniciales para que luego el equipo comente sobre ellas. En *Scrum*, la regla es simple: El equipo de desarrollo, aquellos encargados de diseñar, construir y testear *PBI*, proveen las estimaciones. Todos deben participar, pues cada uno puede aportar parte de su visión del *PBI* para reducir errores al estimarlo.

El rol del *PO* es estar presente en la estimación para describir o clarificar ciertos *PBI*, pero no debe guiar o limitar la estimación del equipo. El *ScrumMaster* también debe estar presente, para guiar y facilitar el trabajo de estimar los *PBI*.

4.2.2. La Estimación no es un Compromiso

Las estimaciones deben representar una medida realista de que tan grande es el *PBI*. Tomarlo como un compromiso solo sirve para añadir un overhead de error que comienza a inflar artificialmente el tiempo del ítem.

Al pensarlo como compromiso, se ve influenciado por constantes externas. Un desarrollador no estima la misma cantidad si se le informa que de su estimación debe ser obligatoriamente precisa, pues de ella depende su bonus salarial de fin de año. Tenderá a inflar la estimación para estar seguro de llegar a tiempo de forma más precisa. Sin embargo, el *PBI* es el mismo (y éste es sólo un factor externo).

4.2.3. Exactitud vs Precisión

Las estimaciones deberían ser precisas sin llegar a ser exactas. Generar números de estimación completamente exactos suele ser una pérdida de recursos, pues llegar al número exacto lleva tiempo potencialmente considerable. Además, si tenemos confianza de que nuestra estimación es exacta y luego tomamos decisiones de negocio acotadas a ese valor (como transmitirle al cliente un precio fijo sin un *buffer* de error), hay buenas posibilidades de equivocarse por la naturaleza cambiante del entorno de negocio.

4.2.4. Utilizar Tamaños Relativos

A la hora de estimar *PBI*, es una buena práctica usar tamaños relativos con respecto a otros, pues nos da una mejor idea de cuánto nos llevaría el total de los *PBI* que si usáramos medidas absolutas para todos. Por ejemplo, si mi *PBI 1* es 5 veces más grande que el *PBI* de menor tamaño (1 *sp*), sabemos inmediatamente que le corresponden 5 *story points*. En la práctica, las estimaciones relativas suelen ser mucho más precisas (pues siempre nos movemos sobre una sola medida absoluta correspondiente al *PBI* de menor tamaño).

4.3. Unidades de Estimación para *PBI*

4.3.1. Story Points

Representan la magnitud de un *PBI* y suelen estar influenciados por complejidad y tamaño “físico” de la tarea. Construir un algoritmo importante de negocio no es algo físicamente grande, pero puede ser muy complejo. Actualizar cada celda de un excel mantenido desde el 2001 no es complejo, pero si es una tarea “físicamente” grande (si no se puede automatizar). Es la unidad de medida ideal para tamaños relativos, como vimos previamente en un ejemplo.

4.3.2. Ideal Days

Unidad **alternativa** de estimación que representa la cantidad de *person-days* necesarios para completar una *story* (recordar que no es lo mismo que un día real). El mayor problema con usar ésta medida cae en errores de interpretación. Todas las personas involucradas en el proyecto deben tener bien en claro que un *PBI* con una estimación de 2 *ideal days* no se termina de Martes a Miércoles (inclusive). Sin embargo, a veces éste no es el caso, y el valor de las estimaciones con ésta medida termina perdiéndose (pues genera más confusión que lo que aporta).

4.4. Planning Poker

Planning Poker es una técnica de estimación basada en el consenso general. Para realizarla, el equipo debe decidir la escala de números a utilizar para la estimación. Se suele usar una escala con saltos, pues no queremos ser exactos (por ejemplo, la secuencia de Fibonacci). Una vez decidida la escala, agrupamos diferentes *PBI* según su tamaño acorde a la misma. Es importante considerar, dado un *PBI* a estimar, el resto de los elementos que fueron estimados con el mismo número.

4.4.1. Cómo Jugar

El foco del Planning Poker es la discusión, que se debe incentivar constantemente, y el consenso. Cada miembro del equipo tendrá 1 carta por cada valor posible de estimación para los *PBI*. Las reglas son:

1. El *PO* selecciona 1 *PBI* y lo lee. El equipo de desarrollo discute el ítem, realizando preguntas sobre el mismo que el *PO* aclara.
2. Cada miembro selecciona 1 carta de estimación para ese *PBI*, sin mostrarla. Una vez que todos eligieron, se muestran todas las cartas juntas.
3. Si todos eligieron la misma carta, hay consenso, y se termina la estimación del *PBI*.
4. Si hay diferentes elecciones, los miembros del equipo exponen sus argumentos de estimación (hace notar diferentes asunciones y errores de entendimiento). Se suele arrancar por el valor mas bajo y alto de estimación, acercándose al centro. Luego de ésto, se vuelve al paso 2.

4.4.2. Beneficios

- La estimación final suele ser mucho más precisa que cualquier estimación individual.
- Se gana información invaluable del dominio para todo el grupo en las discusiones de estimación, lo cual debería llevar a mejor trabajo futuro sobre el producto (éste es el punto más importante del Poker).

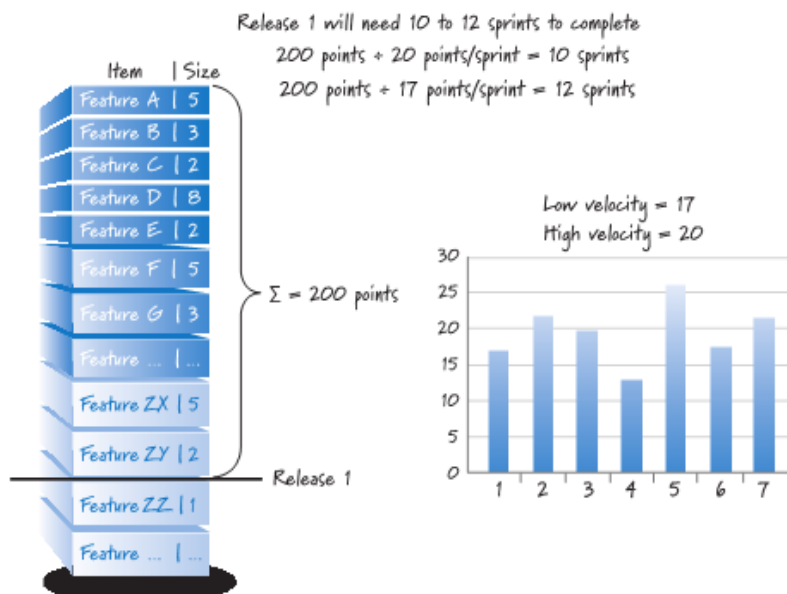
4.5. Velocidad

Representa la cantidad de trabajo completado en un Sprint, obtenido de sumar todos los tamaños de los **PBI completados** al final del mismo. Es una medida para el output del equipo, no para el resultado (el valor de negocio de lo que se entregó), pues un *PBI* de tamaño 8 no es necesariamente más valioso en términos de *BV* que uno de tamaño 3.

La velocidad tiene un doble propósito. Por un lado, es una herramienta invaluable a la hora de planificar diferentes *Releases* en cantidad de *Sprints* (es cuestión de dividir la totalidad de puntos de los *PBI* del *Sprint* por la velocidad). Además, funciona como métrica de diagnóstico para el equipo. Al observar la variación de la velocidad en el tiempo, se pueden encontrar fallas o mejoras en el proceso de *Scrum* de la empresa y corregirlas para entregar más valor para el cliente en cada *Sprint*.

4.5.1. Calculando un Rango de Velocidad

Siguiendo nuestro objetivo de “precisión, no exactitud”, utilizaremos un rango de posibles velocidades a la hora de estimar dentro de la planificación. La variación dentro del rango depende de la etapa en la que nos encontremos dentro del producto (en una etapa inicial sin información y conocimiento del dominio se espera más variación).



En la imagen podemos ver que para el primer *Release*, en vez de declarar exactamente cuando entregaremos toda la funcionalidad (lo cual posiblemente sería una adivinación), utilizamos un rango como respuesta. Para éste cálculo se utilizaron dos valores de velocidad fijos, pero en realidad uno suele fabricar un valor menor y otro mayor de estadísticas sobre velocidades del equipo en previos *Sprints* (menor y mayor promedio, intervalos de confianza 90 %, etc).

4.5.2. Prediciendo Velocidad

Si bien es sencillo medir velocidad si se dispone de datos históricos previos, habrá que caer en la predicción a la hora de definir velocidad dentro de equipos recién formados. Una manera muy común de predecir es planificar los primeros dos *Sprints* lo más preciso posible y utilizar los dos valores de velocidad obtenidos de la planificación como mayor y menor. También se puede tener en cuenta la velocidad de los miembros en equipos de trabajo anteriores (si los hubo). De cualquier forma, es notoria la baja calidad de la predicción en comparación al valor estadístico, por lo que se debe utilizar un valor histórico apenas se tenga una muestra disponible.

4.5.3. Afectando la Velocidad

El comportamiento de la velocidad para un equipo debería variar a lo largo del tiempo. En concreto, se espera un crecimiento luego de los primeros *Sprints*, por la amplia adquisición de conocimiento que ocurre en etapas de inepción del proyecto. Una vez que el equipo posee un buen conocimiento del dominio, suponiendo que hagan un monitoreo de su proceso de *Scrum* en retrospectivas aplicando mejoras, se espera que la velocidad crezca lentamente hasta llegar a un pico.

Llegada la cima, se puede mejorar la velocidad dándole al equipo más herramientas y capacitación. Ésto mejora la velocidad a futuro, con el costo de bajar la velocidad actual (hasta que el equipo termina de absorber los cambios). Ciertas composiciones de personas en un equipo pueden tender a mayor velocidad, pero en general no es recomendable sacar y poner personas en diferentes equipos frecuentemente, pues ésto baja considerablemente su velocidad.

La última herramienta consiste en hacer *overtime*. Trabajar más horas suele mejorar la velocidad actual, si necesitamos con urgencia entregar ciertos *PBI*. El problema es que conlleva una recaída posterior mientras el equipo se recupera, la cual termina costando más que lo previamente ganado.

4.5.4. La Velocidad Mal Utilizada

La velocidad no debe ser utilizada como una medida de performance de todos los equipos. Supongamos que, en una empresa, se dará el bono más grande al equipo con mayor velocidad. Es extremadamente probable que los diferentes equipos usen distintas medidas para estimar sus *PBI*, por lo que la idea fracasa de base, pero supongamos que la medida se implementa igual. Ahora los equipos potencialmente cambiarán su manera de medir los *PBI* para poder tener mayor velocidad, lo cual desemboca en un efecto denominado *inflación de puntos*.

Supongamos que, milagrosamente, todos los equipos miden los *PBI* de la misma forma. Existe la posibilidad de que ahora los equipos empiecen a preocuparse por terminar cada *PBI* en vez de **completarlo**, bajando **fuertemente** la calidad del producto entregado. En general, utilizado como métrica macro-grupal tiene consecuencias mayormente negativas, y debería como medida para ayudar internamente a cada equipo.

5. Deuda Técnica

El concepto de deuda técnica se refiere a los diferentes atajos y medidas que tomamos para entregar software excesivamente rápido y las consecuencias en el sistema (no todas son evitables). Algunas de ellas son:

- **Mal diseño:** aquel que alguna vez tuvo sentido pero ahora, debido a cambios en el negocio, es obsoleto.
- **Defectos:** problemas del software que aún no han sido removidos.
- **Mala cobertura de tests:** áreas que debreían estar testeadas y no lo están.
- **Excesivo testeo manual:** testeo hecho a mano que debería estar automatizado.
- **Falta de experiencia en la plataforma:** tener aplicaciones principales escritas en un lenguaje que ninguno de los programadores domina.

El excesivo testeo manual es una categoría de deuda conocida como deuda *ingenua*, pues es frecuentemente accidental y fruto de irresponsabilidad (ya que es solucionable con entrenamiento de personal y buenas decisiones de negocio). También está la deuda *inevitable*, como lo puede ser el diseño. Nuestro entendimiento de que es un buen diseño avanza con el tiempo, nuestro entendimiento del producto y experiencia, por lo que decisiones de diseño e implementación en etapas tempranas pueden terminar siendo refactorizadas. Ésto simboliza una deuda, pero inevitable, pues a veces es necesaria para un buen diseño.

El último tipo de deuda es la *estratégica*, la cual representa una herramienta que ayuda a las organizaciones a comprender las consecuencias de importantes decisiones de negocio. Por ejemplo, una empresa cuyos fondos son muy limitados puede tomar la decisión de sacar un producto al mercado antes de tiempo, lo cual genera deuda técnica pero tal vez sea la única opción para seguir funcionando.

5.1. Consecuencias de la Deuda Técnica

5.1.1. Punto de Inflexión Impredecible

La deuda no tiene un tamaño fijo y crece de forma no lineal. Cada pequeña adición a la misma contribuye a su total, hasta que llega un punto en el cual se vuelve inmanejable y el producto entra en caos. No se necesita una deuda enorme para llegar a ese punto, pues pequeñas deudas puede tener consecuencias no acordes a su tamaño.

5.1.2. Tiempo de Entrega en Alza

Caer en deuda implica obtener un préstamo de tiempo hoy y pagarlo en el futuro, con intereses dependiendo la deuda. A mayor deuda, menor *velocidad* del equipo. A menor velocidad, más tiempo ocurre entre cada entrega del producto iterativo. En puntos críticos, la velocidad es tan baja y el costo tan alto que el producto empieza a atrofiarse, perdiendo espacio para nuevas funcionalidades.

5.1.3. Número Significativo de Defectos

Una deuda significativa complejiza el producto, produciendo fallos críticos con mayor frecuencia. Ésto trae consigo un overhead de tiempo creciente cada vez más grande por implementar funcionalidades en una base repleta de errores que sólo terminan por componer a los mismos.

5.1.4. Costos de Desarrollo y Soporte Creciente

Con una gran deuda técnica, pequeños cambios en el software de repente se vuelven grandes, y grandes cambios se vuelven inmensos. Cuando la deuda llega a su punto de inflexión, el overhead es tan alto que nuestro producto pierde capacidad de adaptarse al entorno cambiante, pues la mayoría de los cambios son simplemente demasiado caros.

5.1.5. Consecuencias Humanas

Un producto con alta deuda técnica suele ser molesto para trabajar, por tener que lidiar con todos los errores y atajos tomados previamente. Perder tanto tiempo con infinitos pormenores de bajo interés acaba con la buena emoción que puede haber detrás del desarrollo del producto. La gente acaba por “quemarse” y comienza a buscar alternativas, tal vez en otros lugares con proyectos mejor mantenidos.

También se ve afectada la gente de negocios. Debido al incremental costo y duración asociado a una gran deuda, la satisfacción de los clientes baja (y su frustración sube). El producto se vuelve poco atractivo, perdiendo potenciales ventas y futuros ingresos.

5.2. Causas de la Deuda Técnica

5.2.1. Presión para Alcanzar Deadlines

Causa primaria de la deuda ingenua y estratégica. Con la planificación terminada, sabemos que necesitamos mantener una cierta velocidad para terminar las funcionalidades adecuadas en la fecha estimada. Si al comenzar el trabajo observamos una menor velocidad y el deadline junto con sus funcionalidades son innegociables, se cae en deuda técnica al intentar terminar las mismas bajando la calidad usual obtenida de trabajar a la velocidad estimada (por ejemplo, perdiendo cobertura de tests o realizando un peor diseño).

5.2.2. Mito: Menor Testeo Incrementa Velocidad

Un dicho muy común es que el tiempo ahorrado por el testeo incrementa la velocidad del equipo. Si bien eso puede ser cierto para la velocidad dentro de 1 *Sprint*, en realidad lo único que logra es adquirir una deuda exponencialmente creciente, reduciendo cada vez más la velocidad del equipo iteración a iteración. Ésto ocurre porque los errores pasados son detectados en el futuro, donde corregirlos toma mucho más tiempo.

El equipo ideal integra la fase de testing con prácticas como **TDD**, lo cual reduce el tiempo de testeo y permite entregar funcionalidad con robusta cobertura frente a errores.

5.2.3. Deuda Construída sobre Deuda

Una vez adquirida la deuda, se debe tener en cuenta y pagarla antes de perder el control. Si se obtiene una deuda en el *Release* 1, es esperable que la estimación de velocidad para el *Release* 2 vuelva a fallar (por el overhead de la deuda). Una solución es construir más deuda encima de la existente, para poder entregar la funcionalidad prometida.

Si éste patrón continúa, se llega a un punto en el cual la velocidad *efectiva* del equipo resulta muy cercana a 0. Éste estado representa un producto en el cual no podemos realizar casi ningún cambio, pues una pequeña alteración en un área podría quebrar las otras 9 del software (y no lo sabremos, pues no hay tests). Ésta situación nos indica claramente que la deuda técnica debe manejarse con cuidado, pero sobre todo, debe manejarse.

5.3. Como Controlar la Deuda Técnica

Obtener un producto sin deuda alguna no suele ser económicamente viable. Sin embargo, si lo es mantener al mismo con una deuda constantemente controlada. Veamos algunas prácticas que hacen que esto sea posible y tenga sentido.

5.3.1. Buenas Prácticas Técnicas

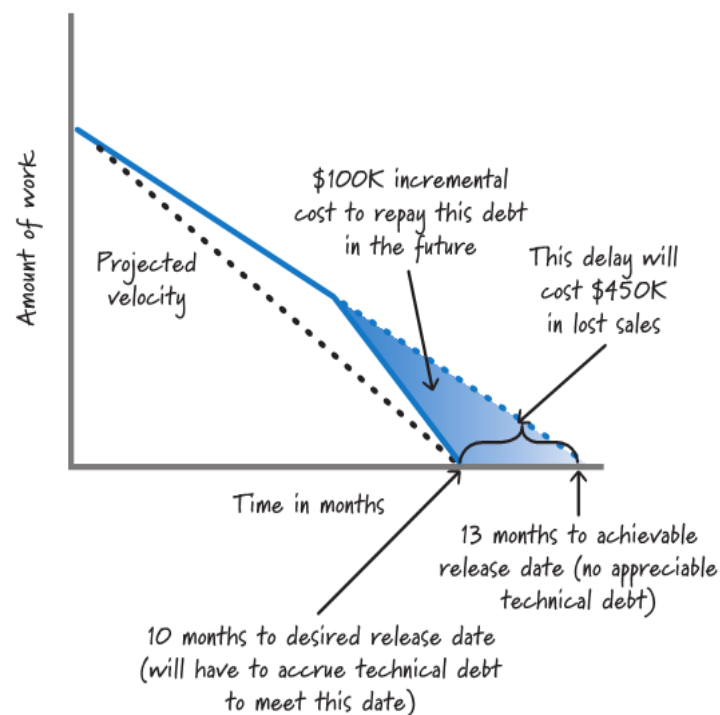
Un primer acercamiento al control de la deuda requiere detener la adquisición constante de deuda ingenua. En esta área, buenas prácticas de desarrollo ayudan a evitar errores que deben pagarse en el futuro. Si bien *Scrum* no las define, suelen usarse prácticas como *TDD*, Integración Continua, Refactoring, Testeo Automático, etc. Para la deuda previamente acumulada, realizar un refactor del código para mejorarlo (por ejemplo haciéndolo escalable o mejorando su performance) ayuda a disiparla.

5.3.2. Definition of Done

Una causa constante de deuda reside en tener que realizar trabajo que debería haber sido realizado dentro de una funcionalidad pasada. Tener una *definition of done* que requiera aptitudes técnicas (como cobertura de tests o criterios de aceptación) nos asegura que terminaremos el *Sprint* con una deuda sumamente menor a que si no la tuviésemos, pues quedan garantizados requisitos mínimos para cada funcionalidad.

5.3.3. Entendimiento de la Economía detrás de la Deuda Técnica

La deuda técnica puede ser un recurso que se puede utilizar a nuestro favor, pero para ello se debe entender el costo económico detrás de cada decisión. Supongamos que tenemos una situación donde cada mes de desarrollo cuesta \$100000, no podemos entregar toda la funcionalidad para la fecha estimada, y todas las funcionalidades deben ser entregadas.



Esto trae consigo dos alternativas:

- Atrasar la entrega 3 meses hasta poder finalizar correctamente todas las funcionalidades prometidas, aumentando el costo total de desarrollo y perdiendo mucho dinero en ventas (pues el producto se lanza más tarde, perdiéndose temporadas clave).
- Acelerar el desarrollo tomando atajos y entregar todas las funcionalidades en la fecha prometida, adquiriendo de esta forma deuda técnica. Para analizar la viabilidad de esta opción, se debe hacer

un análisis del costo de contraer la deuda. Supongamos que se le pregunta al equipo de desarrollo, y ellos responden que necesitarán 4 meses para dejar el sistema limpio de los atajos tomados. Es un mes extra comparado a la opción anterior (\$100000 extra), pero se gana todo el dinero de ventas (\$450000), haciendo ésta opción tentadora. Sin embargo, en general se escapan varios puntos al hacer un análisis de deuda, los cuales transforman ésta opción en una inviable, como por ejemplo:

1. Cuál es el costo del delay a nivel lanzamiento del producto ocasionado por tener que reparar la deuda técnica?
2. Cuál es el costo compuesto de seguir aumentando la deuda en el futuro? (en el caso de que se vuelva a caer en el ciclo de preferir funcionalidades sobre reparaciones, lo cual suele suceder).

En general, si luego de un buen análisis las condiciones económicas son favorables, se puede caer en deuda, pero la realidad es que el potencial alcance de la deuda suele ser subestimado, haciendo el análisis previo poco realista.

5.4. Visibilidad de la Deuda Técnica

5.4.1. Visibilidad a Nivel Negocio

Una manera de proveer visibilidad de la deuda a nivel negocio es monitoreando y mostrando la velocidad a lo largo del tiempo, pues un aumento en la deuda suele traducirse en reducción de velocidad, lo que termina desembocando en consecuencias económicas fácilmente entendibles por ellos. En líneas generales, traducir el costo técnico a un costo económico es una buena manera de mostrar visibilidad dentro del ámbito de negocios.

5.4.2. Visibilidad a Nivel Técnico

Si bien la gente del ámbito técnico suele tener conocimiento tácito de la deuda, ese entendimiento puede no ser lo suficientemente visible como para poder analizarlo y arreglarlo. Existen varias maneras de hacer la deuda visible, como por ejemplo:

1. Logear deudas como *bugs* dentro del sistema de *tracking*, tageando las deudas de forma unívoca.
2. Generando un *PBI* por cada deuda, dentro de un Backlog especial para la deuda técnica. Durante la planificación del *Sprint*, el equipo de desarrollo junto con el *PO* revisan si las tareas a realizar en el mismo se entrecruzan con cards del Backlog de Deuda. Si lo hacen y éstas cards son lo suficientemente pequeñas, se añaden al *Sprint* y se completan cuando se trabaje sobre el área del *PBI*.

5.5. Pagando la Deuda Técnica

5.5.1. No Toda Deuda Debería Ser Pagada

- No vale la pena invertir una cantidad sustancial de tiempo pagando la deuda técnica de un producto cerca del fin de su vida, pues ese dinero sería mejor invertido en productos de mayor valor (o en caso de que no exista, un nuevo desarrollo). Lo mismo ocurre para productos con un ciclo de vida predefinido y corto, pues su valor suele estar en salir rápido al mercado y ganar una enorme cantidad de dinero inicial, que en un incremento largo y gradual de ingresos y calidad del producto (por lo que solucionar la deuda técnica sería dinero perdido).
- Para los prototipos creados para una rápida demostración o adquisición de conocimiento sobre un dominio, puede no tener sentido pagar toda la deuda técnica acumulada. El valor de los prototipos suele ser en el negocio detrás de ellos (por ejemplo si el mismo era un *MVP* para un contrato con una empresa) o el conocimiento ganado, no tanto el código.

5.5.2. Boy Scout Rule: Pagar la Deuda Cuando se Encuentre

Siempre que se agrega código o diseño a un producto se lo intenta dejar en un mejor estado que el anterior. Ésto quiere decir que, si un desarrollador se encuentra con un problema mientras trabaja (conocido como **happened-upon debt**), éste lo debería arreglar en el acto (claro que ésto se aplica hasta un cierto tamaño de deuda, pues no debe interferir con el objetivo del *Sprint*). Ésto debería tomarse en cuenta a la hora de estimar los *PBI*, por ejemplo agregando un overhead considerando la potencial deuda que será encontrada y cuanto de ella quiere pagarse.

5.5.3. Pagar la Deuda de Forma Incremental (en cada *Sprint*)

A la hora de trabajar sobre un producto con una alta cantidad de deuda, es mejor pagarla de forma constante e incremental en vez de un pago enorme mucho tiempo después, para poder seguir entregando funcionalidad interesante para los stakeholders. En cada *Sprint* se debe entregar trabajo que tenga valor para el cliente, por lo que realizar cosas como “Sprints de Deuda” atentan contra los principios de *Scrum*. Sólo se deberían realizar en los casos donde la deuda sea tan alta que realizar funcionalidades sea económicamente inviable (y en ese caso preguntar porque se permitió que la deuda llegue hasta este punto sin pagarse).

5.5.4. Pagar la Deuda más Importante Primero

No todas las deudas técnicas son iguales, y siempre se debe intentar comenzar a pagar la que más importa. Por ejemplo, podemos tener deuda en un módulo central que usamos a diario y necesita un refactor. Pagamos el interés de esa deuda diariamente, y cada día es más complicado modificar el módulo. Ésta deuda es más interesante que un par de *bugs* sobre una sección del producto que no es muy usada, por ejemplo (a no ser que dicha sección sea crítica, claro está).

Es común que la deuda más importante sea también la más grande en el producto, por lo que un *approach* iterativo e incremental a su resolución es esencial.

5.5.5. Pagar la Deuda y Entregar Trabajo de Valor

Una excelente manera de hacer todo lo anterior es pagar la deuda mientras se entrega funcionalidad valiosa para el cliente. Por lo tanto, nuestro trabajo sigue un esquema bien definido por cada *PBI* trabajado:

- Nos comprometemos a hacer un trabajo de calidad, para no añadir deuda *ingenua*.
- Aplicamos la regla del Boy Scout, limpiando una cantidad razonable de *happened-upon debt*.
- Pagamos la deuda técnica asociada al área de trabajo del *PBI* de forma incremental. Por ejemplo, realizando la card del Backlog de Deuda asociada al área del *PBI*.

Seguir éste flujo de trabajo trae numerosas ventajas:

- Alinea la tarea de reducción de deuda con tareas valiosas para el cliente que pueden ser priorizadas por el *Product Owner*.
- Establece el mensaje que todos los miembros del equipo son responsables del limpiado y mantenimiento de la deuda técnica.
- Refuerza prácticas de eliminación de deuda (pues se hacen en cada *PBI*), mejorando la calidad del código.
- Evita pagar deuda técnica en áreas de poco valor o productos innecesarios.

6. Product Owner

El *PO* es el punto central de liderazgo dentro de un producto, mirando constantemente en 2 direcciones. Por un lado, debe entender y representar las necesidades de los stakeholders y usuarios, sirviendo como manager del producto. Por el otro, debe comunicarle al equipo de desarrollo que construir y en que orden hacerlo, verificando que el trabajo realizado satisfaga el nivel de calidad acordado para el proyecto. A veces, la cantidad de trabajo es tanta que un *PO* no puede representar los intereses de ambos frentes, necesitando la formación de un equipo para ello (ésto será explicado más adelante).

6.1. Responsabilidades Principales

6.1.1. Manejo de la Economía

- A nivel *Release* el *PO* hace continuos *trade-offs* de funcionalidad, scope, y fecha de lanzamiento a medida que entra información económicamente sensible. Por ejemplo, si añadir una funcionalidad retrasa el *Release* un 4 %, pero aumenta los ingresos un 20 %, es trabajo del *PO* decidir si agregarla o no. También se encargará de decisiones de inversión, como por ejemplo decidir si otorgar fondos para un segundo *Release* o terminar el desarrollo luego del primero (pues tal vez el producto está completo o el mercado cambio y se debe alterar su dirección).
- A nivel *Sprint*, se encarga de asegurarse que en cada uno de ellos haya un buen retorno de inversión (es decir, se entreguen suficientes funcionalidades de valor para los stakeholders y los usuarios del producto). Además, es el encargado de priorizar los *PBI* a realizar en cada *Sprint*, trasladando las inquietudes y necesidades de los stakeholders al *PB* del equipo de desarrollo.

6.1.2. Participación en la Planificación

El *PO* es un jugador central en todos los niveles de planificación. A nivel *Portfolio*, se encarga de determinar la posición del producto en el *Backlog* para determinar cuando empezará y terminará su desarrollo. A nivel producto, el *PO* trabaja junto a los stakeholders para determinar la visión y composición del mismo. A nivel *Release* y *Sprint*, el *PO* trabaja con los stakeholders y el equipo de desarrollo para determinar que funcionalidades implementar y cuando llevar a producción los cambios realizados.

6.1.3. Grooming

Dentro de la actividad de *Grooming*, el *PO* asiste en la creación, refinación y clarificación de diferentes *PBI* para el Backlog, al transmitir su visión de producto al equipo de desarrollo (aunque no se encarga de estimarlos).

6.1.4. Definir Criterios de Aceptación (y asegurarse que se cumplan)

El *PO* es el responsable de definir criterios de aceptación (frecuentemente acompañados de tests) para determinar cuando un *PBI* se encuentra terminado. Por ésta razón, muchos usuarios de *Scrum* incluyen la existencia de criterios de aceptación como un ítem a cumplir dentro de la *Definition of Done* del equipo. También debe asegurarse que se cumplen, verificando que los criterios de aceptación sean alcanzados antes de la finalización del *PBI* (y que sus tests asociados pasen).

Es importante que ésta verificación ocurra antes de la *Sprint Review*, de forma tal que el equipo de desarrollo tenga tiempo de arreglar errores o malentendidos que hayan surgido en la planificación.

6.1.5. Colaborar con el Equipo de Desarrollo

Una tarea fundamental de todo *PO* es su colaboración constante con el equipo de desarrollo. Éste frecuente flujo de información minimiza el impacto de errores en el producto, pues la validación se obtiene

mucho más rápido que con otra metodología. Como regla, poder validar dudas con el *PO* o alguien que represente sus intereses 1 vez por *Sprint* (además de la *Review*) suele ser suficiente. Éste punto es uno que frecuentemente suele fallar, pues el *PO* asume que su trato bajo *Scrum* debería ser igual a otros *frameworks* más tradicionales.

En un desarrollo *Waterfall*, suele haber alta participación al principio (para entregar el documento de Requerimientos) y al final, para la validación. El problema es que suele ser demasiado tarde y costoso encontrar errores en la etapa final, que podrían haber sido capturados y revertidos fácilmente si el *PO* mantuviese la comunicación a lo largo de la ejecución del desarrollo.

6.2. Características y Habilidades

6.2.1. Conocimiento del Dominio

Un buen *PO* tiene un alto conocimiento del dominio del producto y es capaz de transmitir una visión del mismo al equipo de desarrollo, guiando y priorizando las actividades a realizar. Ésto no implica un conocimiento total del producto, mas bien una certeza de la naturaleza cambiante dentro entorno del mismo.

6.2.2. Habilidades Sociales

El *PO* frecuentemente es “la voz del stakeholder”, por lo que una buena relación con los mismos es esencial. Los stakeholders pueden tener necesidades conflictivas entre sí dentro del producto, y es trabajo del *PO* poder llegar a un consenso entre las mismas mediante la negociación. Luego del consenso, él es el encargado de transmitir las inquietudes al equipo de *Scrum* (y actuar de nexo entre ambos bandos).

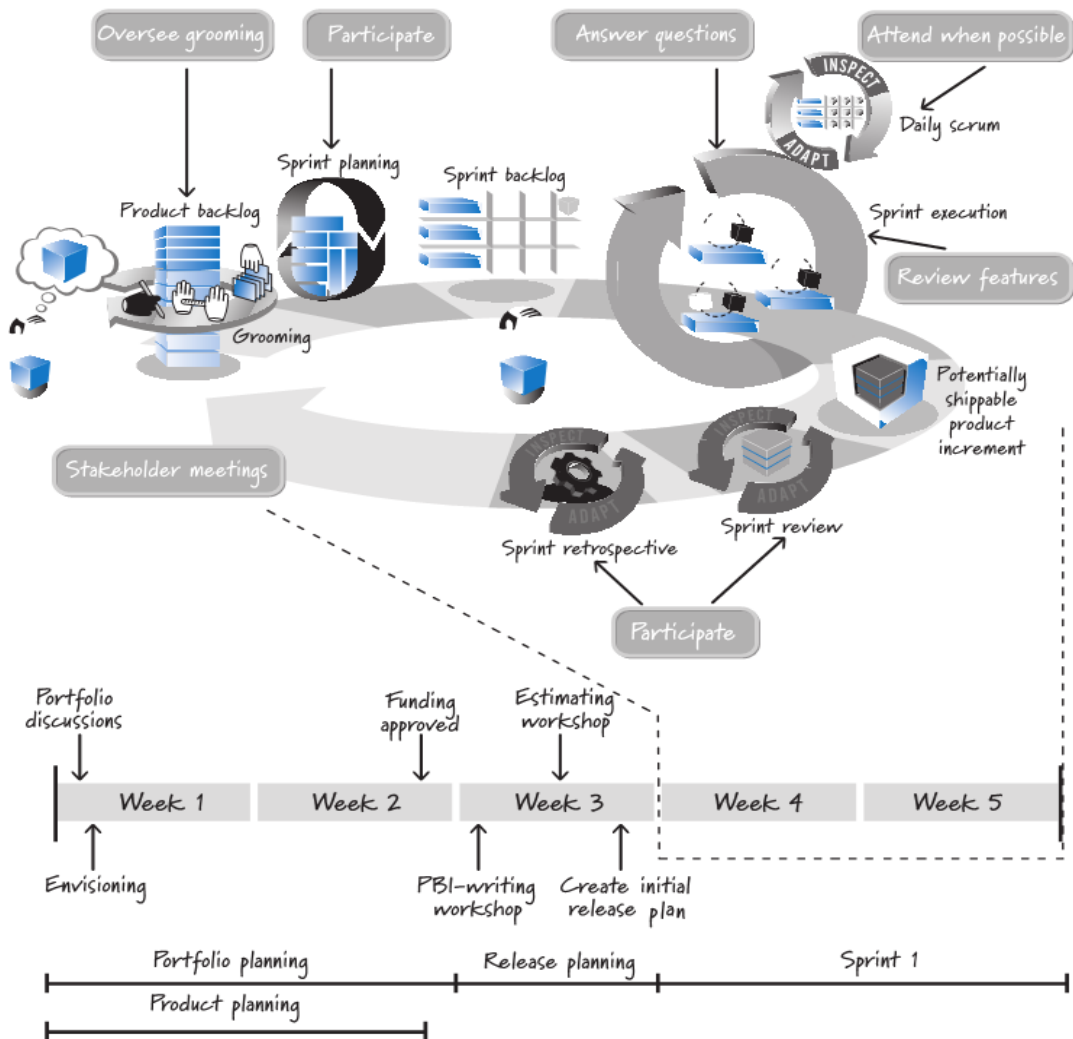
Buenos *Product Owners* tienen confianza en sus ideas, conocen el dominio del que hablan, saben comunicar sus pensamientos de una forma clara y concisa, y pueden motivar a ambos grupos cuando el entorno del producto no se encuentra en el mejor estado.

6.2.3. Decisiones y Responsabilidad

La persona encargada de ser el *Product Owner* debe tener el poder para tomar decisiones importantes y ser decisivo en ellas (es decir, no cambiarlas sin una excelente razón). Frecuentemente las decisiones complicadas caen sobre sus hombros (cuestiones de cuando lanzar el producto, quitar funcionalidades, etc), y deben tener las herramientas necesarias para tomarlas en tiempo y forma manteniendo el balance entre necesidades de negocio y viabilidad técnica, con elementos como adquisición de *deuda estratégica*.

El *PO* es el encargado de asegurarse que los recursos del producto se estén usando de la forma más económicamente sensible. Posee múltiples oportunidades para reformar el *Product Backlog*, reajustar prioridades del producto, o hasta cancelar su desarrollo. Por todo ésto, cae bajo el la responsabilidad de entregar buenos resultados de negocio. Ser *PO* es un trabajo de tiempo completo que lo hace parte del equipo de *Scrum*, por lo que trabajar en conjunto con la gente de desarrollo es esencial (éste no es un caso de “Nosotros Contra Ellos”).

6.3. Ejemplo de un Día en la Vida del *Product Owner*



Las primeras dos semanas se van en planificación del producto y del *Portfolio*. El *PO* es el encargado de reunirse con stakeholders del producto para unificar su visión y desarrollar una propuesta atractiva para conseguir fondos, y influenciar la misma con lo aprendido en las reuniones con los managers del *Portfolio*. Una vez que se completa la planificación del producto, se presenta en la empresa y se procede a conceder o denegar fondos para el mismo.

La tercer semana se dedica a la planificación inicial del *Release*, en la forma de workshops para crear y estimar *PBI* de alto nivel junto a los stakeholders y el equipo de desarrollo, que serán usados en planificaciones futuras. Luego, el *PO* se reúne nuevamente para estimaciones más precisas, reduciendo el tamaño de los *PBI* que entrarán en el primer *Sprint* y estableciendo una fecha de finalización para el mismo junto al equipo de desarrollo.

En las semanas 4 y 5, el equipo de desarrollo realiza el primer *Sprint*. En el transcurso del mismo, el *PO* intenta estar lo más presente posible, participando del *Daily Scrum* cuando puede para clarificar dudas y transmitir mejor la idea detrás del producto. También está disponible fuera del *Daily* para responder preguntas o revisar funcionalidades, delegando éstas tareas si no pudiese realizarlas él mismo. Terminado el *Sprint*, participa de la *Review* para verificar el *Release*, y de la *Retrospectiva* si es posible.

6.4. Quién Debería ser *Product Owner*?

6.4.1. Desarrollo Interno

Si un equipo de desarrollo interno está produciendo un software para el sector de Ventas, entonces una persona de Ventas con el poder para tomar las decisiones previamente descritas es un *PO* indicado.

6.4.2. Desarrollo Comercial

Si se está construyendo un producto para clientes externos, el *PO* debería ser un miembro de la organización que represente los intereses de los clientes (por lo que usualmente suele venir del área de Project Managment o Marketing).

6.4.3. Desarrollo con *Outsourcing*

Supongamos que el cliente “Compañía A” establece un contrato con la “Compañía B” para construir una pieza de software. En éste caso, el rol del *PO* debería salir siempre del cliente, pues es la persona con más entendimiento y visión del producto (en general, B siempre suele tener un representante de sus intereses también, quien se comunica con el *PO*)

6.4.4. Desarrollo con Componentes

Algunas organizaciones desarrollan pequeñas partes del producto que luego serán integradas en una solución completa. En éstos casos, el *PO* para el equipo del componente debería ser alguien del ámbito técnico, por ejemplo alguien capaz de definir y priorizar *PBI* con detalle dentro del *Backlog*. El equipo de desarrollo de componentes necesita un *PO* más especializado pues frecuentemente debe comunicarse y integrar al trabajo la visión de varios equipos de negocios encargados de diversas áreas del producto.

6.5. *PO* Combinado con Otros Roles

Si la capacidad lo permite, la misma persona podría actuar de *PO* para distintos productos. Sea de uno o más proyectos, ésta persona no debería ser *ScrumMaster*, pues los roles se desbalancean entre sí y podrían representar un conflicto de intereses.

6.5.1. *Product Owner Team*

Cada equipo de *Scrum* debe tener **una** persona con el poder y la determinación para tomar todas las decisiones y responsabilidades previamente, actuando como voz de los stakeholders y nexo entre ellos y el equipo de desarrollo. Sin embargo, hay veces en las cuales el trabajo del *PO* es tanto que no puede ser realizado por una única persona full-time.

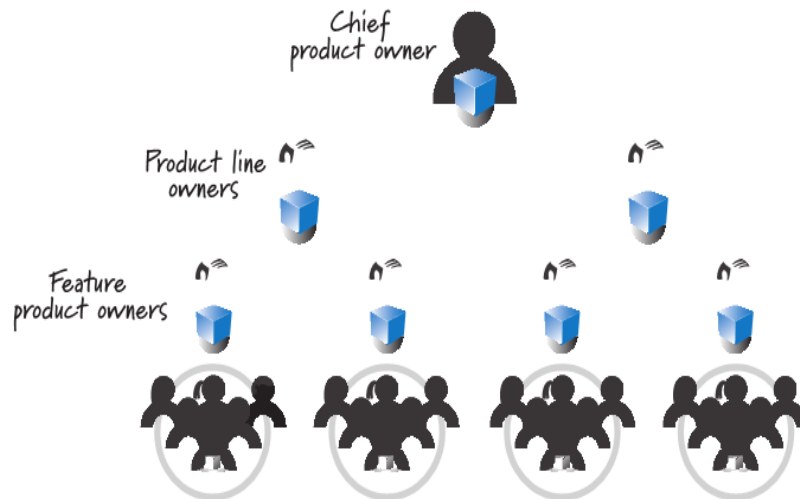
Para ocasiones como éstas, se suele formar un equipo, el cual realizará tareas delegadas por el *PO* previamente. Es importante que el equipo no cause que todas las decisiones comiencen a aprobarse por “comité”, porque ésto ralentizaría gravemente el avance del producto (es decir, el núcleo central de decisiones debe mantenerse en el *PO*). Como vemos, generar un equipo puede ser peligroso, por lo que se lo debe considerar con cuidado (tal vez el problema de verdad radique en que el *PO* actual necesita un proyecto más pequeño, o el producto gigante actual debería particionarse, por ejemplo).

6.5.2. *Product Owner Proxy*

Un **POP** (*Product Owner Proxy*) es una persona designada por el *PO* para actuar en su nombre en situaciones particulares. Es alguien que puede tomar ciertas decisiones por sí sólo, pero no las más importantes. Ésto le permite al *PO* delegar ciertas actividades como la participación en el *Daily Scrum* a otra persona, en caso de que la cantidad de tareas a realizar sea abrumadora, y es una solución mucho más controlable que crear un equipo para el *Product Owner*.

Para que ésta solución sea efectiva, el *PO* debe darle poder al proxy para tomar algunas decisiones por sí sólo, y no alterar cada una de las decisiones tomada por el mismo sin razón, pues ésto dañaría la confianza entre el proxy y el equipo de desarrollo.

6.5.3. Chief Product Owner



Supongamos que tenemos un producto masivo, con un equipo de desarrollo de 2500 personas y equipos de 10 personas. Tenemos entonces 250 equipos, por lo que tener un sólo *PO* para todo el producto no sería la decisión más sabia. En éste caso, una jerarquización del rol de *Product Owner* es la mejor opción.

Siempre tiene que haber un *Chief Product Owner*, encargado de las decisiones más importantes. A éste reportan un conjunto de *Product Line Owners*, que representan los intereses de un área del producto en la cual trabajan. A cada uno de éstos reportan *Feature Product Owners*, encargados de funcionalidades del producto. Es importante que cada *PO* tenga poder para tomar decisiones en su nivel, caso contrario ésto se transforma en una delegación infinita de decisiones hacia arriba, perdiendo el sentido de la jerarquización, y perdiendo demasiado tiempo valioso.

7. *ScrumMaster*

El *ScrumMaster* se encarga principalmente de ayudar al equipo de *Scrum* a lograr su propia adaptación del *framework*, ayudando a que todos adopten los valores centrales y prácticas dentro de los lineamientos de *Scrum*.

7.1. Responsabilidades Principales

7.1.1. Coaching

El *ScrumMaster* es el coach ágil del equipo de *Scrum* (incluído el *PO*). Al supervisar a ambas partes, se remueve la barrera entre ellos y permite al *PO* afectar el desarrollo del producto de forma directa. Tiene un rol de agente de cambio, ayudando a equipos a transformarse y adaptar el proceso de *Scrum*, haciendo claras las ventajas del mismo a corto y largo plazo.

Con respecto al equipo de desarrollo, el *ScrumMaster* se encargará de revisar como el mismo utiliza *Scrum* e intentará llevar al equipo a la implementación ideal para su caso. Esto puede ser mediante ciertos rituales (como la *Retrospective*) o con métricas (por ejemplo ayudando al equipo a medir su velocidad para mejorar). En contraste, el *ScrumMaster* ayudará al *PO* a cumplir sus responsabilidades principales (por ejemplo asistiendo con tareas de *Grooming*), además de manejar las expectativas que el mismo tiene con respecto al producto, siendo a la vez el receptor de sus quejas.

7.1.2. Líder en Servicio

Si bien el *ScrumMaster* es el coach del equipo, tiene un rol de servicio, asegurándose que las necesidades mas prioritarias del equipo sean atendidas día a día.

7.1.3. Autoridad del Proceso

Dentro del equipo, el *ScrumMaster* es el encargado de ayudar a mejorar el proceso de *Scrum*, con el objetivo de maximizar el *Business Value* entregado en cada *Sprint*, manteniendo los valores y prácticas principales del proceso. No es el responsable de que el trabajo del producto se complete, pero sí el responsable de que el equipo encuentre un proceso óptimo para asegurarse que el trabajo se complete.

7.1.4. Removedor de Impedimentos

Esta responsabilidad se basa en quitar bloqueos que podrían bajar la productividad del equipo (cuando no miembros no pueden hacerlo). Por ejemplo, si un equipo de *Scrum* no puede cumplir los *Sprints* por problemas de servidor (y ellos no tienen acceso al mismo), el *ScrumMaster* actúa, haciendo todo lo posible por solucionar el problema, colaborando con el departamento de la empresa encargado de mantener los servidores.

7.2. Características y Habilidades

7.2.1. Experto en Scrum

Para realizar la función de coach es esencial tener un buen conocimiento de cómo y por qué funciona *Scrum*. Además, se debe tener un conocimiento técnico del producto que maneja el equipo y conocimiento del dominio de negocio del producto (para poder colaborar con el *PO*).

7.2.2. Cuestionador Paciente

A la hora de mejorar la implementación de *Scrum*, un buen *ScrumMaster* no responde directamente sino que lleva a los miembros naturalmente a una evolución dentro de su *framework*. Ésto puede lograrse haciendo que se cuestionen por qué se realizan las cosas de determinada forma, y si podrían (o deberían) ser diferentes.

Como el *ScrumMaster* no da una respuesta directa, debe ser paciente y guiar al equipo a que encuentren sus respuestas apropiadas a su debido tiempo, estableciendo constante diálogo de ayuda a lo largo del proceso (claro que ésto también está sujeto a la urgencia del producto y la empresa).

7.2.3. Colaborativo y Protector

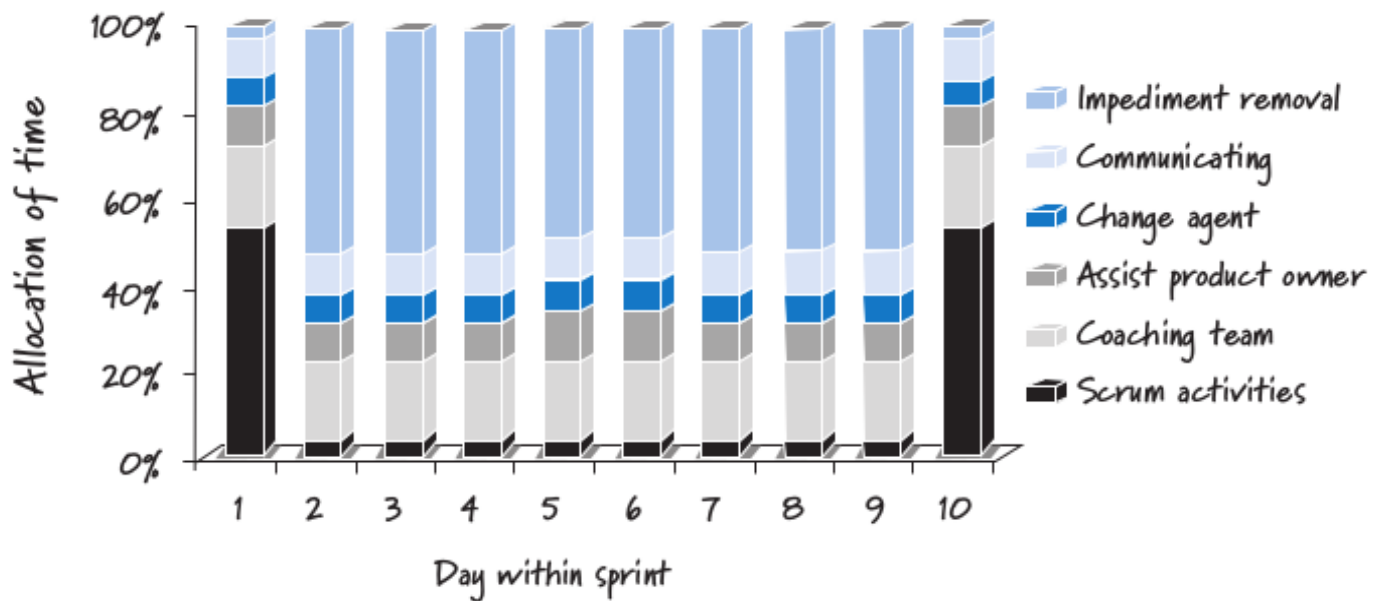
El *ScrumMaster* debe tener habilidades excelentes para colaborar con el *PO*, el equipo de desarrollo, y otras partes que puedan estar involucradas (por ejemplo cuando remueve impedimentos). Además, si se busca maximizar el nivel de colaboración en el equipo, puede fomentarlo con el ejemplo.

Además de colaborador, se tiene un rol de protector asociado al equipo, en el sentido de que lo protege contra los impedimentos técnicos que podrían ralentizar el *Sprint* o personas con diferentes agendas. En general, el contexto superior a la protección es alcanzar un balance entre las necesidades de negocio del producto y el bienestar de trabajo del equipo.

7.2.4. Transparente

La característica más importante de todas. Un *ScrumMaster* debe ser transparente en todas sus formas de comunicación. Agendas diferentes al bienestar del equipo o ocultamiento de información general del mismo dificultan la tarea de organización y adaptación al proceso, lo cual hace que *Scrum* no pueda alcanzar sus objetivos de negocio para el equipo.

7.3. Un día en la vida del *ScrumMaster*



El gráfico presentado actúa a modo de ejemplo para un equipo de *Scrum* recién formado. Podemos ver un alto porcentaje al comienzo y final, correspondiente a actividades de *Scrum*, como pueden ser *Sprint Planning*, *Retrospective*, *Dailies*, etc. El tiempo de coaching está asociado a esto, pues es aquí donde el *SM* ayuda a los miembros del equipo a mejorar su uso de *Scrum* y otras prácticas técnicas.

Parte del día también se dedica a comunicación, con tareas como revisar el *Backlog* con el equipo, actualizar *Burndown Charts*, discusiones, etc. Además de esto, ayuda al *Product Owner* con el *grooming* o a establecer *tradeoffs* de funcionalidad-dinero del producto.

Como tarea final, el *SM* remueve impedimentos del equipo, los cuales tendrán una carga dinámica de tiempo. Por ésta razón, se puede sacar tiempo de otras áreas para poder removerlos, por lo que resulta la variable más grande dentro del gráfico.

7.4. Cumpliendo el Rol

7.4.1. Quién Debería ser *ScrumMaster*?

Para nuevas organizaciones en *Scrum*, cualquier persona que tenga las características descritas puede ser un efectivo *ScrumMaster*. Se puede estar tentado a elegir siempre al *dev-lead* como *ScrumMaster*, y si bien esto puede resultar una gran opción, también puede no resultarlo. Las personas con un nivel técnico muy alto suelen estar en esa posición justamente por eso, y un nivel técnico excepcional no es un requisito para ser *ScrumMaster*, por lo que podría resultar potencial perdido (pues pasan menos tiempo liderando a nivel técnico).

Los managers de área o recursos también pueden serlo, aunque sería positivo si ya no tuvieran acceso al

management de personas si lo hacen. Ésto es porque los miembros pueden confundirse con respecto a si el individuo está hablando como *ScrumMaster* o manager (agendas divididas).

7.4.2. Trabajo de Tiempo Completo?

El resultado de ésta pregunta depende del nivel del equipo de desarrollo. A medida vayan encontrando un *framework* óptimo para trabajar, la carga de coaching sobre *Scrum* será menor. Algunas otras cosas siempre son altamente variables (como remover impedimentos). Además, una vez que los problemas a nivel equipo se resuelven, un *SM* tiene lugar para resolver impedimentos y ser un agente de cambio a nivel empresarial.

En general, no resulta un trabajo de tiempo completo (aunque si pesado), y puede combinarse con otros roles.

7.4.3. *ScrumMaster* Combinado con Otros Roles

Si una persona tiene capacidad para ser parte del equipo de desarrollo y *SM*, se puede intentar hacerlo, pero pueden surgir problemas. Los más comunes son de conflicto de interés: Qué ocurre si hay poco tiempo y hay tareas críticas de desarrollo por terminar, pero aparece un impedimento que el *SM* debe resolver? Cualquier *tradeoff* realizado en ésta situación bajaría el nivel del producto o de *Scrum*, y ésta situación es bastante común dada la impredecibilidad de los impedimentos.

Una mejor alternativa suele ser que una persona actúe de *SM* para varios equipos de *Scrum* (si realmente tiene la capacidad libre). Ser un buen *ScrumMaster* requiere un conjunto de habilidades valiosas y no tan comunes, por lo que es preferente que una persona que ya las tenga pueda compartirlas con varios equipos siempre que pueda. Como siempre, la respuesta correcta definitiva depende de la organización y su contexto.

Una combinación que se debe evitar es la de *ScrumMaster* y *Product Owner*. El *SM* actúa como coach del equipo, por lo cual es el coach del *PO*, y es muy difícil ser tu propio coach. Además, el *Product Owner* tiene la autoridad para transmitir ciertas demandas al equipo de desarrollo, y el *ScrumMaster* es el que debería actuar como agente de balance entre estas demandas y las necesidades y habilidades del equipo de desarrollo, lo cual es dificultoso de lograr si una persona es ambos roles.

8. *Development Team*

El *devteam* hace referencia a la denominación que *Scrum* utiliza para englobar a todos los individuos con las habilidades requeridas para entregar el *business value* prometido al cliente. Engloba diferentes roles más tradicionales como UI Designer, Tester, etc.

Ésta denominación elimina la necesidad de tener equipos de roles específicos que se pasan el trabajo entre sí (por ejemplo, los programadores entregando el producto a los testers), acercándonos más a los principios ágiles del proyecto y alejándonos de un desarrollo secuencial.

8.1. Responsabilidades Principales

8.1.1. Ejecución del *Sprint* y Tareas de Planificación

Éstas son la funciones principales del equipo. Primeramente, en relación a la ejecución, se espera que todos puedan construir, testear e integrar los *PBI* del *Sprint* de forma incremental, terminando con el producto en estado *potentially shippable*.

Además de lo anterior, se espera que todo el *devteam* participe de diferentes tareas de planificación. Algunas de ellas son el *Daily Scrum* (para inspeccionar el progreso del objetivo del *Sprint*, validar tareas y adaptar) y *Grooming* (crear, refinar, estimar y priorizar *PBI's*, aproximadamente un 10-15 % de cada *Sprint*).

Sumado a los rituales mencionados, el equipo participa de la planificación inicial al comienzo de cada *Sprint*, donde en colaboración con el *PO* y el *SM*, se establece el objetivo del *Sprint* y los *PBI* asociados al mismo. Para un *Sprint* de 2 semanas, ésta planificación debería durar aproximadamente medio día.

Los últimos rituales de los cuales el *devteam* es parte son la *Sprint Review*, donde se revisan las funcionalidades recientemente completadas y se valida el objetivo cumplido del *Sprint*, y la *Retrospective*, donde todo el equipo de *Scrum* inspecciona su implementación del proceso y analiza fallas para evolucionar de forma iterativa, con el objetivo final de poder entregar más valor al cliente.

8.2. Características y Habilidades

8.2.1. Auto-Organizable

Los miembros del equipo se auto-organizan entre ellos para determinar la mejor forma de enfrentar el *Sprint*. No hay un rol de *Project Manager* que tome todas las decisiones y les diga a todos como hacer su trabajo (y un *SM* jamás debería hacerlo), por lo que la auto-organización se genera de forma emergente. Ésta manera de organizar fuerza a todos los miembros del equipo a ganar contexto del dominio del producto, lo cual luego lleva a una mejor distribución del trabajo. De la misma forma, una mala organización puede ser señal de una falta de entendimiento del producto, suponiendo que todos los miembros participen y el dueño del producto no cambie el objetivo del *Sprint* a medio camino (lo cual arruina la organización).

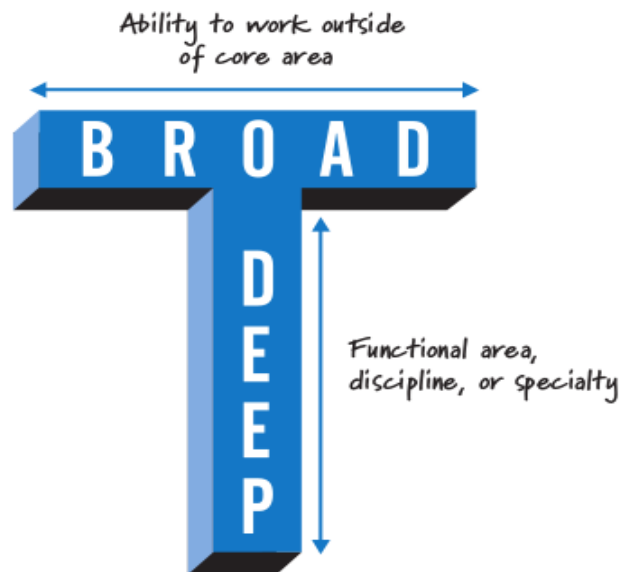
8.2.2. Diversamente Multifuncional

Esta característica va de la mano con la clasificación de *devteam*, pues al no haber equipos para cada rol diferente (desarrollo/QA/etc), el único equipo existente tiene que tener todas las habilidades necesarias para sacar un ítem del *Backlog* y refinarlo hasta que cumpla con la *Definition of Done* del equipo. En estructuras más tradicionales donde se tienen equipos con roles, cada entrega que un equipo hace al otro puede acarrear problemas de comunicación y errores costosos.

Otra característica nace desde la diversidad, pues al haber tantas personas diferentes en un grupo se obtienen varias estrategias, interpretaciones y modelos para la solución de problemas, situación que debería

llevar a una solución final de mayor calidad. Es importante que el equipo de desarrollo tenga un buen mix *senior/junior*, para no sufrir de problemas clásicos como puede ser falta de habilidad del equipo.

8.2.3. Habilidades en Forma de T



Tener habilidades en forma de T flexibiliza la capacidad que tiene el equipo de desarrollo para resolver problemas. La mejor forma de entender este tipo de habilidades es con un ejemplo, como lo es el caso de Juan. Juan es especialista en desarrollo, por lo que sabemos que su trabajo en esa área será de alto nivel (profundidad de la T). Sin embargo, Juan puede también trabajar fuera de su área de especialización, por ejemplo haciendo documentación, testing o UX. No realiza el trabajo tan bien como un especialista, pero puede ayudar a resolver cuellos de botella que el equipo pueda estar teniendo en esas áreas o colaborar en algunas *User Stories* al respecto (anchura de la T). Es poco realista apuntar a que todos puedan hacer todo el trabajo igual de bien, pero siempre se debería poder colaborar en algunas áreas fuera del *expertise* personal.

Es ideal intentar formar los equipos que mejor cumplan con los requisitos del proyecto, pero también se debe tomar en cuenta la naturaleza evolutiva de los mismos, por razones como cambios en las reglas de negocio o evolución del producto. Por éste motivo, es crítico poseer un ambiente de trabajo donde las personas del equipo estén en constante aprendizaje de sus áreas actuales, además de tener un tiempo para que puedan aprender nuevas tecnologías y experimentar.

8.2.4. Actitud de Mosquetero

Todo el equipo de *Scrum* debería tener la actitud de *Los Tres Mosqueteros*: “*Todos para uno, y uno para todos*”. Ésto refuerza el punto de que el equipo posee la responsabilidad colectiva de terminar el trabajo bien y a tiempo (no algún manager dentro del equipo). No existe frase como: “Yo hice mi parte de la card, vos no hiciste la tuya, todos fallamos.” Las deficiencias inesperadas son cosas que tienen que anticiparse, y se debe trabajar en conjunto para solucionarlas cuando ocurran. En definitiva, el fallo resulta problema de todos.

Tener Habilidades en Forma de T ayuda en el proceso de autosuficiencia y a la hora de ayudar a los demás en otras áreas, fomentando la actitud necesaria para solver las deficiencias previamente mencionadas. Sin embargo, puede ocurrir que una persona no sea capaz (por inexperiencia) de ayudar a otra con su trabajo, caso en el cual se puede resolver con ayuda de *coaching* junto a una persona mas especializada. Esto nos asegura de estar incrementando las habilidades del equipo de forma recurrente.

Aunque existan limitaciones al nivel de habilidades de cada persona, los miembros del equipo pueden organizar su trabajo de forma tal que estas deficiencias estén consideradas y ningún miembro del equipo

tenga exceso de trabajo (por ejemplo, no dejar el testing para el último momento a cargo únicamente del especialista en ello).

8.2.5. Comunicación Eficiente

Scrum abandona muchas prácticas tradicionales y las reemplaza con un foco en la comunicación, por lo que es sumamente crítico que éstas se lleven a cabo de forma eficiente. Todo el equipo de *Scrum* debe comunicarse entre sí, por lo que se deben identificar y utilizar momentos claves para compartir información, y compartirla minimizando el *downtime* para reducir el overhead de las reuniones. Esta comunicación frecuente le provee al equipo más oportunidades para inspeccionarse y adaptarse, lo cual resulta en una toma de decisiones cada vez más rápida y efectiva.

El *Manifiesto Ágil* establece algunos puntos para ayudar con este objetivo. Uno de ellos consiste en **maximizar la comunicación cara a cara**. Es decir, intentar en la medida de lo posible que todo el equipo éste en un mismo lugar para poder tener éstas conversaciones en los rituales de la forma más eficiente posible (en comparación a alternativas como *e-mails*). En el caso de que el equipo se encuentre desafortunadamente distribuido, se puede utilizar alguna tecnología de teleconferencia para conseguir un resultado similar (pero peor) al cara a cara.

Otro punto a considerar es la reducción del tiempo consumido en rituales donde los miembros del equipo hacen un proceso que casi no agrega valor. Por ejemplo, que un miembro deba atravesar *CUATRO* niveles de indirección hasta poder hablar con un usuario o consumidor es un impedimento sobre la comunicación eficiente que debería ser resuelto (proponiendo un mejor esquema para poder acceder al usuario). Tener que crear documentos que no aportan valor y que deben ser aprobados luego de un largo proceso reduce la eficiencia de la comunicación, y es un fenómeno habitual dentro de metodologías tradicionales. Éstas situaciones deben ser identificadas y eliminadas para mejorar la performance.

Finalmente, tener equipos pequeños ayuda a la comunicación, pues mientras más personas tiene un equipo más *overhead* se agrega. Ésto es porque las vías de comunicación dentro del equipo no escalan de manera lineal con el tamaño n del mismo, sino que lo hacen de acuerdo a la fórmula $N(N-1)/2$ ¹.

Demás está aclarar que la transparencia a la hora de comunicar es clave. Para garantizar los puntos anteriores (como la actitud mosquetera), la confianza entre los miembros del equipo es fundamental, por lo que nunca se debe comunicar con intención de engañar o “ocultar la verdad”.

8.2.6. Tamaño Apropriado

Scrum favorece equipos de 4 hasta 8 personas. Diferentes papers² proponen beneficios de este tamaño, como por ejemplo:

- Menos holgazanería social: realizar un menor esfuerzo creyendo que los demás harán tu trabajo.
- Menos tiempo invertido en esfuerzos de coordinación entre el equipo.
- Mayor número de interacciones constructivas (con un equipo que respeta características previas)
- Todos “se sienten parte del equipo”, pues al tener el trabajo distribuído ninguno puede simplemente desaparecer y no ser notado.

Si bien *Scrum* favorece equipos pequeños, no significa que no sea escalable. A la hora de escalar *Scrum*, no se aumenta el tamaño del *devteam*, sino que se aumenta la cantidad de equipos de *Scrum* en el proyecto. Un avance de esto muy conocido es el concepto de *Scrum de Scrums* (más en el Capítulo 12)

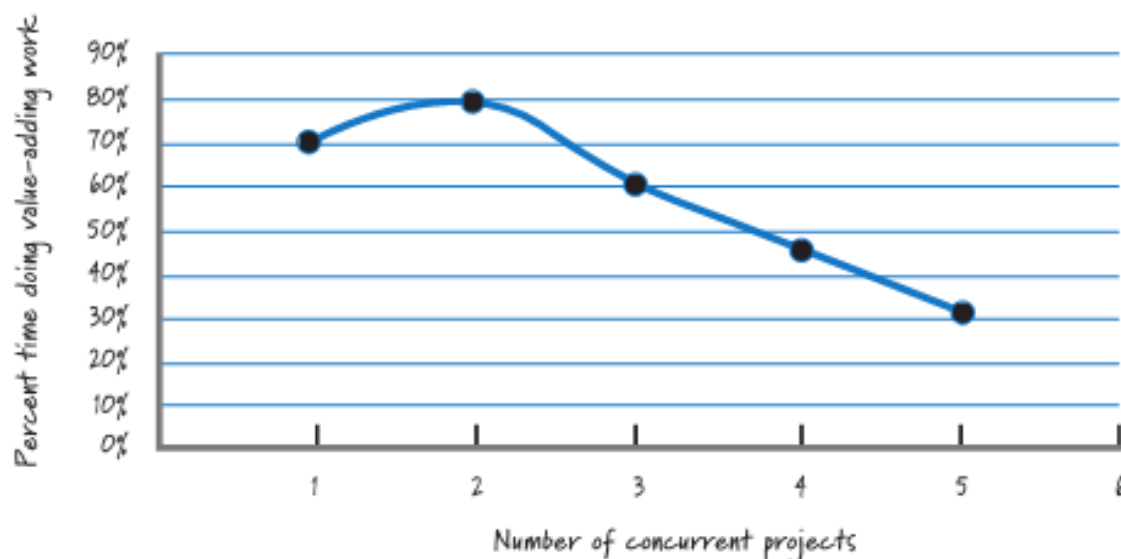
8.2.7. Concentración y Compromiso

Trabajar bajo *Scrum* requiere de un equipo concentrado en su desempeño y comprometido con la finalización correcta de los objetivos del grupo. Si una persona trabaja sobre uno o dos productos, es mucho más

¹<http://www.pmpmath.com/cc.php>

²Putnam 1996, Putnam and Myers 1998, Cohn 2009

sencillo mantener este nivel de concentración y compromiso. De hecho, existen varios estudios³ que muestran un declive hacia abajo en la productividad de una persona de acuerdo aumentan la cantidad de proyectos de los cuales forma parte.



Se parte de la base que ninguna persona es 100 % productiva, pero su productividad aumenta cuando se le agrega un proyecto a su repertorio. Ésto se basa en la idea de que si la persona se encuentra bloqueada en un proyecto, puede pasar al otro y realizar actividades más productivas sobre el mismo. Ahora bien, trabajar en 3 o más proyectos en simultáneo reduce drásticamente la productividad de las personas, pues se pasa mucho más tiempo coordinando, recordando y trackeando información para cada proyecto que haciendo trabajo de valor dentro del mismo.

8.2.8. Trabajo a un Ritmo Sustentable

Utilizando el desarrollo secuencial, solemos delegar las tareas de testing e integración para el final. Éstas tareas con claves y arduas en finalizar, lo cual conlleva muchos días y noches de trabajo sumamente excesivo hasta terminarlo.

Uno de los pilares de *Scrum* consiste en trabajar a un ritmo considerado sustentable, permitiéndonos mantener un buen clima y ritmo de trabajo, mientras entregamos productos de la mayor calidad posible. Si comparamos la intensidad de ambas metodologías, la secuencial es parecida a una exponencial (muy ardua en el final), mientras que *Scrum* se mantiene constante, mediante el uso de buenas prácticas dentro de todos los *Sprints* (como refactor, integración continua y tests automáticos). Todo ésto nos permite entregar valor al cliente mucho más rápido, de forma continua, y con un estrés mucho menor.

9. Estructuras de Equipo de *Scrum*

9.1. Equipos de *Feautres* vs Equipos de Componentes

Un equipo de *features* representa un equipo multifuncional y multicomponente, que puede sacar *features* del *Product Backlog* y completarlas. Como ya vimos en el capítulo anterior, *Scrum* favorece éstos equipos.

Por otro lado, un equipo de componente se concentra en un aspecto específico del desarrollo, por lo que podría ser usado para crear una parte de esa *feature*. Por ejemplo, al desarrollar un sistema de GPS uno

³Por ejemplo, Wheelwright and Clark, 1992.

podría tener un equipo de componente encargado de manejar el código asociado al cálculo de rutas para el resto del GPS. Es común que un equipo de componentes reúna gente de la misma área], por ejemplo tener un equipo centralizado de UX, encargado de realizar diseño de UI para cada aplicación por separado.

Supongamos que tenemos una empresa con varios equipos de componentes para sus áreas, a las cuales quiere asignarle trabajo de múltiples proyectos. Por cada proyecto, se tendrá un *PB*, el cual tendrá diferentes *PBI's*. Cada *PBI* es dividido en diferentes capas en relación a la cantidad de componentes distintos de la tarea (por ejemplo una capa para UX), trabajo usualmente realizado por un arquitecto. Luego de dividido el *PBI*, se envía la capa al equipo de componente correspondiente, el cual la deposita en su propio *Backlog*. Una vez finalizado el trabajo de todas las capas, se integra todo con cuidado haciendo *Scrum de Scrums*. Se tiene una situación como la siguiente:

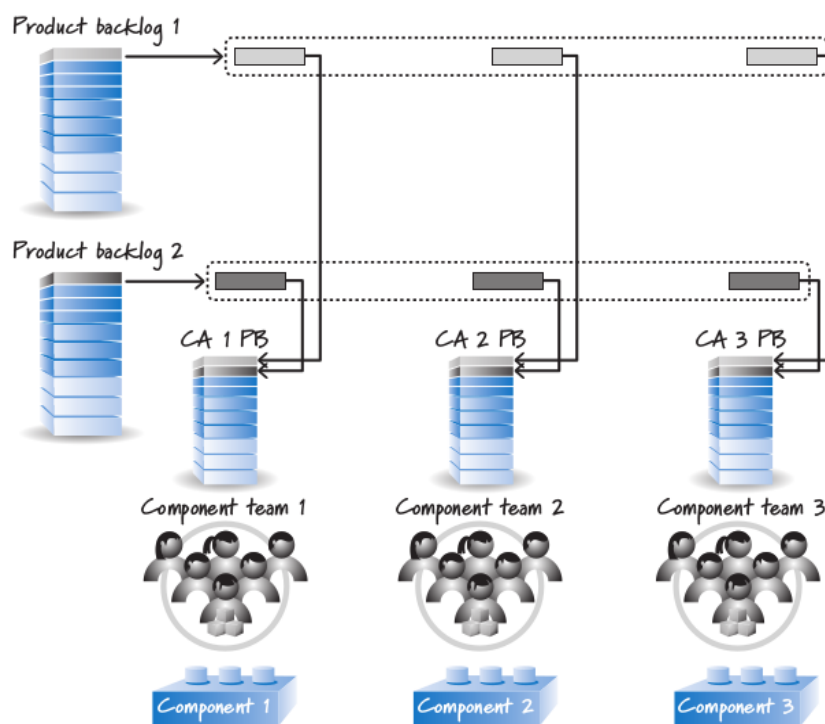


Figura 6: Dos productos y múltiples equipos de componentes

Con sólo dos productos es manejable. Sin embargo, con más productos, el hecho de que cada equipo de componentes deba priorizar tanto trabajo de muchas fuentes diferentes, además de coordinar e integrar su trabajo con todos los demás equipos, hace que éste sistema no escale con éxito. Además, como los equipos de componentes no son multifuncionales, un fallo en alguna capa del *PBI* podría retrasar todo el ítem, aunque el resto de los equipos hayan completado su trabajo (y probablemente no puedan ayudar al equipo con problemas pues no son multifuncionales). Si un equipo decide no entregar la funcionalidad esperada por temas de prioridades con el resto de sus tareas tenemos el mismo problema, toda la *feature* se paraliza.

Un balance ideal contiene un modelo combinado compuesto mayormente por equipos de *features*, con equipos de componentes para cuando la idea de centralizar los recursos de un área tenga sentido económico (por ejemplo tener 2 o 3 proyectos con poca carga de UX y tener un diseñador como equipo de componente para cada equipo de *features* de cada proyecto)

9.2. Coordinación entre Múltiples Equipos

Scrum escala aumentando la cantidad de equipos de *Scrum* en el proyecto, no la cantidad de miembros dentro de un equipo (en general se apunta a 2-7 miembros). Existen dos técnicas para coordinar un proyecto

con múltiples equipos, las cuales veremos a continuación.

9.2.1. *Scrum of Scrums*

Un proyecto tiene varios equipos. Cada equipo selecciona un miembro que ellos creen que es el mejor para transmitir los problemas de dependencia inter-equipos que están teniendo actualmente. De la selección de cada equipo se arma un nuevo grupo, que será el encargado de hacer el *Scrum de Scrums*. Este ritual tiene diferentes enfoques posibles, pero el más típico es realizarlo unas veces por semana (no diariamente). Las preguntas a responder son similares a las del *Daily Scrum*:

- Qué trabajo hizo mi equipo desde la última vez que podría afectar otros equipos?
- Qué trabajo va a hacer mi equipo antes de que nos volvamos a juntar que podría afectar otros equipos?
- Existe algún problema actual de mi equipo que requiera la ayuda de otro equipo para resolverse?

El tiempo asociado al *Scrum de Scrums* debería ser similar al de una *Daily Scrum*. La resolución de algún problema en particular que se encuentre en la reunión se puede dejar para después del ritual, de forma tal que solo los participantes que formen parte de la exploración y resolución de ese problema tengan que atender (a no ser que ese problema sume a todos los equipos, caso en el que se extiende el *SoS*).

Si se tienen muchísimos equipos, esto teóricamente escala. Un producto muy grande suele estar dividido en clusters por área, por lo que tiene sentido hacer un *SoS* por área, y luego un *SoS* de nivel superior (algo como un *SoSoS*) para englobar todas las áreas. Aún así, hay métodos más eficientes para éstos casos.

9.2.2. *Release Train*

Release Train es un enfoque que provee sincronización entre todos los equipos basada en que todos sigan un ritmo común. La metáfora del tren tiene que ver con que existen *features* que se “iran de la estación de tren” en un tiempo determinado, por lo que los equipos deberán depositar su cargamento antes de la ida. Como en la vida real, éste tren no espera. Sin embargo, siempre se puede tener la seguridad de que otro tren partirá en el futuro. Las reglas de un *Release Train* se definen como⁴:

- Días de planificación y *Release* prefijados (con scope variable).
- Diferentes milestones (metas) intermedios, globales y por objetivo.
- La longitud de cada iteración es común a todos los objetivos.
- Se implementa integración continua a nivel componente/*feature* y sistema (capa superior).
- *Potentially shippable increments (PSI)* del *Release* están disponibles a intervalos regulares, listos para revisión del consumidor y testing interno.
- Se pueden hacer iteraciones con el objetivo de reducir deuda y afianzar validaciones a nivel *Release*.
- Uso compartido de herramientas según área (interfaces, SDK's, servicios web, frameworks).

⁴Leffingwell 2011

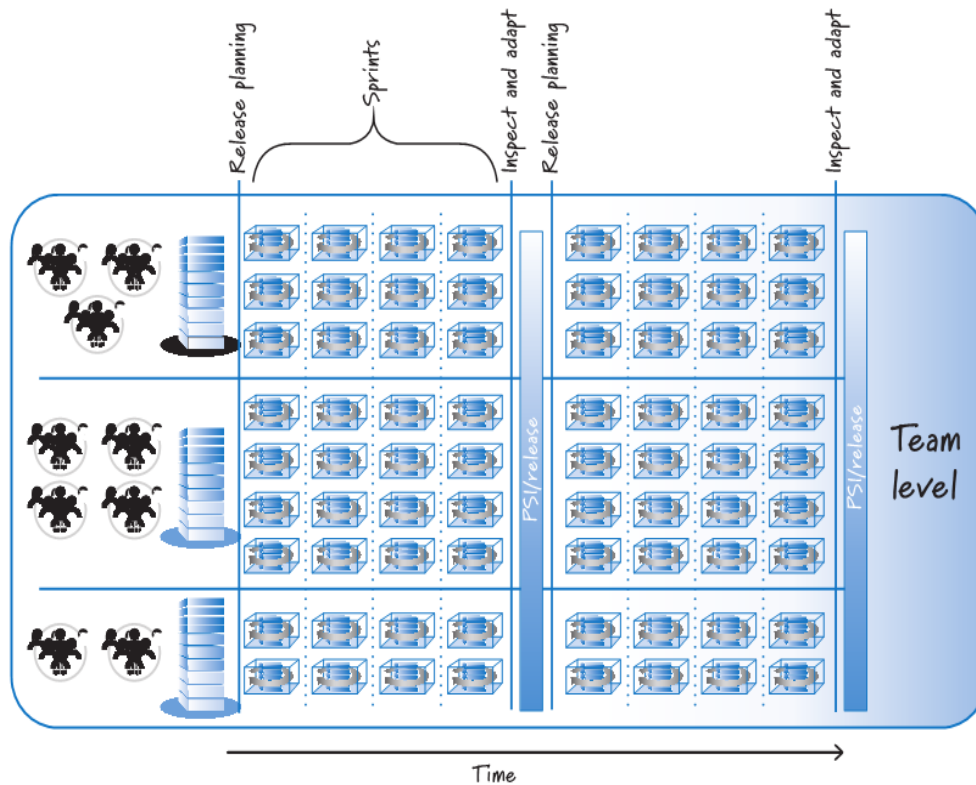


Figura 7: Estructura de un *Release Train*

Como vemos en la imagen, tenemos 9 equipos divididos en 3 áreas según como equipos de *feature*. Cada uno realizará su propio *Sprint*, sacando tareas del PB asociado a la *feature*. Luego, usando *SoS*, cada área integra su trabajo y coordinan cada *Sprint*. Además, en la medida de lo posible, se deberían realizar tareas de integración y testing fuerte para asegurarse que todas las *Features* estén compuestas de forma correcta antes del *PSI*. Algo común para esto es utilizar el último *Sprint* solo para testing/integración, aunque esta tendencia debería decrecer con la maduración de los equipos.

La duración de todos los *Sprints* es idéntica para todos los equipos, y además comienzan y terminan en la misma fecha (habilitando la sincronización mencionada previamente para cada *feature* del producto). Finalmente, luego de una cantidad prefijada de *Sprints*, se obtiene un *PSI*. Saber que estos momentos de entrega ocurrirán en un tiempo determinado habilita a las organizaciones a ordenar sus otras actividades para fechas futuras.

Cada *Release Train* comienza con una reunión para planear el *Release* entre todos los miembros. Se necesita un cuarto bastante grande. Generalmente, el Chief *PO* lidera esta actividad, mientras que los equipos de *Scrum* se juntan en clusters por *feature* o área. Una vez que el Chief *PO* dió el mapeo general para el *PSI*, los *PO* de cada área dan su visión a los equipos de *Scrum* correspondientes.

Cada equipo de *Scrum* comienza ahora a poner *features* en *Sprints* (*Sprint Mapping*). Como este producto es multi-equipo, se esperan dependencias entre los mismos. Cuando un equipo encuentra una, simplemente se para de su mesa y va a la mesa encargada de resolver la dependencia, y le pregunta si puede hacerlo antes del siguiente *PSI*. En ese caso, la *feature* entra al *Sprint*. Mientras esto ocurre, las personas con roles multifuncionales como el Chief *PO* van de mesa en mesa asegurando una visión unívoca y central de lo planificado.

Cuando se tiene que entregar el *PSI*, ocurren las mismas actividades que en *Scrum* tradicional pero a mayor escala entre todos los equipos (revisión de lo completado y retrospectiva para mejora del proceso mucho más complejo)

10. Managers en *Scrum*

Si bien el *framework* de *Scrum* no menciona el rol de manager, ésto no quiere decir que el mismo no sea importante para la organización ágil del producto. Específicamente hablaremos de *FAM's* (Functional Área Managers) y *PM's* (Project Managers).

10.1. Responsabilidades de un *Functional Manager*

10.1.1. Construir Equipos

- **Definir fronteras:** si bien un equipo de *Scrum* se organiza por si solo, en general los proyectos que encara son definidos por los managers.
- **Definir objetivos claros:** los managers proveen objetivos concretos para los equipos, dándoles propósito y dirección en el producto. El objetivo es bastante *macro*, luego entre el *ScrumMaster* y el *devteam* se baja a tierra.
- **Definir miembros:** los managers suelen ser los que deciden la composición de cada equipo, aunque ésto no quita que los miembros puedan dar su opinión al respecto para mejorarlo. Cada *FAM* representa un área (i.e UX, testing), y entre todos ellos se llega a una decisión colectiva de formación del equipo, para asegurarse que todas las necesidades estén cubiertas. Se debería crear un equipo con los requisitos vistos en el Capítulo 11, como diversamente multifuncional y con Habilidades en Forma de T.
- **Cambiar su composición:** un manager tiene la obligación de cambiar los miembros de un equipo si dicha acción beneficiaría al mismo (por ejemplo, removiendo algún miembro que tenga una actitud negativa y poco productiva, no solucionable con conversaciones) o a la empresa (agregando un miembro que tenga habilidades claves o para *coaching* en una etapa específica del producto). Ésto siempre tomando en cuenta las potenciales contras de rearmar equipos de *Scrum*, intentando mantener la integridad del equipo como unidad . No se debería sacar gente para trabajar en un proyecto aleatorio en el medio del *Sprint*, o asignar muchas personas para múltiples equipos, por los *overheads* asociados.
- **Darle autoridad:** para que un equipo se autoorganice (como debería hacerlo uno de *Scrum*) necesitan que el manager asociado les provea de la autoridad requerida para poder tomar decisiones por su cuenta, sin necesidad de un proceso de aprobación vertical para cambios menores. Ésto se puede lograr delegando acciones al equipo, con diferentes criterios de “responsabilidad”. Una escala de niveles es propuesta por Appelo⁵, la cual es:

TABLE 13.1 Appelo’s Seven Levels of Authority, with Examples

Level	Name	Description	Example
1	Tell	Manager makes the decision and tells the team	Relocate to a new office building
2	Sell	Manager convinces the team about the decision	Decision to use Scrum
3	Consult	Manager gets input from the team before making the decision	Select new team members
4	Agree	Manager and team make the decision together	Choose logo for business unit
5	Advise	Manager advises to influence the decision made by the team	Select architecture or component
6	Inquire	Manager inquires after the team has made the decision	Sprint length
7	Delegate	Manager fully delegates the decision to the team	Coding guidelines

⁵Appelo, 2011

10.1.2. Nutrir Equipos

- **Motivación:** motivar a los miembros mediante objetivos claros y elevadores, fomentar la realización de un gran trabajo. No quitar libertades inherentes a *Scrum*.
- **Desarrollar habilidades competentes:** dentro de una organización bajo *Scrum*, cada miembro del equipo reporta a algún tipo de manager/Project Leader, que no suele ser el *SM*. Éste manager tiene la responsabilidad activa de ser un coach para los miembros a nivel técnico, promoviendo oportunidades para que las personas puedan mejorar sus habilidades y llevar mejor el producto. Accionables sobre esto pueden ser ir a conferencias sobre temas de aprendizaje, proveer feedback de lo realizado en cada *Sprint*, o designar un porcentaje de tiempo para que cada persona pueda entrenarse en algún área de interés. Dentro del feedback que se da en el *Sprint* se debe analizar si conviene según el contexto que sea individual o más *macro* a nivel *devteam*.
- **Proveer Liderazgo en el Área Funcional:** un manager funcional debería tener el mayor conocimiento del área sobre la que está designado, con respecto al resto de los miembros. Esto le permite proveer liderazgo en diferentes formas (las cuales no implican ser un jefe que les dice a todos como hacer su trabajo, lo cual sería destructivo para un equipo que se autoorganiza). Por ejemplo, puede hacer de *coach* para los nuevos miembros del área o establecer ciertos estándares como buenas prácticas para el área. También, actúa como un estabilizador de estándar, pues su expertise le permite revisar el trabajo que se hace sobre su área y aconsejar al respecto, asegurándose que todo cumpla un estándar de calidad.

10.1.3. Alinear y Adaptar el Entorno para Adoptar *Agile*

- **Promover Valores Ágiles(Cap. 3):** la mejor manera de promoverlos es adoptarlos. Los managers deberían conocerlos y utilizarlos en su día a día, además de promover que el resto del equipo lo haga. Un equipo que requiere *Scrum* puede fallar si la gente de *management* no lo adopta.
- **Quitar Impedimentos Organizacionales:** si bien el *SM* se encarga de esto, muchos de los impedimentos a nivel organización necesitan de un manager para poder ser efectivamente removidos.
- **Alinear Grupos Internos:** En general, el grupo de ingeniería o *IT* es el primero en adoptar *Scrum*, pero no deben ser los únicos. Es la obligación de los managers de alinear los diferentes grupos de la empresa para que trabajen en conocimiento de ésta metodología. Por ejemplo, el área de Recursos Humanos debería contratar gente para trabajar bajo *Scrum*, lo cual no es lo mismo que un esquema tradicional (por ejemplo, necesitas Habilidades en Forma de T). También, en el área de Ventas se debería tomar esto en cuenta, con acciones como no vender software con requerimientos inamovibles a fechas incambiables, y comunicar el uso de *Scrum* al cliente.
- **Alinear *Partners*:** transmitir a los clientes o partes tercerizadas del proyecto el uso y adopción de *Scrum*. No utilizar estructuras tradicionales con contratos inamovibles e intentar disminuir las negociaciones *fixed-price*, pues va en contra de la naturaleza iterativa y cambiante de un producto a lo largo de varios *Sprints* (se puede caer en tener que disminuir *features* o caer en deuda si lo anterior es imposible, ambos escenarios bastante negativos).

10.1.4. Manejar el Flujo de Creación de Valor

- **Manejar la Economía:** los *FAM's* suelen manejar los recursos económicos asociados a su área, es decir como se distribuyen y en que proyectos. Con el feedback iterativo obtenido en los *Sprints*, se puede decidir redirigir ciertos fondos a algún proyecto que pueda necesitar más soporte en un área específica. La decisión de si un proyecto en su totalidad sigue teniendo viabilidad económica debería ser responsabilidad del manager del *Portfolio*.
- **Monitorear Métricas y Reportes:** Como en general éstos elementos son generados para los managers, es su responsabilidad generar los únicos y necesarios para mejorar el equipo, haciendo que dichas métricas se alineen con los valores de *Scrum*. Por ejemplo, se las puede utilizar para determinar que tan

rápido se completa el ciclo de aprendizaje de un equipo (planificar, construir, *feedback*, inspeccionar, adaptar).

10.2. Project Managers

Un error común es asumir que el *ScrumMaster* es el *PM* para *Agile*. Si bien tienen cosas en común (remueven impedimentos), mayormente poseen diferencias (*SM* es un líder al servicio del equipo y el *PM* se concentra más en comandarlo y direccionarlo). Veamos cuales son las responsabilidades tradicionales de un *PM* y como se mapean en *Scrum*, donde ese rol no existe.

Project Management Activity	Description
Integration	Identify, define, combine, unify, and coordinate the various processes and project management activities.
Scope	Define and control what is and is not included in the project, ensuring that the project includes all of the work required.

Project Management Activity	Description
Time	Manage timely completion of the project by defining what to do, when to do it, and what resources are necessary.
Cost	Estimate, budget, and control costs to meet an approved budget.
Quality	Define quality requirements and/or standards, perform quality assurance, and monitor and record results of quality-focused activities.
Team (human resource)	Organize, manage, and lead the project team.
Communications	Generate, collect, distribute, store, retrieve, and dispose of project information.
Risk	Plan, identify, analyze, respond, monitor, and control project risks.
Procurement	Acquire products, services, or results needed from outside the project team.

Figura 8: Responsabilidades de un *PM* tradicional.

Project Management Activity	Product Owner	ScrumMaster	Development Team	Other Manager
Integration	✓			✓
Scope	Macro level		Sprint level	
Time	Macro level	Helps Scrum team use time effectively	Sprint level	
Cost	✓		Story/task estimating	

Project Management Activity	Product Owner	ScrumMaster	Development Team	Other Manager
Quality	✓	✓	✓	✓
Team (human resource)			✓	Formation
Communications	✓	✓	✓	✓
Risk	✓	✓	✓	✓
Procurement	✓			✓

Figura 9: Tareas de un *PM* mapeadas a roles de *Scrum*.

Con la información de la tabla, un *PM* puede transformarse en diferentes roles, como puede ser *SM* (si puede desligarse del management de comandos y control, pues va en contra de la auto-organización y libertad del equipo) o *PO* (si tienen las habilidades necesarias, como por ejemplo buen conocimiento del dominio).

10.3. Reteniendo el rol de *Project Manager*

Si bien la regla es que no debería haber alguien que coordine los equipos de *Scrum* (pues éstos son auto-organizables), existen casos de productos muy grandes donde la logística de organización es tan compleja que los equipos no pueden comandarla.

Éste no es el caso de simplemente muchos equipos para un producto, pues aquí se espera que dichos equipos formen clusters de área y se comuniquen entre ellos (por ejemplo, eligiendo representantes para cada área funcional, y que ellos se encarguen del *Scrum of Scrums*).

Sin embargo, cuando se tienen demasiados clusters, puede que la auto organización resulte demasiado compleja en todos los niveles, por lo que un *Project Manager* podría tener sentido. En éste caso, el *PM* se encarga de la organización a alto nivel (comunicación entre clusters y organización *Macro* del producto), mientras que cada equipo de *Scrum* sigue organizando sus propias tareas y comunicándose dentro de su cluster de área funcional. **El objetivo no es que el *PM* esté a cargo de todo, sino que sea la persona encargada de comunicar para resolver las múltiples dependencias entre todos los equipos y que éstos puedan comunicarse con mayor facilidad.**

11. Principios de Planificación en *Scrum*

El razonamiento que indica que *Scrum* requiere mínima planificación es un mito. En general, se le dedica aproximadamente el mismo tiempo que en métodos más tradicionales, solo que ese tiempo se utiliza de forma muy diferente. Veamos como los principios de *Scrum* se aplican a la planificación.

11.1. No se Puede Obtener la Planificación Correcta por Adelantado

El enfoque tradicional consiste en crear un plan muy detallado antes del desarrollo, tratando de conseguirlo bien de entrada. Un argumento a favor de éste plan es que, sin él, no se sabe a que dirección va el desarrollo lo cual hace difícil coordinar el recurso humano y sus actividades, especialmente en grupos grandes.

Scrum reconoce ésto, pero asumiendo que no se puede obtener una planificación correcta de entrada (como suele ser el caso). Entonces, se realiza la planificación mínima necesaria al comienzo para establecer un balance entre la planificación por adelantado y la planificación “justo a tiempo” que ocurre iterativamente y suele ser mucho más precisa.

11.2. La Planificación por Adelantado Debería Ayudar sin ser Excesiva

Como sabemos, no suele ser posible predecir con mucha certeza cuando se deberá cambiar el curso de la planificación (ni por qué factores). Por lo tanto, planificar por adelantado cada detalle a lo largo de un producto no tiene sentido, pues simplemente no tenemos una bola de cristal y hacerlo suele ser excesivamente caro. Además, podría nublar la visión real de lo que está cambiando en el producto, por querer aferrarse 100 % a toda la planificación, en vez de responder al cambio con un pivoteo. En conclusión, la planificación por adelantado es necesaria para establecer un mapeo de los siguientes pasos del producto hacia el desarrollo, pero no debe ser tan excesiva en las minucias de los detalles del futuro tal que pierda sentido.

Lo que todo ésto significa es que se debe planificar por adelantado lo mínimo, y dejar otras cuestiones de planificación futura necesarias para el último momento, donde se tiene mucha mejor información (además de reducir el costo en comparación a hacerlo todo por adelantado y que el plan falle).

11.3. Foco en Adaptación y Replanificación sobre el Plan

La planificación por adelantado es útil, pero se debe realizar en conocimiento de que se la hace cuando menos información y mas ignorancia sobre el producto se tiene. Cuando el mapeo eventualmente sufra desviaciones -escenario que no debería ser ignorado- se debe replanificar y adaptar el proceso actual para tomarlo en cuenta. Aferrarse al mapeo inicial aún en contexto de cambio implica seguir un camino ideal que no refleja la realidad de un producto.

En *Scrum*, se considera que la planificación por adelantado es útil, pero que se debe favorecer una frecuente replanificación y adaptación a medida que se validan las asunciones con el cliente, con el transcurso de los *Sprints*. Esta validación nos da un aprendizaje mayor sobre el producto que utilizamos para producir planes mejores y más útiles de forma continua. No es preocupante que éstos planes no sean 100 % correctos, pues serán reemplazados con mejores planes en sucesivas iteraciones.

No existe cantidad de planificación por adelantado que pueda sustituir hacer algo, aprender rápido y pivotar ⁶ si es necesario. Nuestro objetivo debería ser movernos por el ciclo de aprendizaje (planificar, construir, *feedback*, inspeccionar, adaptar) mencionado en el capítulo anterior de forma rápida y económica, por lo que nuestro plan debería tener como objetivo principal el aprendizaje desde el comienzo.

11.4. Manejar Correctamente el Inventario de Planificación

Considerando ambas planificaciones ya mencionadas, deberíamos tomar en cuenta que crear un inventario grande de *PBI's* suele ser una pérdida de recursos. Supongamos que tenemos un gran diagrama de Gannt

⁶Cambiar la dirección utilizando lo aprendido como base (Ries 2011)

para un plan de 36 meses. A lo largo de la planificación nos iremos dando cuenta que cometimos errores al planificar todo por adelantado, generando 2 tipos de pérdida:

- Pérdida de haber producido y detallado *PBI's* que ahora son descartados (no *epics*). Éste tiempo podría haber sido invertido en entregar trabajo de valor, en vez de producir elementos innecesarios.
- Pérdida por tener que replanificar y actualizar el plan (necesaria).

Si bien uno debe poder responderle al cliente cuando se lanzará el producto o cuando estará lista una *feature*, siempre se debería balancear la planificación hecha en un momento dado contra la probabilidad de que esa planificación sea una pérdida (por ejemplo si en una acción a realizar en 8 meses).

11.5. Favorecer *Releases* Pequeños y mas Frecuentes

Scrum favorece ésta estructura pues provee *feedback* mas rápido, incrementando el *ROI* (*Return on Investment*) del producto. Ésto se basa en la mejora de los ciclos de vida de lucro del producto aprovechando el desarrollo incremental para hacer *Releases* chicos con un subset de *features* vendibles.

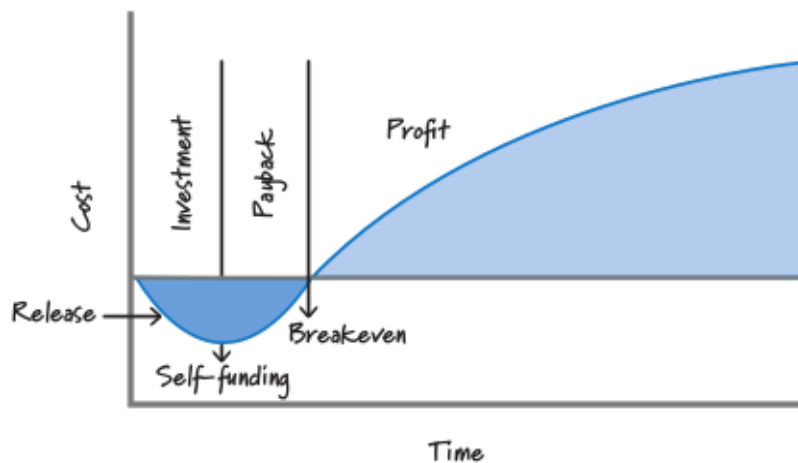


Figura 10: Economía *Single-Release* (Denne and Cleland-Huang, 2003)

Consideremos la economía de un producto con un único *Release* (con posteriores bugfixes). Existe una primera etapa de inversión, donde ocurre el comienzo del desarrollo. El *Release* ocurre en ésta etapa, mientras se sigue invirtiendo dinero. Cuando los ingresos del producto igualan sus costos de desarrollo, llegamos a la etapa de *self-funding*. Cuando ganamos más de lo que perdemos, entramos en la etapa de *payback*. Una vez que recuperamos todos los costos llegamos a *breakeven*, y todos los ingresos siguientes se consideran *profit*.

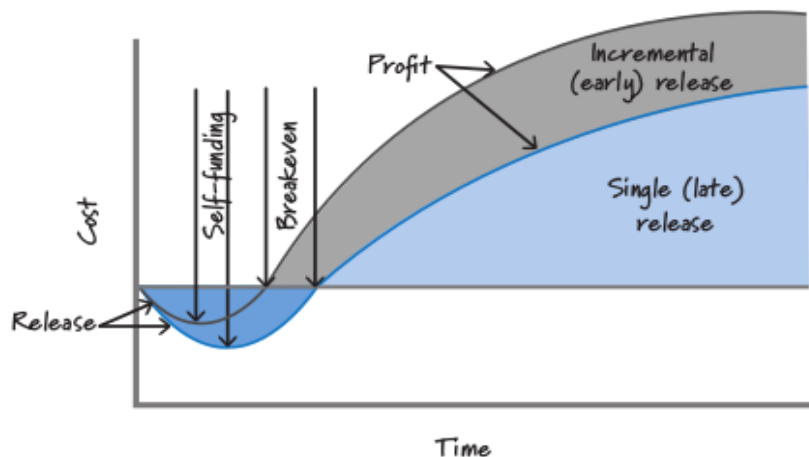


Figura 11: Economía *Multi-Release*

El gráfico anterior compara el producto antes visto con un producto *Multi-Release* (por ejemplo 2). Se puede ver que llegamos a *self-funding*, *breakeven* y *profit* antes, ya que los *Releases* son más chicos, incluyendo menos *features*. Ésto mejora el *ROI* del producto, pues tiene mejores ciclos de vida de lucro al particionar los costos de desarrollo y generar más oportunidades de venta con dos *Releases* con un subset de *features* vendibles menor.

Éste enfoque posee algunas limitaciones. Por cada producto, existe un conjunto mínimo de *features* que hacen al *Release* vendible. No se puede achicar infinitamente, pues se volvería tan pequeño que ninguna cantidad de Marketing podría atraer clientes a él. Además, en ciertos mercados, *Releases* más pequeños y frecuentes podría no tener sentido. Sin embargo, si el mercado ésta abierto a ésta opción, entregar *Releases* vendibles pequeños y frecuentes es un principio muy importante a seguir.

12. Planificación Multinivel

Cuando se desarrolla un producto bajo *Scrum* en una organización, hay diferentes niveles de planificación que entran en juego (como *Portfolio*, *Release*, etc). Veamos una tabla comparativa para cada uno de ellos como primer vista general del problema:

Level	Horizon	Who	Focus	Deliverables
Portfolio	Possibly a year or more	Stakeholders and product owners	Managing a portfolio of products	Portfolio backlog and collection of in-process products
Product (envisioning)	Up to many months or longer	Product owner, stakeholders	Vision and product evolution over time	Product vision, roadmap, and high-level features
Release	Three (or fewer) to nine months	Entire Scrum team, stakeholders	Continuously balance customer value and overall quality against the constraints of scope, schedule, and budget	Release plan
Sprint	Every iteration (one week to one calendar month)	Entire Scrum team	What features to deliver in the next sprint	Sprint goal and sprint backlog
Daily	Every day	ScrumMaster, development team	How to complete committed features	Inspection of current progress and adaptation of how best to organize the upcoming day's work

En los siguientes Capítulos 15,16,17 y 18 veremos cada uno de éstos en mucho más detalle por separado.

13. Portfolio Planning

Portfolio Planning se refiere a la actividad que determina qué items del *Portfolio Backlog* realizar, en qué orden y por cuánto tiempo. Éste ítem puede ser un producto, un incremento de un producto (*Release*) o un proyecto de la empresa, entre otros. El problema principal que suele haber es que la planificación a éste

nivel suele ir en contra de los principios ágiles, y a continuación veremos como se soluciona ese problema.

13.1. Resumen

13.1.1. Timing

Esta planificación es una actividad que no debería detenerse, siempre que tengamos nuevos productos o productos mantenidos en el tiempo. Si bien es de más alto nivel que la planificación de un producto específico, puede ocurrir que se utilice el conocimiento obtenido de dicha planificación para decidir si financiar el producto (si es propio) o tomar un proyecto.

13.1.2. Participantes

Como la planificación del *Portfolio* incluye productos nuevos y en proceso, posee varios participantes. Algunos de ellos pueden ser *stakeholders* internos, los *PO* de cada producto, líderes técnicos, etc. Los *stakeholders* internos deben tener una perspectiva del negocio lo suficientemente amplia como para poder tomar decisiones de los productos en proceso dentro de la empresa. Para facilitar éste proceso se pueden armar comités que supervisen este proceso de planificación.

Los *PO* participan al ser los representantes del producto que se va a construir, midiendo las expectativas y recursos esperados del producto. En ésta área, el input de líderes técnicos o arquitectos puede ser útil para tomar en cuenta limitaciones técnicas dentro de las decisiones de planificación.

13.1.3. Proceso

El proceso de *Portfolio Planning* tiene varios inputs y outputs asociados a su implementación. Los inputs son aquellos ya mencionados (productos en proceso y nuevos). Cada input viene con su propio set de datos. El nuevo producto viene con la información obtenida en la *Product Planning*, elementos como costo, duración, riesgo, etc. Los productos en proceso vienen con *feedback* del consumidor o cliente, niveles de deuda técnica, estimaciones de scope, costo actualizado, etc. Luego de obtener éstos inputs con datos, el proceso debería devolver outputs.

El primer output devuelto es el *Portfolio Backlog*, que contiene una lista priorizada de proyectos/productos futuros. El segundo es un set de productos *activos*, es decir productos aprobados para su producción y productos en proceso aprobados para continuar. Para llegar a éstos outputs, los participantes realizan 4 actividades, que veremos a continuación.

13.2. Estrategias de Planificación

En rasgos generales, la planificación del *Portfolio* asigna los recursos limitados de la organización para los productos de una forma económicamente sensible, siguiendo diferentes estrategias como las mencionadas posteriormente.

13.2.1. Optimizar para las Ganancias del Ciclo de Vida de un Producto

Para optimizar el ordenado de los productos en el portfolio, se deben decidir variables de evaluación para determinar si la optimización del mismo funcionó en el futuro. Una sugerencia⁷ es utilizar un *framework* económico donde todas las decisiones se gobiernan por las *lifecycle profits* (ó ganancias de ciclo de vida).

Para un producto, sus *lifecycle profits* representan la potencial ganancia del mismo a lo largo de su vida. Como estamos concentrados en mejorar ésta variable a nivel *Portfolio*, es probable que tengamos que suboptimizar determinados productos para maximizar las ganancias de todo el *Portfolio*⁸. Reinertsen menciona que dos variables muy importantes para medir el impacto de las *lifecycle profits* son el costo del retraso y la duración de un producto. En base a éstos parámetros, podemos armar una tabla de estrategias de *scheduling* sugeridas:

⁷Reinertsen 2009

⁸Poppendieck & Poppendieck 2003

(If) Cost of Delay	(And) Duration/Size	(Then) Scheduling Approach
Same across all products	Varies across products	Shortest job first
Varies across products	Same across all products	High delay cost first
Varies across products	Varies across products	Weighted shortest job first

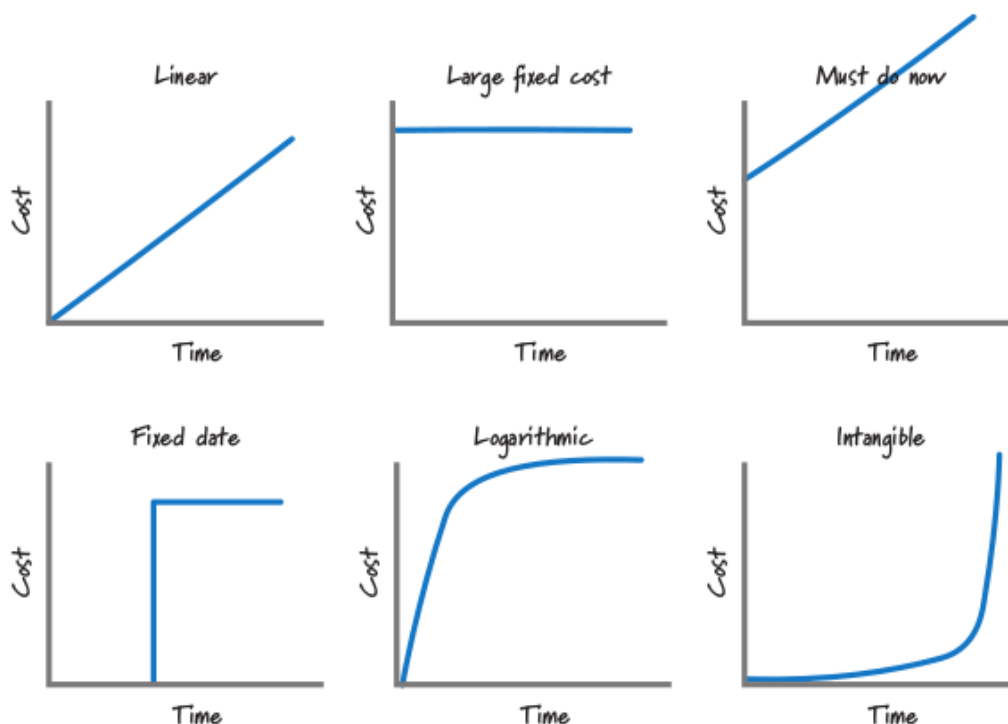
Figura 12: Estrategias con $weighted\ shortest\ job\ first = cost\ of\ delay / duration$.

13.2.2. Calcular el Costo del Retraso

Al secuenciar ítems en el *Portfolio Backlog*, estamos decidiendo trabajar en algunos productos antes que otros. Los productos que quedan para más adelante tienen un comienzo retrasado y una fecha de entrega final retrasada, para las cuales existe un costo. Si bien muchas organizaciones solo hacen los productos que le dan más ganancias primero sin mirar más -por ejemplo concentrándose en el *ROI* de cada producto- el costo de retraso debería ser tomado en cuenta (por ejemplo un producto puede tener un *ROI* menor pero un costo de retraso tan alto que resulta conveniente trabajar en él primero). En general, analizar el costo del retraso nos indica que, a un mayor nivel, el tiempo afecta múltiples variables como costo, beneficio, riesgo, etc. Un modelo⁹ propuesto para calcular el costo del delay consiste en el conjunto de 3 atributos del producto:

- *User Value*: Valor en los ojos del usuario.
- *Time Value*: Como ese valor decae con el tiempo.
- *Risk Reduction/Opportunity Enablement*: Valor de mitigar un riesgo o explotar una oportunidad.

A cada uno de éstos parámetros se les asigna un número en escala del 1 al 10, y la suma de esos 3 es el costo del retraso. En el caso de que calcular números precisos suponga un costo bloqueante, una alternativa puede ser caracterizar el perfil del costo del retraso, por ejemplo con los siguientes tipos de funciones (con significados asociados):



- **Lineal**: costo de retraso que crece de forma constante.

⁹Leffingwell 2011

- **Gran Costo Fijo:** producto que deviene un gran costo de retraso si no se actúa pronto.
- **Debe Hacerse Ahora:** costo alto de retraso que crece de forma inmediata y agresiva.
- **Fecha Fija:** no tiene costo de retraso hasta que llegue la fecha de entrega fija, momento en el cual se adquiere todo el costo inicial planificado.
- **Logarítmica:** se adquiere la mayoría del costo al comienzo, con un posterior incremento muy bajo.
- **Intangible:** producto que **parece** no tener costo aparente por un período de tiempo, y “de repente” adquiere un costo altísimo de retraso. Ejemplo: como muchas empresas tratan a la deuda técnica.

13.2.3. Estimar para Precisión, no Exactitud

A la hora de estimar cada ítem del *Portfolio Backlog*, se busca la mayor precisión posible sin exactitud, dada la limitada cantidad de datos que se suelen tener al momento de realizar la primera estimación y los costos asociados a estimar de forma exacta. En vez de usar números (que indican exactitud), se sugiere como medida alternativa utilizar talles de remeras.

Se utilizan los talles **XS, S, M, L y XL**. Cada talle corresponde a un rango de costo creciente, por ejemplo S podría significar un costo entre \$10k y \$25k y L entre \$125k y \$350k. El beneficio de usar talles es que es una estimación rápida, lo suficientemente precisa, y provee información accionable al nivel del *Portfolio*.

13.3. Estrategias para el *Inflow* (Flujo de Entrada)

13.3.1. Aplicando el Filtro Económico de la Empresa

Luego de realizar la planificación a nivel de un Producto determinado, se obtienen ciertos datos del mismo que pueden ser usados como *input* para la planificación del *Portfolio*. Basándose en éstos datos, la organización debe decidir si aprueba o rechaza un producto pasándolo por su “filtro económico”. Si bien el filtro dependerá de cada organización en particular, la idea es que el mismo identifique rápidamente las oportunidades donde las ganancias de un producto superan ampliamente a los costos de desarrollarlo. Suponiendo que una organización no posea circunstancias extraordinarias, en general se suelen tener varias oportunidades de buen rendimiento, por lo que aquellas que no cumplan una ganancia base deberían ser descartadas sin pensar demasiado. Un buen filtro además sirve para regular la entrada de productos al *Backlog*, pues en épocas de saturación basta con aumentar su estándar para bajar la cantidad de entradas.

13.3.2. Balancear las Fechas de Entrada y Salida

Idealmente, se busca que los productos entren y salgan del *Portfolio Backlog* a una velocidad constante y equivalente. Suele ser común que las organizaciones hagan reuniones con las personas de mayor renombre para decidir la dirección estratégica a tomar en el siguiente año fiscal. Es aquí donde se deciden los productos, y durante ésta reunión se sobrecarga el *Portfolio Backlog* con todos los productos futuros. Si bien las reuniones estratégicas son necesarias, decidir todo con tanta antelación y potencial incertidumbre va en contra de unos de los principios de *Scrum*: “Dejar opciones de planificación abiertas hasta el último momento responsable”.

Además, se viola el principio de utilizar tamaños económicamente sensibles para los lotes -los productos-. Procesar tantos elementos es muy costoso y genera un potencial desperdicio muy grande (por la incertidumbre de planificar hasta un año atrás). Para combatir éste problema, se pueden introducir productos al *Portfolio* de forma incremental y recurrente, por ejemplo bi-mensual en vez de anual. Éste cambio provee estabilidad y predictabilidad al *Backlog*, además de reducir el costo y esfuerzo necesario para priorizarlo.

Además, se debería concentrar la atención en productos más pequeños y frecuentes (similar a lo sugerido con *Releases*). Ésto significa una alta cantidad de productos terminados sacados del *Backlog*, dejando lugar para los siguientes del pipeline de forma eficiente.

13.3.3. Aprovechar Oportunidades Emergentes Rápidamente

La planificación del *Portfolio* debe aprovechar oportunidades emergentes -aquellas previamente desconocidas o que en algún momento se descartaron- lo más rápido posible. Por ejemplo, un producto internacional puede estar sujeto a la jurisdicción de cada localidad en particular, por lo que es importante que si una jurisdicción cambia para admitir al producto, ésta información se aproveche lo antes posible para no entrar en costos de retraso. Algunas oportunidades emergentes pierden su valor muy rápido, por lo que es conveniente analizar éstos casos de forma recurrente.

13.3.4. Planear *Releases* más Pequeños y Frecuentes

Además de las razones detalladas en el capítulo de los Principios de Planificación, hay otra razón para favorecer ésta medida: evitar el efecto *convoy*. Éste representa el embotellamiento de productos del *Backlog* que impide a varios productos pequeños avanzar por un producto amasador de recursos que se encuentra delante. Recordemos que todos esos productos sin realizar podrían estar acumulando costos de retrasos, por lo que se ve claramente que los productos monolíticos tienen un alto riesgo de bajar las ganancias del ciclo de vida de todo el *Backlog* (a no ser que su ganancia sea sumamente extraordinaria).

Otra contra es la evidente dependencia de un producto, por lo que si algo negativo sucede con él, toda la organización podría venirse abajo, o se podrían tomar decisiones sólo para el bien de ese producto y no de la organización. Una manera de solucionar esto es colocando una cota para cuanto esfuerzo puede llevar un *Release* de un producto. Si nuestra cota son 8 meses y hay un *Release* de 25 meses, esto nos fuerza a dividirlo y favorecer *Releases* más pequeños y frecuentes.

13.4. Estrategias para el *Outflow*

13.4.1. Disminuir el Trabajo Ocioso en vez de Trabajadores Ociosos

Es común que en las organizaciones se saque el producto del tope del *Backlog* y se asignen trabajadores al mismo hasta que todos estén al 100 % de su capacidad, lo cual ralentiza el trabajo del producto y introduce errores asociados a la sobrecarga laboral. Una mejor estrategia es sólo comenzar un producto cuando se tiene el equipo de *Scrum* disponible, habiendo asegurado un ritmo de trabajo sostenible en el tiempo (no explotar al equipo sólo para cumplir con fechas ridículas impuestas por el cliente). Además, se debe tener la certeza de que el flujo del nuevo producto no entrará en conflicto con los otros proyectos de la empresa.

13.4.2. Establecer Límite para el *WIP*

Nunca deberíamos sacar más productos del *Backlog* de los que podemos completar/mantener. Hacer esto reduce la capacidad de cada producto generando costos de retraso y una reducción en la calidad de cada uno. Como vimos antes, para limitar el *WIP* no es conveniente utilizar la cantidad de personas disponibles, pero si la cantidad de equipos de *Scrum* de la organización (además de la información de que productos puede encarar cada equipo).

Entonces, en vez de limitar la cantidad de productos a trabajar en simultáneo por “la cantidad de *devs*” o “la cantidad de *UX's*”, se limita por “la cantidad de equipos de *Scrum* disponibles y los tipos de proyecto que pueden afrontar”. El equipo de *Scrum* se convierte en la unidad de capacidad que establece el límite del *WIP*.

13.5. Estrategias Dentro del Proceso

Éstas estrategias nos informan si es apropiado seguir con el producto, entregarlo, iterar o terminarlo, y

en general suelen ser recurrentes (al final de cada *Sprint* o *Release*).

13.5.1. Usar Economías Marginalistas

Desde un punto de vista económico, todo el trabajo hecho en el producto hasta el momento de la decisión actual son “costos adquiridos”. Sólo nos interesan las consecuencias económicas de dar el siguiente paso, por ejemplo si conviene seguir invirtiendo o terminar un producto. Éste foco económico en la última unidad a producir sin considerar lo anterior se conoce como economía marginalista. En general, el flujo de decisión de economías marginalistas se concentra sobre la relación costo/beneficio actual del producto, y debería seguir el siguiente esquema:



Figura 13: Flujo de decisión basado en marginalismo.

14. Product Planning

14.1. Resumen

Esta etapa se centra en realizar un plan a grandes rasgos de cómo llevar a cabo un producto o futura versión de uno ya existente, teniendo solo la “idea” del mismo al comienzo. Bajo *Scrum*, sabemos que no podemos realizar toda la planificación de un proyecto por adelantado, por lo que esta etapa genera los planes suficientes para poder pasarlo por *Portfolio Planning* antes de su eventual comienzo. Queremos llegar rápidamente a la etapa de *Sprints*, porque eso implica el pasaje del momento “creemos que sabemos las necesidades del cliente”, al entorno donde recibimos *feedback* de forma rápida, basado en la realidad sobre la cual nuestro producto debe desenvolverse.

14.1.1. Timing

Esta planificación (también llamada *Envisioning*) es una actividad recurrente, por ejemplo para cada *Release* de un producto. Comienza con simplemente una idea, la cual es pasada por el filtro estratégico de la empresa, para asegurar la consistencia con la dirección de la organización. Si es aprobado, se pasa a la etapa de *Envisioning*.

En esta etapa, se decide que *Features* podrían componer un **mínimo** primer -si se trata de un nuevo producto- *Release*. La idea de la minimalidad es entregar algo a los usuarios lo antes posible para obtener *feedback* sobre el cual podemos actuar y decidir si mantener la visión actual o pivotear la solución si es necesario, de forma eficaz.

14.1.2. Participantes

El único participante **requerido** es el *Product Owner*. En general, el *PO* realiza un *Envisioning* inicial junto a stakeholders internos del producto y especialistas del área de interés (marketing, user experience, etc). Idealmente, el equipo que estará realizando las tareas de valor en el *Sprint* debería participar en este ritual, pero de una forma mas pasiva (con el objetivo de interiorizar la visión del *PO*). Desafortunadamente, es común que el equipo de *Scrum* se arme después del *Envisioning* inicial. Aún así, es importante que ellos estén presentes para la planificación de los futuros *Releases*.

14.1.3. Proceso

El input principal para el *Envisioning* inicial es una idea que ha superado el filtro estratégico. En cambio, el input para un *Reenvisioning* es una idea pivotada. Es decir, una idea que fue actualizada con el *feedback* de los clientes o alguna razón de mercado, como cambios fuertes en los competidores. Claramente, necesitamos otros inputs también, como por ejemplo:

- **Horizonte de planificación:** que tal lejos en el futuro deberíamos considerar al hacer *Envisioning*.
- Cuando esperamos completar las actividades de *Envisioning*, y qué recursos tenemos para hacerlo.
- **Umbral de confianza:** la *Definition of Done* del *Envisioning*. Representa la información necesaria para poder decidir si financiar o no el proyecto, en la capa de *Portfolio Planning*.

14.2. Ejemplo Empresa ReviewEverything: *SmartReviewForYou* (SR4U)

El *Management* de la empresa designa a Roger -interno especialista en marketing- como *PO* de su nuevo filtro para un sistema de reviews online. Éste producto posee el objetivo de mostrar aquellas calificaciones que sean de mayor utilidad para el usuario final. Sólo se tiene una descripción enviada por marketing, describiendo audiencia esperada, ventajas principales y *features* de alto nivel. Para el *Envisioning* se sumarán *stakeholders*, especialistas en algoritmia de filtrado y un investigador de mercado. El equipo debe producir lo siguiente para que los de *Management* puedan responder si seguir adelante con el desarrollo inicial del producto:

- Visión inicial del producto, su *Backlog* y su *Roadmap*.
- Validación de la suposición que indica que los usuarios prefieren los resultados filtrados por SR4U frente a los resultados comunes.
- Descripción de las suposiciones acerca de los potenciales usuarios y el producto, que éste debe testear en el primer *Release*. Con esto, proveer métricas accionables para testear suposiciones y ver si SR4U cumplió las expectativas.

14.2.1. Visión

La primer tarea del equipo es desarrollar una visión en común del producto, **en términos simples**. Ésta suele ser expresada en términos de **cómo los *stakeholders* obtienen valor**, según áreas generales:

- **(CE) Condiciones de Entrada:** establecer paridad con la competencia, respetar normas legales (i.e HIPAA), entregar *features* requeridas.
- **(H) Habilitación:** entrar a nuevo mercado o hacer posible ventas de otros servicios.
- **(D) Diferenciador:** diferenciarse de la competencia, sorprender al consumidor.
- **(S) Spoiler:** eliminar el diferenciador de la competencia, redefinir reglas del mercado, subir el estándar.
- **(RC) Reductor de Costo:** bajar el TTM¹⁰, mejorar márgenes de ganancia o el expertise, o reducir la cantidad de personal necesario para la tarea o su cantidad de tiempo.

Para SR4U, creemos que existen 3 áreas principales. Primero, se debe reducir el tiempo que les toma a los usuarios buscar reviews (RC). Además, SR4U debe hacer que los usuarios sientan que el servicio ha sido sorprendente para realizar una compra informada (D). Finalmente, queremos que SR4U suba el estándar para los servicios de review online, creando caos y expectativas altas para los competidores (S).

La visión finalizada debe ser capaz de superar el *Test del Ascensor*¹¹. El formato la visión puede ser variado, pero debería poder responder a preguntas básicas como “**Para quién es**”, “**De qué categoría es el producto**”, “**Su beneficio y diferenciador principal**” y “**Por qué es mejor que la competencia**”. Un formato sugerido es el siguiente (sujeto a adaptaciones para hacerlo encajar a cada situación):

For	<u>(target customer)</u>
Who	<u>(statement of need or opportunity)</u>
The	<u>(product name)</u> is a <u>(product category)</u>
That	<u>(key benefit, reason to buy)</u>
Unlike	<u>(primary competitive alternative)</u>
Our product	<u>(statement of primary differentiation)</u>

¹⁰ “In commerce, time to market (TTM) is the length of time it takes from a product being conceived until its being available for sale.”

¹¹ La habilidad de poder explicar el proyecto a alguien en 1 minuto o menos, en teoría lo equivalente a un viaje de ascensor.

14.2.2. Creación de un *Product Backlog* de Alto Nivel

En el paso siguiente, debemos crear los *PBI's* de alto nivel (*epic*) que se alineen con la visión propuesta y sean de uso para el equipo de *Scrum*. Se involucran las mismas personas que en la visión, solo que ésta vez es *requisito* que el equipo de desarrollo participe, o en el caso de que aún no esté armado, que participe gente de perfil técnico con interés en el producto.

En SR4U, al no estar el producto aprobado, todavía no tiene designado un *devteam*. Entonces, se le pide a Yvette -una arquitecta experta- que participe. De la reunión, surgen las siguientes *epics*:


- Como usuario típico quiero poder enseñarle a SR4U que tipos de reviews ocultar para que sepa que características usar a la hora de descartar reviews en mi lugar.
- Como usuario típico quiero una simple interfaz parecida a Google a la hora de realizar una búsqueda de reviews para así no perder mucho tiempo describiendo lo que quiero.
- Como usuario experto quiero decirle a SR4U que sitios ignorar para que pueda descartar reviews de lugares que no son de mi interés.
- Como un vendedor quiero poder mostrar un resumen de reviews para mi producto marcado por SR4U para que los consumidores puedan visualizar la opinión del mercado sobre mi producto.

14.2.3. Definición del *Roadmap* del Producto

El *roadmap* es una **serie de *Releases*** del producto que llevarán nuestra visión a cabo. Como queremos concentrarnos en *Releases* pequeños y frecuentes, cada uno solo tendrá el conjunto mínimo de *features* necesarias para cumplir con expectativas de calidad para el consumidor (también llamado *MVP*). Como complemento, algunas organizaciones hacen *Releases* estáticos (ej 1 c/4 meses) para simplificar el *roadmap*, con cuidado de no poder deadlines imposibles. Éste enfoque es simple y provee a todos con *Releases* predecibles y fáciles de sincronizar.

Cada *Release* debería tener un **claro objetivo** que comunique lo que se espera del mismo. Es importante que todo el *roadmap* cubra a la potencial segmentación de usuarios del mercado al que el producto apunta. En SR4U, la audiencia esperada son usuarios interesados en leer reviews antes de comprar un producto o servicio. Luego, se los separó en “usuarios típicos” y “usuarios expertos” según el grado de control que se les permite tener. Con ésta información, se decidió que el *Release* inicial apunte a usuarios típicos. El equipo de *envisioning* también ve como potenciales usuarios a vendedores que quieran tener éstas reviews en su sitio como marca de calidad, pero primero para eso se deberá establecer la marca en el mercado.

Al hacer el *roadmap*, también se debe tener en cuenta qué plataformas -web,iOS, etc- se quiere atacar y cuándo. En general, debería representar una primera **aproximación** de cuando deberían hacerse los *Releases*, considerando todos los factores que son relevantes para nuestra solución. Recordando que se esta bajo *Scrum*, se debe planear el *roadmap* hasta un futuro razonable por su naturaleza iterativa. SR4U requiere financiación por 9 meses, por lo que el *roadmap* cubrirá esa franja de tiempo. Además, se hará coincidir algunos *Releases* con eventos y expos para mejorar visibilidad. Notar que el tercer *Release*, al estar tan lejos del presente, no se lo intenta hacer coincidir con nada, pues aún hay incertidumbre al respecto:



	Q3—Year 1	Q4—Year 1	Q1—Year 2
Market map	Initial launch	Better results More platforms	Sophisticated users
Feature/benefit map	Basic learning Basic filtering	Improved learning Complex queries	Define sources Learn by example
Architecture map	100K concurrent web users	iOS and Android	Web services interface
Market events	Social Media Expo	Review Everything User Conference	
Release schedule	1.0	2.0	3.0

14.2.4. Otras Actividades

La tarea de *Envisioning* puede incluir otras actividades que ayudan a llegar al nivel de confianza requerido: análisis de mercado, modelo básico de negocios para el producto, entre otros. Algunas organizaciones dividen el *Envisioning* en *Sprints* cortos con tareas a realizar. Éste es el caso de SR4U, que decide hacer 1 *Sprint* de 1 semana de adquisición de conocimiento, en la cual se corroborará la suposición de que los usuarios prefieren las reviews filtradas por SR4U que las normales.

En éste *Sprint*, el equipo hará una página simple en HTML donde un grupo de usuarios hará una consulta por reviews. A los dos días se les enviarán dos resultados: las reviews sin filtrar y las reviews filtradas (ellos no sabrán cuales fueron removidas). La idea no es realizar la tecnología de filtrado automático, sino que los especialistas en técnicas de filtrado filtren a mano y le envíen una lista curada de reviews a los usuarios, de forma tal que se pueda validar la suposición previamente hecha. La idea no es testear la automatización, sino testear la suposición de que la gente prefiere una lista curada de reviews frente a la alternativa usual.

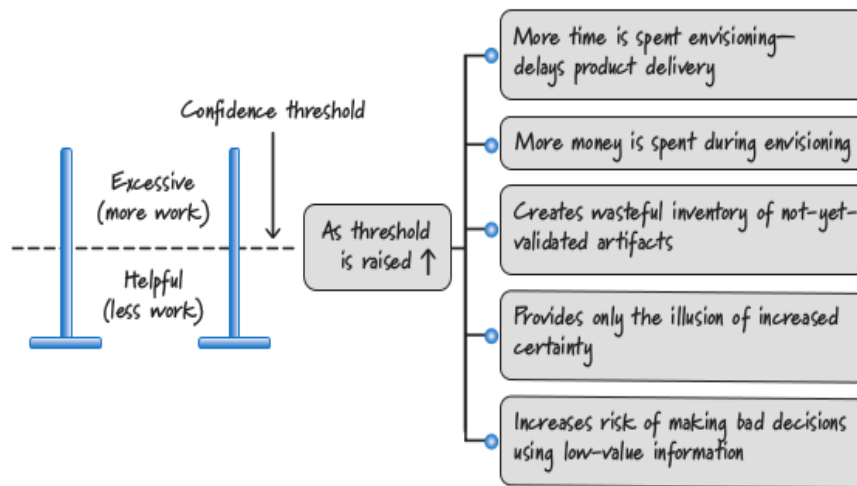
14.3. *Envisioning* Económicamente Sensible

El *Envisioning* debería ser llevado a cabo de una forma económicamente sensible y alineado a *Scrum* (es decir mantenerlo simple). Representa una inversión para darle al *Management* la información necesaria para poder decidir si financiar/seguir adelante con un producto o descartarlo. Con poco *Envisioning* el objetivo no se cumple, y con demasiado pasaríamos a tener un largo inventario de productos de ésta planificación que tal vez necesiten una vuelta de tuerca cuando empezemos a adquirir conocimiento validado en los *Sprints*.

Para también alinearlos con *Scrum*, asumimos que no podremos tener todos los requerimientos y un calendario exacto al final del *Envisioning*, pues ésto implicaría aplicar un desarrollo secuencial como base de nuestro producto ágil, con todas las contras ya vistas. Es mejor hacer la cantidad justa de planificación predictiva y tareas de adquisición de conocimientos asociada al nivel de riesgo, tamaño y naturaleza del producto. Lo demás siempre puede hacerse justo a tiempo, similar a los ciclos de *Sprint*, lo cual ayuda a adoptar la noción de cambio del producto.

14.3.1. Apuntar a un Umbral de Confianza Realista

El umbral de confianza necesario para tomar la decisión de seguir adelante o no con un producto debería ser realista. Una vez superado éste umbral, se puede construir el producto y validar suposiciones claves acerca del mismo, por lo que tenerlo muy alto puede traer consecuencias económicas. Mientras más alto está, más tiempo se requiere para superarlo, lo cual implica dinero extra para *Envisioning* y delays en los potenciales *Releases* del producto (el cual ya vimos, tiene un costo propio). También existen múltiples otras contras relacionadas a la obtención de información no validada dentro del *Envisioning*, y como el cambio natural del producto podría invalidar lo obtenido luego del producto. Todas las contras pueden resumirse en este gráfico:



14.3.2. Concentrarse en Horizontes Cortos, Actuando Rápido

Al hacer *Envisioning*, el foco debería estar en el set de *Features* obligatorias del primer candidato a *Release*. Planear demasiado en el futuro aumenta el riesgo de estar planeando para cosas que jamás ocurran. Además, si nuestro producto es innovador, todavía faltaría validar varias suposiciones, por lo que es probable que cuando se lo someta a situaciones reales con consumidores ganemos información importante para adaptar el producto a una visión más realista.

Como vimos, el objetivo está en llegar a la etapa de validación, por lo que el proceso de *Envisioning* debería ser rápido y eficiente. Una manera de asegurar que no se pierda tiempo en ésta etapa es colocar una fecha **estimada** de finalización para el *Envisioning*, tomando en cuenta los factores mencionados previamente (por ejemplo, si el producto es innovador se va a necesitar más tiempo que un *Release* común y corriente).

14.3.3. Invertir en Aprendizaje Validado

Todas las actividades de *Envisioning* deberían ser evaluadas y financiadas bajo 3 factores: como contribuyen a la adquisición de aprendizaje validado, y como contribuyen al set de *Features* y el producto en general. En la medida de lo posible, es mejor pagar por obtener alguna métrica para validar alguna de las suposiciones principales ahora antes que luego de invertir en uno o más *Releases*. Actividades con bajo impacto en éstos puntos deberían ser consideradas con cuidado.

Por ejemplo, las actividades predictivas con un alto grado de incertidumbre generan poco valor y son potencialmente inútiles a largo plazo, por el cambio natural que sufre un producto en su ciclo de vida. Además, ésta información puede nublar el juicio al hacer parecer que se tiene un mapa completo del producto, ocasionando que se tomen decisiones importantes antes de tiempo bajo la ilusión de la certeza.

En el caso de SR4U, el *Backlog* y el *Roadmap* representan información con incertidumbre, por lo que se debe tener cuidado en no detallar demasiado todos los *Releases*, pues los contenidos de ambos están sujetos al cambio a medida que el *devteam* adquiere conocimiento en la etapa de desarrollo. En el *Envisioning*, se decidió pagar para obtener validación de la suposición que indica que los usuarios prefieren las *reviews* curadas (se invirtió en tiempo de *Envisioning*).

14.3.4. Usar Financiación Provisoria/Incremental

La financiación del producto debería ir de la mano con la generación de elementos de planificación del *Envisioning*. Como ya discutimos, queremos planificar solo lo justo y necesario, por lo que la financiación del producto debería poderse reajustar a medida que se va aprendiendo y validando lo construido. No tendría sentido planificar para el primer *Release* y en base a eso financiar todo esfuerzo futuro del proyecto. En general, es mejor financiar el desarrollo hasta la siguiente situación crítica de validación y potencial pivoteo, momento en el cual se reevalúa (por ejemplo, de a *Releases*). Como es evidente, financiar de forma incremental va de la mano con reducir el *scope* del *Envisioning* a lo justo y necesario, sin generar excedente

potencialmente desperdiciable.

Para que todo ésto tenga sentido conviene pensar al *Envisioning* como un ciclo de “Aprendizaje y Pivoteo Veloz”. Queremos realizar las tareas de forma eficiente llegando rápido a la validación del aprendizaje para poder pivotear y adaptar nuestra solución, mejorando el producto de forma incremental.

15. *Release Planning*

15.1. Resumen

El *Release Planning* responde a las preguntas del estilo “*Cuándo estará terminado?*” o “*Cuánto costará el desarrollo?*” balanceando valor para el cliente y calidad del producto contra limitaciones del *scope*, planificación y presupuesto.

Una de las cosas que se debe decidir es la cadencia de los *Releases*. Algunas organizaciones aprovechan el estado *Potentially Shippable* al final de cada *Sprint* para hacerlo en ese momento, mientras otras prefieren juntar varias *features* o hacer un *Release* por cada *feature* (también llamado *continuous deployment*).

15.1.1. Timing y Participantes

Esta actividad es una recurrente por *Sprint* que permite determinar el siguiente paso para lograr el objetivo del producto (viene después del *Product Planning* claramente). En particular, se suele hacer dentro de la *Sprint Review*, revisando como se está encaminando hacia el *Release* después de la demo.

Antes de comenzar un *Release*, se suele hacer un plan preliminar para el mismo. Al desarrollar un nuevo producto, éste plan no será preciso, pues la idea es que con el aprendizaje validado del producto a lo largo de los *Sprints* se vaya actualizando para reflejar mejor la realidad.

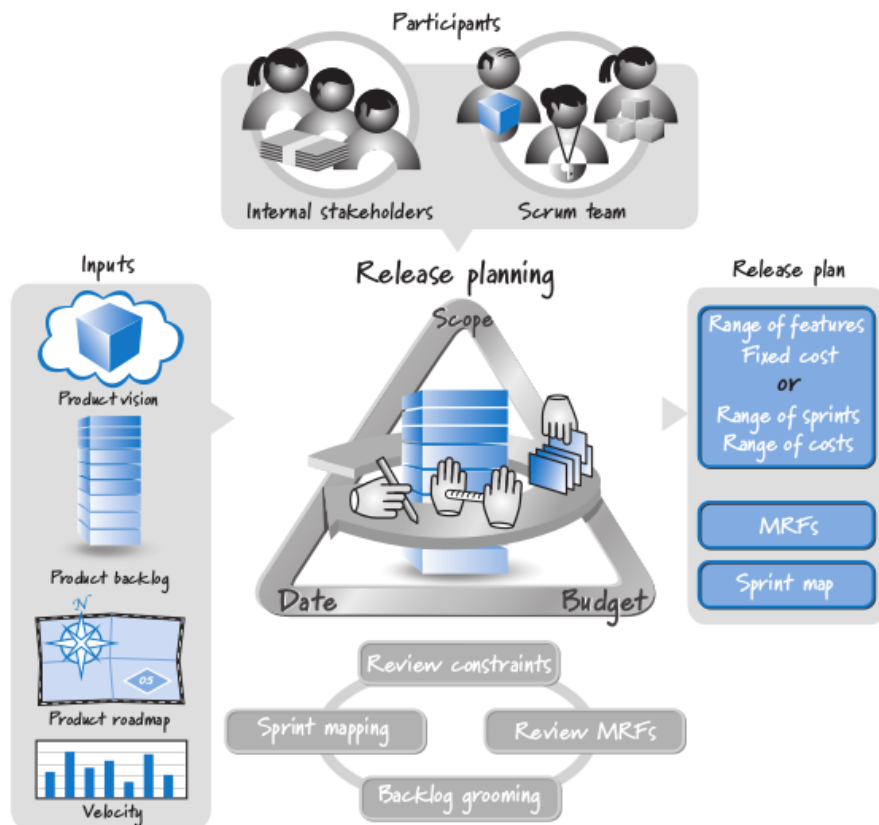
El *Release Planning* involucra al equipo de *Scrum* y los *stakeholders* o *Product Owner*. Se espera que los *stakeholders* participen al comienzo del *Release*, y más adelante el seguimiento se haga vía *PO* en las *Sprint Reviews*.

15.1.2. Proceso

Éste proceso recibe como input el *Product Planning* -la visión del producto, el backlog de alto nivel y el roadmap-, además de la velocidad del equipo. Con ésto se realizan actividades recurrentes como revisar la fecha, el *scope* y el presupuesto para el *Release* a lo largo de los *Sprints* y ajustarlo en consecuencia. Además, durante la *Sprint Review* ésto enlaza con el *Grooming* a realizar, pues un ajuste en el *Release* suele implicar el refinado/aparición/eliminación de *PBI's*. Éstas actividades suelen realizarse en la planificación inicial, con revisiones en los sucesivos *Sprints*.

Cada *Release* debería tener un set mínimo de *features* (llamado *MRF*). Ésto se puede obtener en *envisioning* y refinar en ésta etapa para asegurarse que realmente representen el producto mínimo del lado del cliente/consumidor. El *Release Planning* también suele tener como output un *Sprint Map*, el cual indica en cuál *Sprint* **cercano** se afrontarán *PBI's* de interés, sirviendo como visualización del futuro cercano del producto.

Los outputs de ésta planificación se conocen como *release plan*. Éste plan comunica, con un nivel de precisión razonable, donde se está parado en éste momento, cuando se termina, cuanto costará y el set de *MRF's* mapeados con el *Sprint Map*. Toda ésta información se puede resumir en el cuadro a continuación, que detalla el proceso completo de *Release Planning*:



15.2. Restricciones del *Release*

Un *Release* puede tener diferentes restricciones: presupuesto, fecha, *scope*, etc. Las restricciones de cada proyecto no son prefijadas para siempre, sino que deberían evaluarse a medida que se va avanzando hacia el *Release*. Si a medida que se avanza hacia el *Release* sale a la luz que el set de *MRF's* es muy grande y hay una restricción de fecha, tendría sentido bajar los *MRF's*, flexibilizar la fecha, o sumar presupuesto (para tercerizar o incrementar el equipo si hay suficiente tiempo en el *Release*).

15.2.1. Fixed Everything

Bajo una planificación tradicional y fuertemente predictiva, se puede asumir que los requerimientos se conocen de entrada y nunca cambiarán en *scope*, por lo que se puede hacer un plan detallado y completo de las funcionalidades, obteniendo estimaciones de costo y *schedule*. En *Scrum*, creemos que la suposición fundamental en la que se basa la planificación tradicional es falsa, por lo que éste tipo de restricción nunca podría funcionar. *Scrum* requiere al menos una variable flexible de las 3 -*scope*, fecha o presupuesto-. Además, cuando éstas variables se encuentran fijas, se hace por defecto flexible la variable de “calidad” del proyecto.

15.2.2. Fixed Scope & Date

Este enfoque sólo deja flexible el presupuesto. El problema clave de éstas restricciones es que la combinación fija de variables es tan difícil de predefinir, que suele ocurrir que una termine cediendo al final.

Llega un punto en el que el tiempo se acaba, y si ninguna variable se flexibiliza se termina cayendo en una deuda técnica enorme con grandes consecuencias para futuros *Releases*. Ésta planificación asume, equivocadamente, que incrementar la cantidad de recursos dados a un proyecto aumentará inmediatamente la cantidad de trabajo terminada. Si bien ésto es cierto en casos específicos (por ejemplo si se terceriza cierta parte del producto para destinar recursos internos a otras áreas), comprar tiempo o *scope* tiene un límite corto y real.¹²

¹² “Nine women do not make a baby in a month” Brooks 1995

La realidad es que un presupuesto flexible significa agregar más gente¹³ o que la misma gente trabaje más horas, violando el principio de *Scrum* de desarrollo a un ritmo sustentable. Si se encuentra un proyecto con ésta característica, el desarrollo iterativo debería permitir identificar éstos problemas con el tiempo suficiente para intentar flexibilizar alguna variable y evitar el desastre.

15.2.3. Fixed Scope

Éste modelo asume más prioritario el *scope*, por lo que se deja abierta la posibilidad de extender la fecha de terminación en caso que no se completen los *MRF's*. Un punto importante a considerar es que, si la organización trabaja con una importante estructura multinivel de *Scrum* (como por ejemplo un *Release Train*) que requiere coordinación, mover la fecha puede ser muy disruptivo para los otros grupos. Éste escenario podría reflejar que el *scope* es demasiado grande, lo cual es solucionable haciendo *Releases* más pequeños y frecuentes, con fecha fija.

15.2.4. Fixed Date

Éste enfoque es considerado uno de los más alineados con los principios ágiles. Podemos fijar la fecha y el presupuesto, pero manteniendo el *scope* flexible. Para realizarlo de forma adecuada, es crucial realizar las *features* en orden de prioridad y tener un *MRF's* realista, de forma tal que al acabarse el tiempo se tengan las *MRF's* listas y lo no terminado corresponda a *PBI's* de poco valor. Las *features* principales deben estar completamente terminadas (con *Definition of Done*), asociando el potencial *WIP* con áreas no fundamentales.

15.3. Tareas de *Grooming*

Durante el *Product Planning*, se produce un *Product Backlog* de alto nivel -*epics* o *themes*- que es utilizado para definir el *MRF* del *Release*. Éstos *PBI* son demasiado grandes como para entrar trabajar en ésta etapa, por lo que es crucial reducirlos a tamaños más manejables con ayuda de técnicas como *User Story Writing Workshop* o *Story Mapping*. Con las *stories* asociadas a partir de los *MRF's*, el equipo puede estimar y empezar a construir el mapa de *Sprints*.

15.4. Refinando *MRF's*

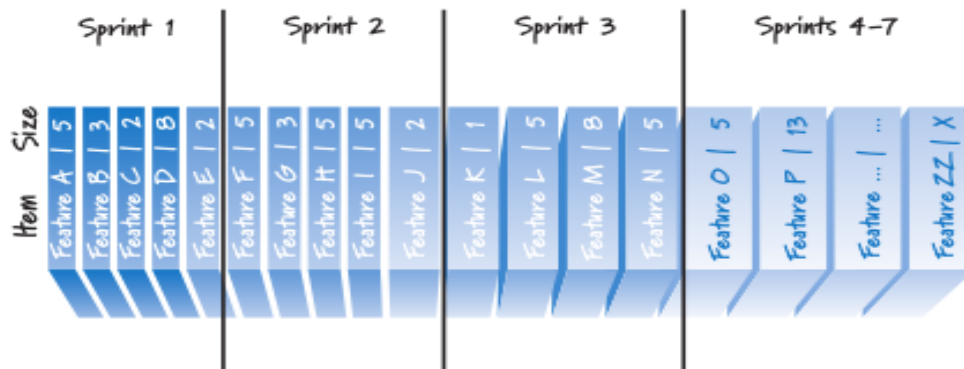
Sabemos que los *MRF's* son las *Features* que deben estar en el *Release* para cumplir con expectativas de valor y calidad para el cliente. Una de las tareas del *Release Planning* es monitorear y refinar aquello que constituye el *MRF's* con el *feedback* obtenido en cada *Sprint*. En *Scrum*, el *PO* es el encargado de definir este set, en colaboración con el equipo de desarrollo y los *stakeholders*. Dependiendo de si hay restricciones de presupuesto o no, éste set puede definirse teniendo en cuenta el costo económico de los mismos, realizándolo paralelo al *Grooming* para tener mejores estimaciones de éste costo.

Es interesante preguntarse porque no se usa un set maximal en vez de uno minimal. La respuesta es que el set maximal probablemente es el que requiera más tiempo, dinero y riesgo, y el set minimal nos permite alinearnos con el principio de entregar *Releases* más pequeños y frecuentes.

15.5. *Sprint Mapping* (ó *PBI Slotting*)

Si bien el equipo y el *PO* no deciden que *PBI's* hacer en el *Sprint* hasta el *Sprint Planning*, puede ser útil hacer un mapeo inicial de ítems que se harán en los primeros *Sprints*. Para realizarlo necesitamos un *PB* priorizado y estimado, y ya con eso podemos mapear utilizando la velocidad del equipo (si éste dato no se tiene se puede realizar un *Sprint* sin éste dato para obtenerlo). Si, por ejemplo, la velocidad fuese 20, se podría obtener el siguiente mapeo (también funciona de forma vertical):

¹³“Adding manpower to a late software project makes it later” Brooks 1995



Al trabajar con un sólo equipo, éste mapeo inicial puede terminar reorganizando el set inicial para asegurarse de tener un *PSP* al final de cada *Sprint*. A la hora de desarrollar con múltiples equipos, se puede querer hacer un mapeo más a futuro bajo técnicas como *Rolling Look-Ahead Planning*¹⁴ que planifican más allá del siguiente *Sprint* (con precisión a nivel *US* en vez de *Task*) para evitar problemas de dependencias.

Es importante recordar que éste mapa probablemente se verá modificado en el *Sprint Planning*, por lo que muchas organizaciones -especialmente aquellas con productos de un sólo equipo- deciden **saltearse** ésta etapa del *Release Planning*, al creer que el esfuerzo requerido para producir el mapeo inicial que será modificado no se condice con el beneficio obtenido.

15.6. Release Planning para Fixed Date

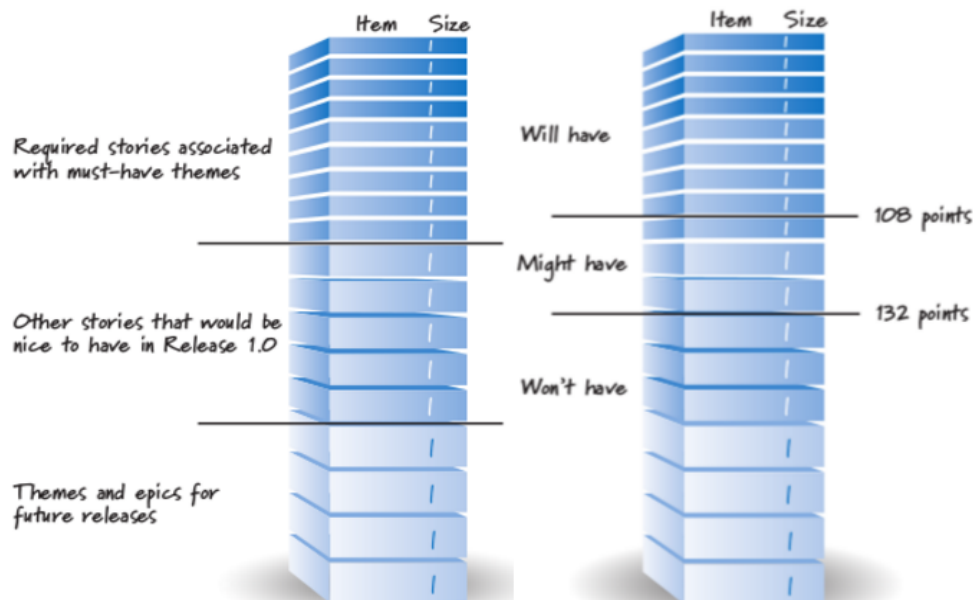
Como ya mencionamos, éste el método preferido para trabajar bajo *Scrum*. Planificar bajo ésta restricción sigue los siguientes pasos:

- Determinar cuántos *Sprints* hay en el *Release*.
- *Groomear* el *PB* creando, estimando y priorizando *PBI's* (*Workshop*, *Story Mapping*, etc).
- Establecer la velocidad del equipo como rango (Una velocidad lenta y otra rápida, Cap. 7).
- Multiplicar la velocidad lenta por la cantidad de *Sprints*. Recorrer el *PB* hacia abajo hasta tener una cantidad de puntos equivalente a la multiplicación. Éste es el compromiso.
- Multiplicar la velocidad rápida por la cantidad de *Sprints*. Recorrer el *PB* hacia abajo hasta tener una cantidad de puntos equivalente a la multiplicación. Ésto es una posibilidad.

A la hora de realizar el *Grooming*, el *devteam* estima las *stories* y el *PO* las prioriza (con input de los *stakeholders* y el *devteam*). Como parte de éste proceso se determina el set de *MRF's* del *Release*. Es deseable que éste set no requiera el 100 % del tiempo del *Release*, pues ésto no nos permite sacar ninguna *Feature* en caso de que el tiempo sea insuficiente ni nos permite agregar una *Feature* que puede surgir luego de lo obtenido en un *Sprint*. Un 70 % es una figura más razonable.

Una vez terminada la estimación y priorización, podremos separar al *PB* en 3 capas: la capa superior tendrá las *stories* necesarias para el *Release*, la del medio son funcionalidades que sumarían valor al producto pero no obligatorias, y la inferior son *themes/epics* para futuros *Releases*. Podemos superponer ésta división del *Backlog* con otra división creada por valores de velocidad obtenidos para el equipo. Ésta superposición nos permite ver, de las funcionalidades que queremos entregar, cuales podremos llegar a hacer dada nuestra velocidad. Suponiendo que la velocidad tenga un **peor caso** de 108 puntos y **mejor caso** de 132, los *Backlogs* mencionados se verían de la siguiente forma:

¹⁴Cohn 2006, ó (sugerencia de Nico A) <https://www.mountaingoatsoftware.com/articles/rolling-lookahead-planning>



Es esperable que las estimaciones sean más precisas conforme se va avanzando en el *Release*. A medida que se completan *Sprints*, podría ocurrir que salga a luz un error en la estimación, implicando que no se puede completar el *MRP* para el *Release*. En ésta situación, se puede revisar el set de *MRP* para analizar si se puede posponer alguna para el siguiente *Release* (es decir, no era realmente obligatoria) o sumar gente al equipo de desarrollo, considerando los costos ocultos que ésto tiene. También se puede adquirir deuda técnica de forma estratégica, adquiriendo tiempo que debería ser pagado en el siguiente *Release*. Todas éstas alternativas van a depender de que tan errada haya sido la estimación (si sólo completamos el 30 % del *MRP*, solucionar todo adquiriendo deuda sería una idea potencialmente terrible o hasta imposible).

Como punto final, es importante considerar los valores de velocidad obtenidos al final de cada *Sprint* del *Release*. Especialmente si se trata de un nuevo equipo sin valores históricos, ya que éstos podrían requerir un cambio en la estimación del *Release*, si la velocidad real es muy diferente de la estimada. También, es importante recordar que por la naturaleza del proyecto pueden aparecer *features emergentes* producto de la finalización de *Sprints*, por lo que el *MRP* no debería impedir éste comportamiento, o al menos debería ser tenido en cuenta.

15.7. Release Planning para Fixed Scope

Antes de comenzar, es importante pensar que suele ser posible transformar un *Release* con *fixed scope* a múltiples *Releases* pequeños de *fixed date*. Ésto sucede porque, frecuentemente, el *MRP* suele estar inflado con soportes opcionales (por ejemplo, una *feature* obligatoria de soporte para múltiples browsers con estándares HTML y CSS cambiantes en la primer versión de una aplicación no suele ser realmente obligatoria comparada con otras áreas mas centrales del negocio del producto). Sin embargo, supongamos que estamos en una situación donde lo anterior no sucede y el set de *MRP* es un factor más limitante que el tiempo. En ese caso, se pueden seguir los siguientes pasos:

- *Grooming* para los *PBI* del *fixed scope*, estimados y priorizados. Sumar para determinar el *Release*.
- Medir (ó estimar) la velocidad del equipo como rango.
- Dividir el tamaño total de los *PBI* por la velocidad más rápida posible del rango (redondeando), obteniendo el menor número de *Sprints* necesario para entregar las *features*.
- Dividir el tamaño total de los *PBI* por la velocidad más lenta posible del rango (redondeando), obteniendo el máximo número de *Sprints* necesario para entregar las *features*.

Éste tipo de planificación difiere de la anterior. Por ejemplo, para *fixed date* se intenta no tener un 100 % de espacio ocupado por el *MRP*, para combatir a la incertidumbre de *features emergentes* sin necesitar mover la fecha. Sin embargo, para *fixed scope* se busca tener un 100 % ocupado por el *MRP*, pues lo limitante es la

funcionalidad y no el tiempo. En caso de *features emergentes* obligatorias, simplemente se las puede agregar al set y extender la fecha del *Release*.

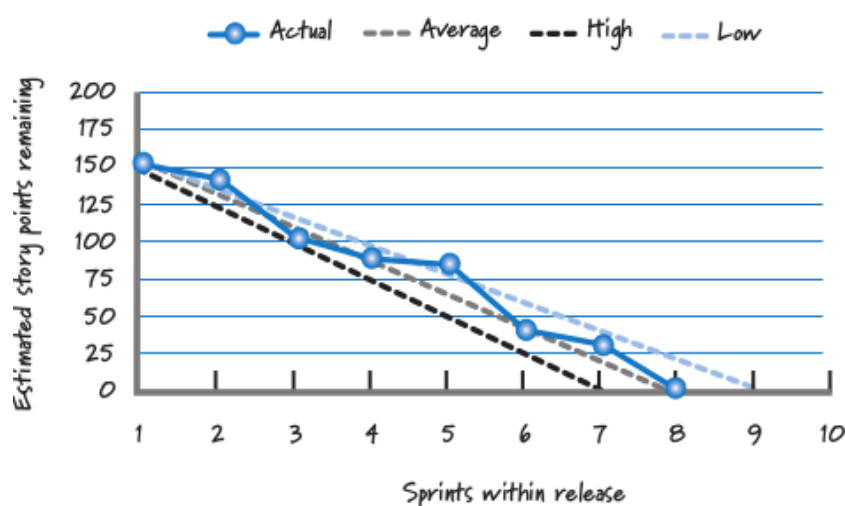
15.8. Calculando Costos del *Release*

- Determinar el equipo: asumir que su composición no cambia durante un *Sprint*, y los potenciales reemplazos a lo largo del *Release* poseen similar salario (con *Scrum* se debería apuntar a no tener equipos tan cambiantes de todas formas).
- Determinar la longitud de los *Sprints*: apuntar a que todos tengan igual longitud.
- Para *fixed date*, multiplicar el número de *Sprints* por el costo por *Sprint*. Para *fixed scope*, multiplicar el menor número de *Sprints* obtenido y el máximo para obtener un rango de costo (el rango va a estar atado a la variabilidad de la velocidad, posiblemente vía intervalos de confianza). A ésto sumarle costos no laborales.
- El costo de cada *Sprint* se puede pensar cómo el costo de mano de obra del equipo por hora, multiplicado por la cantidad de horas por *Sprint*.

15.9. Comunicando el Progreso

15.9.1. *Fixed Scope-Release Burndown Chart*

Éste gráfico sirve para comunicar como se va avanzando hacia la finalizacion de las funcionalidades. En el eje vertical se ubican los *story points* a realizar en el *Release*, y *Sprint* a *Sprint* se va actualizando con lo consumido hasta el momento (la línea “Actual”). Se termina cuando se consumen todos los *story points*. Se puede agregar más información para comparar, como el momento ideal de finalización, el peor, etc (considerando el rango de velocidad). Un ejemplo puede verse en el siguiente gráfico:



15.9.2. *Fixed Scope-Release Burnup Chart*

Básicamente un concepto inverso al anterior. Tenemos como objetivo final los *story points* totales del *Release*, y *Sprint* a *Sprint* vamos actualizando cuanto completamos. Se termina cuando se llega a la “Target Line”, representada por la totalidad de puntos del *Release*. Algunas personas prefieren éste formato porque muestra de una forma sencilla un cambio de *scope* para el *Release* (para aquellos casos donde no se respetan las restricciones). Para mostrar éste cambio, solo necesitamos mover la “Target Line” hacia arriba.

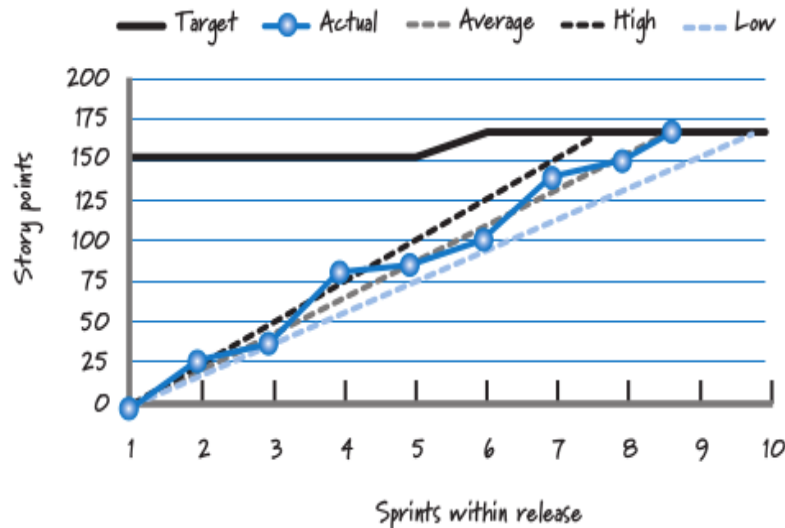
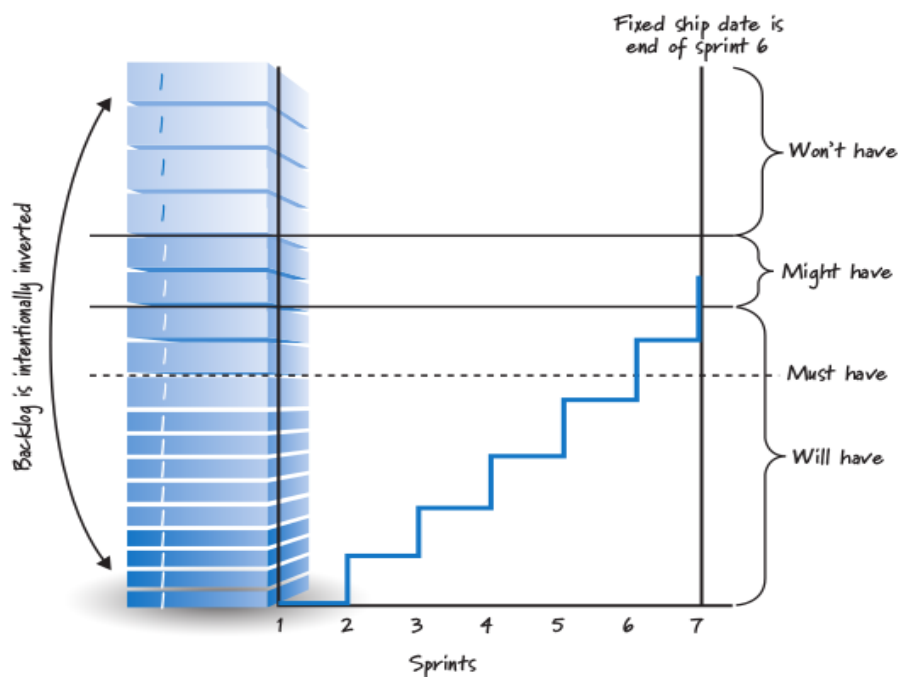


Figura 14: *Burnup Chart* de *scope* (“Target Line”) variable.

15.9.3. Comunicando Progreso en *Releases Fixed-Date*

Con ésta restricción sabemos el número de *Sprints* en el *Release*, pero esperamos un *scope* potencialmente más variable. El foco entonces cambia, y lo que queremos comunicar es el rango de *scope* que se podrá entregar en la fecha fija.

Podemos pensar entonces un nuevo gráfico. Invertiendo el *PB*, tendremos la capa más importante del mismo debajo y la menos importante arriba. Con éste orden, podemos utilizar el *Backlog* al final de cada *Sprint* para marcar cuantas *stories* fueron completadas, utilizando la división de capas del *Backlog* (pues se lo integra al gráfico). De ésta forma, tenemos un mapa de que tan lejos estamos del objetivo del *Release* y como estamos avanzando hacia el mismo, considerando la cantidad finita de *Sprints*. Si bien en el gráfico de prueba no hay proyecciones con velocidad máxima o mínima, éstas son fácilmente agregables.



16. *Sprint Planning*

16.1. Resumen

Un *PB* puede representar mucho mas trabajo que el de un *Sprint*. Por ésto, el equipo de *Scrum* determina un objetivo para el siguiente *Sprint*, el cual será usado por el *devteam* para determinar que ítems del *Backlog* se alinean con ese objetivo y pueden entregar en tiempo razonable. Para adquirir confianza en que pueden efectivamente entregarlo, el *devteam* genera un plan para completar los ítems.

El objetivo del *Sprint* resume el objetivo de negocio del mismo, por lo cual suele ser propuesto por el *PO* y potencialmente refinado por el equipo.

16.1.1. Timing y Participantes

Se realiza al comienzo de cada *Sprint*. Para uno de 2 semanas, debería tomar entre 4 y 8 horas. Todo el equipo participa de la *Planning*. El *PO* presenta el objetivo y ayuda a priorizar el *Backlog*, respondiendo dudas. El *devteam* analiza *PBI's* asociados al objetivo y determina lo que puede entregar en el *Sprint*. El *ScrumMaster* facilita la planificación, haciendo preguntas de exploración de tareas y guiando el compromiso del *devteam*.

16.1.2. Proceso

Ésta planificación requiere ciertos *inputs*, como por ejemplo:

- ***PB* Groomeado:** antes del *Sprint Planning* los *PBI's* del tope deben cumplir con la *Definition of Ready*. Usualmente éstos implica criterios de aceptación, tamaño y priorización.
- **Velocidad del equipo histórica:** un indicador de cuánto trabajo puede completar el equipo.
- **Restricciones:** restricciones técnicas o materiales que podrían afectar lo que el *devteam* puede entregar. Ejemplos: depender de equipos tercerizados para partes de una aplicación, trabajar sobre nueva tecnología, etc.
- **Capacidades del Equipo:** qué tan disponible estarán los miembros del equipo para el siguiente *Sprint* y qué habilidades tienen (para hacerlas rendir el máximo fruto posible).
- **Objetivo del *Sprint*:** el objetivo de negocio que el *PO* quiere concretar en éste *Sprint*.

Éste proceso podría implicar una negociación entre el *PO* y el equipo de desarrollo, para establecer un punto medio entre lo que el *PO* quiere y lo que el equipo puede razonablemente entregar. A su vez, debería haber un espacio para evaluar alternativas que surjan en el momento, para decidir si dichas tareas emergentes no se alinean mejor que las actuales del *Backlog*.

Los *outputs* producidos son el objetivo y *Backlog* del *Sprint*, el cual contiene los *PBI* a realizar y un plan para ellos, que en general consiste en dividir los *PBI's* en tareas estimadas en horas. Para mantener la precisión, no debería haber un tarea singular en un *PBI* que tome más de 8 horas. En esa situación, suele ser más sencillo intentar descomponerla y estimar sus partes por separado para obtener una estimación más precisa.

16.2. Enfoques para la *Sprint Planning*

16.2.1. Separación en Dos Partes

- El **qué:** Primero, se determina la capacidad -cantidad de *story points*- que el equipo puede completar en el *Sprint*. Luego, se seleccionan *PBI's* para cumplir con esa capacidad, alineándolos con el objetivo propuesto por el *PO*.
- El **cómo:** Con los *PBI's* seleccionados, se arma un plan para el *Sprint*. Éste plan suele consistir en el desarmado de cada *PBI* en tareas estimadas en horas. La suma de horas se contrasta con la capacidad

del *Sprint*, para evaluar si el compromiso es realista. Si lo es, se termina, y si no lo es se reevalúan los *PBI* a realizar hasta que lo sea. Podría no serlo por cierta restricción del *Sprint* que impide que ciertos *PBI* se desarrollen a la vez, por un tema de horas insuficientes, o otros.

16.2.2. Sin Separación

Suele ser la más común. Con éste enfoque, el *devteam* comienza determinando cuál es su capacidad. Con éste dato, se evalúa si refinar o no el objetivo del *Sprint* y se comienza a extraer *PBI's* del *PB*. Por cada *PBI*, se lo baja a tareas y se lo estima, comparándolo con la capacidad restante del equipo en el *Sprint* actual. El ciclo se repite por cada *PBI*, y termina cuando se acaba la capacidad.

16.3. Determinando la Capacidad

16.3.1. Qué es?

La capacidad representa la cantidad de trabajo que podemos realizar dentro del *Sprint*. La capacidad total del *Sprint* debería estar dividida en varias categorías, pues no podemos utilizar todo el tiempo para el desarrollo. Algunas de las áreas importantes son:

1. Tiempo para *Sprint Planning*, *Review* y *Retrospective*: para *Sprint* de dos semanas, éste suele ser un día de trabajo.
2. Tiempo para asistir al *PO* con tareas de *Grooming* (crear/refinar/estimar/priorizar *PBI's*): aprox un 10 % de la capacidad.
3. Tiempo para hacer actividades fuera del *Sprint* (trabajar en otro producto, etc).
4. Tiempo personal: para cumplir con las responsabilidades del trabajo (reuniones, asambleas, interrupciones, etc).

Lo que resulta de esto es la cantidad de tiempo que podremos trabajar en *PBI's* para el *Sprint*. Sin embargo, es fuertemente recomendable dejar de eso un *buffer* para ciertos imprevistos, por ejemplo estimaciones erradas. La metodología para determinar un *buffer* depende de cada proyecto en particular¹⁵, pero como medida ciega entre un 7 y 8 % suele ser razonable.

16.3.2. Capacidad en *story points*

La velocidad de un equipo se mide en relación a cuántos *PBI's* completo y que tan “grandes” eran. Si para esa estimación se usan *story points*, determinar la capacidad resulta lo mismo que predecir la velocidad que se tendrá en el *Sprint* actual.

Para esto, es más conveniente utilizar como data previa la velocidad histórica del producto o la velocidad del *Sprint* anterior (suponiendo longitud de *Sprint* fija). A éste número se lo debe ajustar por las particularidades del *Sprint* actual para obtener una estimación razonable en poco tiempo. Por ejemplo, se debería tomar en cuenta si en éste *Sprint* habrá desarrolladores de vacaciones o ciertos miembros tendrán trabajo intenso en otra área de la empresa, y cómo eso podría influir en la ejecución del *Sprint*.

16.3.3. Capacidad en *effort hours*

Para expresar la medida en *man hours* ó *effort hours*, primero se deben eliminar ciertas variables. Como primer paso, se debería identificar cuántos días disponible se tiene, por miembro, para trabajar en el *Sprint* (incluye tareas de planificación). A esto, se le restan los días destinados a otras actividades de *Scrum*, y se determina una aproximación para las horas diarias que cada miembro podrá dedicar al proyecto (tomando en cuenta que no todo el tiempo laboral suele ser para el proyecto). El rango final determina las horas

¹⁵Hay ejemplos en Cohn 2006: *Agile Estimating and Planning*.

totales disponibles. Una buena estrategia es tomar trabajo que sea superior al valor mínimo del rango, pero mucho menor al valor máximo del mismo, para poder tener un *buffer*. Todo el proceso se puede resumir en el siguiente cuadro:

Person	Days Available (Less Personal Time)	Days for Other Scrum Activities	Hours per Day	Available Effort-Hours
Jorge	10	2	4-7	32-56
Betty	8	2	5-6	30-36
Rajesh	8	2	4-6	24-36
Simon	9	2	2-3	14-21
Heidi	10	2	5-6	40-48
Total				140-197

16.4. Seleccionando *PBI's*

La selección de *PBI's* para el *Sprint* puede hacerse de dos formas. Si hay objetivo para el *Sprint*, se seleccionan los ítems alineados con el objetivo. Si no lo hay, se seleccionan los *PBI's* del tope del *PB*. En caso de no tener capacidad suficiente para alguno en particular, siempre se puede agarrar el siguiente.

Una regla fundamental en la selección es no empezar aquello que no pueda ser terminado. Por ello, si un *PBI* resulta demasiado grande se debería descomponer el mismo en *PBI's* mas pequeños. Ésto suele ser posible reduciendo la complejidad de la funcionalidad a entregar en el *PBI*. Por ejemplo, si hay una tarea de sincronización, ésta podría ser reducida en la “sincronización ideal” y la “sincronización real” (que incluye cosas como condiciones de carrera y ajustes finales). Si el *PBI* es irreducible, siempre se puede tomar el siguiente prioritario. En definitiva, el objetivo final es reducir el *WIP* y los desperdicios que eso genera.

16.5. Adquiriendo Confianza

Para tener confianza en el compromiso que se quiere asumir en el *Sprint* (por ejemplo 25 *story points*), suele ser excelente el uso de la velocidad como métrica. Si estimamos una velocidad de 45 para éste *Sprint*, sabemos que nuestro compromiso debería ser cercano a ese número. Sin embargo, únicamente la velocidad tiene el riesgo de que lo estimado en *story points* podría no corresponderse con la realidad. Ésta realidad se conoce al desglosar los *PBI* en aquellas tareas necesarias para su completitud. La completitud de cada *PBI* está medido por la *Definition of Done* del equipo. Ésta actividad resulta en el *Sprint Backlog*, que contiene cada *PBI* y su descomposición en tareas, por lo que se terminan teniendo los *story points* y las *effort hours* de cada una.

En resumen, para adquirir un buen grado de confianza, no sólo se debe usar la velocidad, sino que es necesario descomponer los potenciales *PBI's* en tareas y revisar las *effort hours* finales para detectar problemas más intrínsecos (como puede ser dependencias, falta de habilidad en el equipo para cierta tarea, etc).

Además, se debería considerar el tiempo que cada uno tiene en el equipo. Si nuestras *effort hours* disponibles son 140-197, y la suma de las planificadas da 150, parecería un compromiso con sentido. Más aún si la suma de *story points* de los *PBI* se ajusta a la velocidad predicha. Sin embargo, dentro de los *PBI* a realizar podría haber muchas tareas únicamente realizables por un individuo. Esa persona tiene suficiente tiempo disponible para trabajar en el proyecto dentro del *Sprint*? Si no lo tiene, se debería contemplar (aumentando su tiempo reduciendo otros proyectos o cambiando los *PBI*), y éste tipo de análisis no serían detectados si solo miramos las horas o los *story points*.

17. Ejecución del *Sprint*

Ésta es la fase del *Sprint* que más tiempo toma. En ésta etapa, el *devteam* se auto-organiza y determina la mejor forma de cumplir el compromiso y objetivo del *Sprint*. El *SM* participa como *coach* de los temas ágiles y removedor de impedimentos, pero no asigna trabajo al *devteam* ni le dice como hacerlo (no es un *Project Manager*). Un *PO* debe estar disponible para responder dudas del dominio del producto, proveer aclaraciones o ajustes en caso de que las circunstancias cambien y verificar que los criterios de aceptación de los *PBI* fueron cumplidos.

La ejecución del *Sprint* requiere del *Sprint Backlog* y el objetivo del mismo. Como *output*, ésta ejecución devuelve un *SPI*: *Shippable Product Increment*, que representa un set de *PBI's* completados con un alto grado de confianza, listos para ser integrados.

17.1. Planificación de la Ejecución del *Sprint*

Como el *Sprint Backlog* posee los *PBI* estimados con tareas en *effort hours*, uno podría querer planificar quién va a hacer éstas tareas y cuando, para todas las tareas, en el *Sprint* (tal vez con un Gantt). Ésto es una mala idea por múltiples razones. La más importante es que, desde un punto de vista económico, no tiene mucho sentido. Al ser una metodología ágil, el Gantt sería impreciso apenas se comienza a trabajar. Una gran cantidad de aprendizaje resulta de realizar una tarea y testearla, aprendizaje que no puede ser planificado de antemano en ningún plan. Entonces, se gastó tiempo en un plan muy preciso, sólo para tener que gastar tiempo actualizando a la realidad, que hubiera surgido de todas formas.

La planificación a nivel de tareas debería usarse sólo para los ejes principales de cada una o dependencias (por ejemplo anticipar que se va a necesitar 2 días para hacer un *stress test* para una funcionalidad). Ésta planificación debería seguir durante el *Sprint* para que el equipo pueda adaptarse a su evolución, en vez de querer resolverla toda de antemano.

17.2. Manejo de Flujo de Trabajo

17.2.1. Trabajo en Paralelo

Una importante decisión a tomar en cuenta es cuántos *PBI's* se quieren estar atacando en paralelo por múltiples personas. Ésto aumenta el porcentaje de multitasking del equipo, aumentando el tiempo que toma terminar elementos individuales, y probablemente reduciendo su calidad. Para entender ésto, existe el ejemplo didáctico de las letras y los números. Hay dos tablas, y la primera se debe escribir por filas y la segunda por columnas:

Letters	Numbers	Roman numerals
a	1	i
b	2	ii
c	3	iii
d	4	iv
e	5	v
f	6	vi
g	7	vii
h	8	viii
i	9	ix
j	10	x

Letters	Numbers	Roman numerals
a	1	i
b	2	ii
c	3	iii
d	4	iv
e	5	v
f	6	vi
g	7	vii
h	8	viii
i	9	ix
j	10	x

Los resultados típicos muestran que la mayoría de la muestra termina la tabla por columnas -equivalente a tareas individuales- en aproximadamente la mitad del tiempo que la tabla por filas -equivalente a multitasking-. Como regla general, ambos extremos tienen repercusiones negativas en el valor entregado por el *Sprint*. El balance que hay que manejar es hacer la cantidad de *PBI's* que maximice el uso de las habilidades en forma de T de cada miembro y su capacidad disponible. Ésto debería reducir el tiempo que toma cada ítem, maximizando lo que se va a entregar.

Un enfoque para ésto se conoce como *swarming*¹⁶. Ésto focaliza que los miembros con capacidad disponible se junten a trabajar en un ítem ya comenzado para terminarlo, antes de pasar al siguiente. Ésto es diferente a enfoques mas usuales en otras metodologías donde ciertos miembros se dedican a codear, otros a testear, y así se van dejando funcionalidades incompletas “esperando que otro la termine”.¹⁷

Swarming no debería ser usado para mantener a la gente 100 % ocupada (ya vimos que eso es malo), sino para mantener a los miembros concentrados a nivel funcionalidad, no tarea. Éste enfoque favorece reducir un poco la concurrencia, pero no al punto de que todos estén trabajando en un sólo ítem, sino balanceando las habilidades de cada uno con las tareas a realizar.

17.2.2. Qué Trabajo Comenzar y Cómo Organizar Tareas

A menos que haya problemas de dependencias de otras tareas o necesidad de algún miembro del *devteam* específico, decidir que *PBI* comenzar es tan sencillo como agarrar el del tope del *Backlog*. Una vez comenzado, se debe decidir como organizarse a nivel *tasks* dentro del *PBI*. Un enfoque común es hacer *Waterfall* dentro, dónde lo analizamos, codeamos y testeamos al final. Sin embargo, la esencia de trabajar bajo metodologías ágiles es que no existe una única correcta manera de organizar tareas. Por lo tanto, lo mejor es darle al *devteam* la libertad de hacerlo justo a tiempo, en base a lo que se quiere entregar. Se debe confiar que el *devteam* siempre asignará a la persona más indicada para realizar la tarea (en la medida que ésta se encuentre disponible), sea por *expertise* técnico superior o por que se quiere capacitar a alguien.

Por ejemplo, si se tiene un *PBI* a medio terminar se puede intentar organizar las tareas para habilitar el *swarming* con 2 miembros del equipo, para resolverlo de forma colectiva. Otro miembro podría estar aplicando *TDD* en otro *PBI* que contiene una funcionalidad crítica, necesaria de testeo robusto. Al mismo tiempo, otro miembro trabajando en otro *PBI* decide tomar deuda estratégica al tratarse de un ajuste en un área poco utilizada del producto, con el objetivo de terminar el *Sprint* a tiempo, por lo que decide no hacer tests.

17.3. Daily Scrum

Ésta actividad dura alrededor de 15 minutos, y ocurre cada 24 horas (en un slot horario predeterminado). Se utiliza como momento de inspección, sincronización y planificación adaptiva (con cosas como *Grooming*). El *Daily Scrum* es esencial por dos motivos. Primero, permite que todo el equipo, comparta la “big picture” de lo que se está trabajando, para saber como se está llegando al objetivo, que otras funcionalidades comenzar y como organizar ese trabajo. Además, previene esperar mucho para resolver bloqueantes (como máximo se espera) sin la necesidad de interrumpir a otros *TL's* (con su costo de *context switch*).

17.4. Prácticas Técnicas para el Rendimiento

- Si bien se espera que el *devteam* posea conocimiento técnico, se debe ir más allá si se quiere ser verdaderamente ágil. En éste entorno, se trabaja en iteraciones cortas donde se espera poder entregar valor al cliente en forma de un *PSP* ó *PSI*, por lo que si no se poseen ciertas habilidades se podría perder el control sobre ciertos aspectos -como la Deuda Técnica- intentando cumplir con el objetivo del *Sprint*. En particular, ciertas habilidades a tener en cuenta son:

- *TDD* y refactoring.
- *Simple Design* y *Pair Programming*.
- *Continuous Integration* y *Collective Code Ownership*.
- Estándares de código y *System Metaphor*.¹⁸

¹⁶<https://www.agilealliance.org/resources/experience-reports/swarm-beyond-pair-beyond-scrum/>

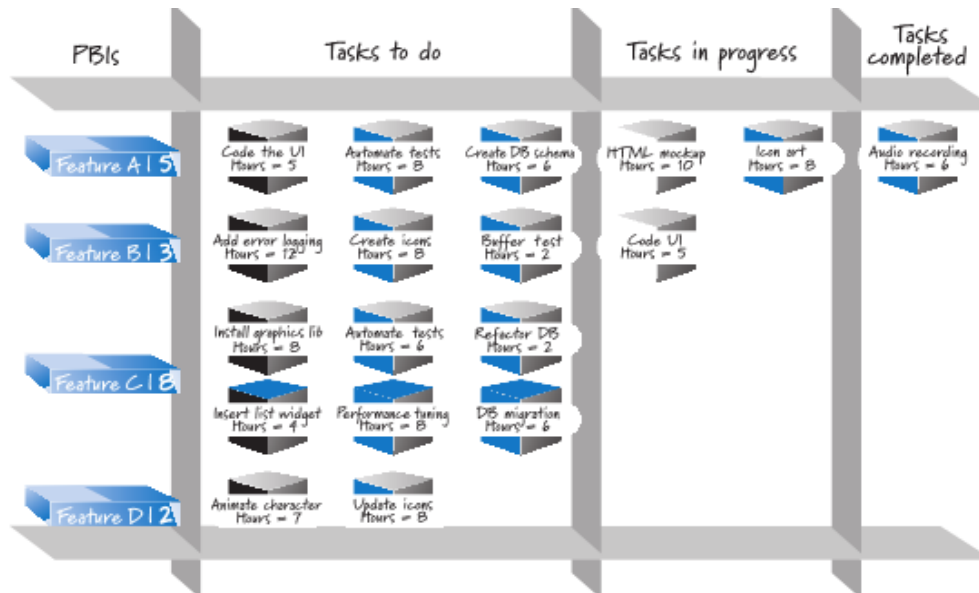
¹⁷Otro enfoque negativo es realizar *Waterfall* dentro del *Sprint*, separando en fases de analisis, coding, testing. Qué sucede si se acaba el tiempo antes de la fase final?

¹⁸Concepto de *Extreme Programming*: “At its best, the metaphor is a simple evocative description of how the program works, such as *this program works like a hive of bees, going out for pollen and bringing it back to the hive* as a description for an agent-based information retrieval system.”

17.5. Comunicación

17.5.1. Task Board

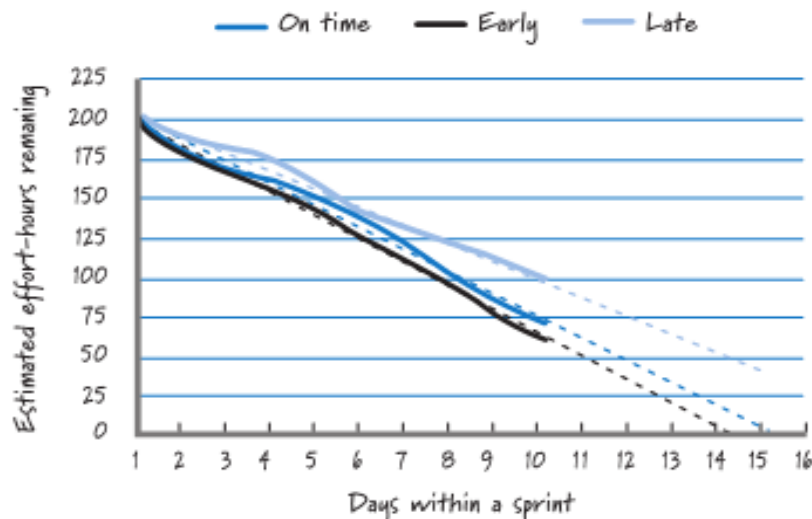
Una de las formas mas eficientes de comunicar progreso en el equipo es con un *Task Board*, el cual muestra la evolución del *PB* a lo largo del *Sprint*. A la izquierda se tienen los *PBI* priorizados del *Sprint* y se manejan las tareas adyacentes. Es sencillo de entender presentando un posible ejemplo de *Task Board*:



17.5.2. Sprint Burndown y Burnup Chart

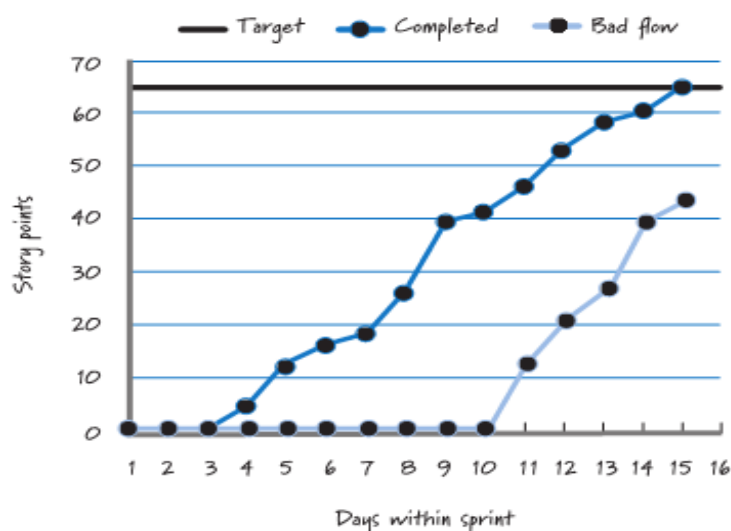
En el *Sprint*, tenemos una serie de *PBI's* bajados a *tasks* con estimaciones en *effort hours*. A lo largo de la iteración, pueden aparecer nuevas tareas a medida que se avanza sobre un *PBI* (si no se contemplaron ciertas ramificaciones de la funcionalidad, por ejemplo). También, puede ser necesario reestimar ciertas tareas dentro del *Grooming* por errores de *scope* cometidos en la estimación inicial (lo cual ocurre por naturaleza dentro de un entorno ágil donde se está siempre aprendiendo).

En definitiva, día a día podríamos tener una cantidad evolutiva de tareas y *effort hours* que abarcan el *Sprint*. Para revisar que el número actualizado tenga sentido con la expectativa de cierre del *Sprint* y poder tomar decisiones en base a ello, es conveniente utilizar alguna herramienta de *tracking*. En el capítulo anterior vimos *Release Burndown y Burnup Charts*, donde el eje vertical era representado por los *story points* del *Sprint* y el horizontal la cantidad de *Sprints*. Se pueden realizar gráficos similares para el *Sprint* actual, utilizando la estimación en *effort hours* y los días del *Sprint* como se ve abajo:



En éste gráfico, se agregaron estimativos “Early” y “Late”, que se pueden conseguir con datos históricos del equipo y el proyecto. Esperaríamos que la línea de “On Time” se ubique en el espacio intermedio de los estimativos. Una línea de “On Time” por debajo de “Early” implicaría, por ejemplo, que se podría considerar tomar trabajo adicional o saldar deuda pendiente por exceso de tiempo. Para los *Burnup Charts*, también se pueden utilizar *story points* del *Sprint* en vez de *effort hours*.

Mucha gente prefiere los *story points* porque, al final del *Sprint*, lo que realmente importa es el trabajo de valor para el cliente realizado, el cual se mide en *story points*, no en tareas con *effort hours*. Además, éste gráfico sirve para medir el “flujo” de completitud de los *PBI*. Por ejemplo, si vemos que no se completan *story points* por varios días y al final todo se completa de repente, podría ser un indicio de que existe algún bloqueante en nuestra *Definition of Done* (Code Review monolitico por ejemplo), o un indicio de que estamos paralelizando demasiado trabajo. Podrían también ser *PBI*’s muy grandes que, para la próxima iteración, podrían ser reducidos para no trabar el flujo del mismo. Como se puede ver, es una buena herramienta para detectar múltiples tipos de problemas (que pueden ser hasta externos al *Sprint*). Veamos un ejemplo de uno con un “target” que representa el *Sprint* terminado, y una línea extra que representa un flujo de trabajo indeseado:



18. Sprint Review

Ésta actividad ocurre luego de la ejecución del *Sprint* y -usualmente- antes de la Retrospectiva. La *Sprint Review* es el momento donde se valida lo creado en la iteración y los problemas que surgieron en la misma,

con el objetivo de proveer una mirada transparente al estado del producto. Al ser una oportunidad crucial de *feedback*, los participantes de ésta actividad deberían ser todos.

El equipo de *Scrum* (*PO*, *devteam* y *SM*) debe participar para poder describir el trabajo realizado y oír el *feedback*. Los que proveen el *feedback* que asegura que se siga progresando de forma económicamente sensible son los *stakeholders* internos (operations managers, ejecutivos que pagan por el producto, usuarios internos si se trata de producto interno) y externos (el input de algún usuario consumidor puede ser invaluable, al menos de forma periódica). También puede sumarse gente de otro departamento de la empresa para sincronizar su trabajo con el equipo de *Scrum* (por ejemplo el de marketing o ventas, para asegurarse que se está llenando por un camino vendible).

18.1. Trabajo previo a la *Sprint Review*

18.1.1. Determinar a Quién Invitar y Programar la Actividad

El equipo de *Scrum* tiene que determinar quienes deberían participar de forma regular primero. Para los demás, es mejor invitarlos de forma esporádica o medida que haga falta, para mantener la visión de cada *Review* clara. En general, suelen participar siempre el equipo de *Scrum* y los *stakeholders* internos, mientras que los demás son más circunstanciales.

Ésta actividad suele ser la más difícil de programar de *Scrum*, debido al número de participantes. Conviene empezar hablando con los *stakeholders* clave para coordinar una fecha inicial, y luego programar el resto de las fechas del *Sprint* alrededor de la fecha propuesta por ellos para la *Review*. Utilizar una cadencia regular en el *Sprint* (cuando son de duración consistente) facilita mucho ésta tarea. La duración de la *Review* depende de la duración del *Sprint*, el tamaño y cantidad de equipos, entre otros factores. En general, suele estar entre 2 y 3 horas para *Sprints* de 1 equipo de duración entre 2 y 3 semanas.

18.1.2. Confirmar que el Trabajo del *Sprint* Esta Terminado

En la *Review*, solo se debería presentar el trabajo que cumple con la *Definition of Done* del equipo. Es decir, alguien tuvo que revisar que se cumpla la definición antes de la *Review*. En particular, es la responsabilidad del *PO* determinar cuando algo está terminado o no. Al revisarlo durante el *Sprint*, se pueden arreglar errores de forma más económicamente sensible. En vez de realizar una funcionalidad y llegar a la *Review*, obtener el *feedback* y eliminar trabajo realizado, se puede hacer esto durante el *Sprint* para reducir la cantidad de trabajo eliminado y terminar el mismo con un producto *potentially shippable*. Esto es especialmente importante para los inicios de funcionalidades más pesadas, donde mirarlo al final de la iteración para descartar el trabajo podría ser muy costoso.

. Existen algunas contras de permitir que el *PO* haga esto dentro del *Sprint*, apenas el *devteam* considera la tarea terminada, en vez de realizarlo todo en la *Review*. Por ejemplo, algunos *PO's* podrían proponer mejoras y ajustes que van más allá de una simple clarificación, alterando fuertemente el objetivo y *scope* del *Sprint*. Se debería tomar en cuenta que un *PO* que cuestiona o rechaza trabajo no representa la intención de “equipo” que propone *Scrum*. En particular, en éstos casos se debería intentar obtener un balance entre que el *PO* no cambie el objetivo de forma regular y obtener *feedback* rápido.

18.1.3. Preparar la Demo y Determinar Participantes

Existe una regla bastante útil: “No gastar mas de 30 /45 min por cada semana de *Sprint* preparando la demo”. La *Review* debería una reunión con poco costo ceremonioso y alto valor para el cliente. El tiempo invertido para prepararla debería gastarse en crear escenarios para mostrar las funcionalidades del producto más fácilmente, no para elementos superfluos como bellas presentaciones de *PowerPoint*. Además, no hacer que el producto sea el foco de ésta reunión genera desconfianza sobre si está “realmente terminado”. Claramente, hay excepciones a la poca preparación, por ejemplo si se trabaja con importantes figuras gubernamentales y ellos asisten a la *Review*.

Antes de la *Review*, el equipo debe decidir quien será el “facilitador” y quién mostrará el trabajo completado. El facilitador suele ser el *ScrumMaster*, aunque a veces el *PO* puede dar una introducción a los

stakeholders. Es recomendable que todo el *devteam* participe demostrando lo terminado, para promover la horizontalidad, aunque sin dejar de lado intentar maximizar la eficiencia y beneficio de la *Review*.

18.2. Enfoque

La *Review* tiene como *inputs* el *Sprint Backlog*, el objetivo y el *PSP/PSPI* generado por el *Sprint*. En general, se suele proveer una sinopsis de lo que se logró y no se logró en base al objetivo, seguido de una demo del *PSI*, y al final una discusión del estado del producto actual y su futura dirección. Ésto produce como *outputs* un *PB* *groomed* y un *Release Plan* actualizado.

18.2.1. Resumir y Demostrar

En general, la *Review* comienza con un miembro del equipo (en general el *PO* o *SM*) presentando el objetivo del *Sprint* con sus *PBI* asociados, y como ellos fueron completados. Si no lo fueron, el equipo da una explicación razonable al respecto. Es importante notar que la *Review* no es el ambiente para echar culpas a nadie, el foco debe estar en describir lo que se logró y usar esa información para seguir adelante. Luego de ésto, se procede a la demostración del *PSP*, informalmente conocido como “la demo”. Éste paso es muy útil para concretar el objetivo del ritual, centrado en una colaboración entre los participantes para permitir que adaptaciones productivas del producto surjan y sean explotadas.

En la demo, uno o más miembros del equipo muestran todos los *PBI* relevantes al trabajo que se realizó. Podría ocurrir que se estuvo trabajando en cosas poco “demostrables”, como puede ser la arquitectura interna. Se podría pensar que no tiene sentido demostrar ésto, cuando en realidad suele tenerlo en la mayoría de los casos. Para que el equipo haya trabajado sobre elementos de arquitectura, por ejemplo, se tuvo que haber convencido al *PO* que los permita en el *Backlog*, por lo que el *PO* debería entender el valor del trabajo y como reconocer si cumple con sus expectativas de calidad. Además, la mayoría de los equipos incluyen en su *Definition of Done* saber como demostrar el ítem.

Como mínimo, el equipo debería tener una serie de tests que demuestren que el trabajo realizado en el *Sprint* fue hecho de acuerdo con la expectativa del *PO*. Por ende, se podrían usar para demostrar el progreso, al menos como una medida mínima de visibilidad. En general, se puede realizar un mejor trabajo de demostración que simplemente un par de tests, por lo que el equipo no debería descartarlo solo porque es “difícil de demostrar”.

18.2.2. Discutir y Adaptar

La demo se convierte en punto central para tener una conversación a fondo con observaciones, comentarios y discusiones acerca del producto y su dirección, de la cual todos deberían participar. También, es el momento en el que el equipo de *Scrum* puede ganar más contexto sobre los aspectos del negocio y marketing asociados al producto, a través del *feedback* obtenido. En particular, se pueden preguntar y responder ciertas preguntas como:

- Los *stakeholders* están conformes con lo que ven? Quieren cambios?
- Nos está faltando alguna *feature* importante?
- Estamos sobredesarrollando o invirtiendo demasiado en una *feature*?

Responder éstas cuestiones provee *input* sobre como **adaptar** el *PB* y el *Release Plan*. En particular, es muy común realizar *Grooming* dentro de la *Review*. Cuando todos ganan un mayor entendimiento del desarrollo actual del producto, se suelen crear nuevos *PBI* con extensiones de funcionalidades desarrolladas o nuevas funcionalidades disparadas por el desarrollo actual, las cuales pueden afectar el flujo del siguiente *Sprint*.

En consecuencia, el *Release Plan* puede verse modificado. Por ejemplo, se puede detectar una nueva funcionalidad crítica que se debe agregar al *Release*, o se puede eliminar una funcionalidad que quedó anticuada por un avance del competidor de mercado (ésto altera el *scope*).

18.3. Problemas de la *Review*

18.3.1. Aprobaciones

Las aprobaciones de los *PBI's* pueden ser una actividad complicada para realizar en la *Review*. Antes de la demo con los *stakeholders*, el *PO* revisa que el trabajo hecho cumpla con la *Definition of Done*, por lo que el trabajo ya se encuentra aprobado por él antes que la *Review* comience (y él representa la visión de los *stakeholders* sobre el producto).

Supongamos que ocurre el peor caso, un *stakeholder* declara que, en su opinión, el *PBI* no está terminado. En ésta situación, es fundamental que el *PO* sea el punto central de liderazgo del producto en su esfuerzo de desarrollo (como se discutió en el Capítulo de los *PO*). Entonces, se debería respetar la decisión que el *PO* considere. Como solución, se debería agregar un nuevo *PBI* con los ajustes propuestos por el *stakeholder*. Además, se debería charlar en la Retrospectiva porque se tuvo una desconexión con un *stakeholder*, y cómo se puede ajustar para intentar que no ocurra nuevamente.

18.3.2. Asistencia Esporádica

Una de las causas más comunes de la asistencia esporádica es la complicada agenda de los *stakeholders*, por lo que compromisos “mas importantes” toman prioridad. Ésto es un síntoma de una disfunción a nivel organizacional, pues los *stakeholders* tienen tantos compromisos que no pueden ir a todos. En éste caso, es conveniente que las organizaciones dejen de trabajar en productos de muy baja prioridad hasta que sean lo suficientemente importantes para que los *stakeholders* puedan presenciar sus *Sprint Reviews*. Si ese día nunca llega, esos productos de poca prioridad en relación a otros en el *Portfolio* simplemente no son valiosos para trabajar.

Otra causa muy común es cuando una organización comienza a usar a *Scrum*. Se suele estar acostumbrado a períodos de desarrollo bastantes más largos, por lo que puede parecer que el equipo de *Scrum* “no puede” producir algo relevante en un par de semanas. La mejor forma de intentar solucionar ésto -mas allá del cambio de percepción que se tiene que tener a nivel organización sobre metodologías ágiles- es construir un *PSPI* en cada *Sprint* que contenga aquellos *PBI* que más valor al cliente le dan. Cuando se ven los rápidos resultados de elementos que dan valor, suele ser más sencillo comprender que las *Reviews* frecuentes valen la pena para darle *feedback* al equipo de *Scrum* y que ellos desarrollen un producto de mucha mejor calidad.

18.3.3. Grandes Grupos de Desarrollo

Si se tienen muchos equipos de *Scrum*, podría tener sentido considerar hacer una *Joint Sprint Review*, una *Review* que incluya el trabajo completado por múltiples equipos que tengan trabajo relacionado. Ésto tiene el beneficio de que los *stakeholders* sólo tienen que venir a una reunión. Además, si el trabajo debe estar integrado, tiene sentido que la *Review* se concentre en la integración final, no en cada parte de funcionalidad por separado por equipo. Para conseguir ésto, es claro que los equipos deberían tener incorporado el testing de integración a su *Definition of Done* (como deberían en cualquier caso). El problema que genera ésta *Review* es que tomará más tiempo y requerirá más espacio, pero aún así debería tomar menos tiempo y ser más productiva que si se hiciesen todas por separado (debido a la relación que tiene que haber entre el trabajo de los equipos).

19. *Sprint Retrospective*

La Retrospectiva (o “la retro”) es uno de los rituales mas fundamentales y menos apreciados de *Scrum*. Le provee al equipo la oportunidad de frenar un segundo el desarrollo ágil de los *Sprints* para examinar qué es lo que está pasando y analizar la forma de trabajo. Todo lo que afecta cómo el equipo crea el producto debería entrar en la mirada crítica (procesos, prácticas, comunicación, herramientas, etc). De ésta introspección, surgen medidas a mejorar *Sprint* a *Sprint*. Éstas mejoras son iterativas, pues se aplican

constantemente durante el desarrollo y no al final de un largo ciclo, donde se podría perder impacto. Se pueden usar preguntas como disparadoras de la reunión, como por ej:

- Qué medidas funcionaron en éste *Sprint* que queremos seguir haciendo? Qué otros aspectos podemos mejorar? Qué cosas fallaron en éste *Sprint* que deberíamos dejar de hacer?

19.1. Participantes

Como la Retrospectiva refleja sobre todo el proceso, todos los miembros del equipo -*devteam*, *PO* y *SM*- deberían participar de ella. Ésto nos asegura de que tengamos en juego todas las perspectivas posibles para poder encontrar la mayor cantidad de mejoras a realizar. El *SM* está presente al ser una parte del equipo y la autoridad del proceso ágil. Ésto **no** significa que le dice al *devteam* cómo mejorar, sino que les indica qué reglas colectivamente consensuadas del proceso no están siendo correctamente cumplidas, además de proponer sus propias ideas y perspectiva.

Algunos equipos sienten que un *PO* presente podría inhibir al equipo en ser honesto con respecto a sus falencias. En éste caso, el problema radica en la falta de confianza dentro del equipo y no en la Retrospectiva. El *PO* es una parte integral del equipo y el canal por donde se transmiten los requerimientos al *devteam*, por lo que debería estar presente para mejoras de esa índole. Por ejemplo, que pasa si los *PBI* no están bien *groomeados* al comienzo de la *Planning*? En ese caso podría ser difícil para el equipo mejorar ese aspecto sin el *PO* presente. Cuando hay falta de confianza, es responsabilidad del *SM* actuar de coach ágil y guiar hacia la resolución de problemas entre el *devteam* y el *PO*, solución que probablemente implique un cambio de actitudes.

Aún así, es responsabilidad del *SM* remover impedimentos, por lo que si se llega a la conclusión final de que el *PO* posee ciertas actitudes (por ejemplo forzando ritmos insostenibles en el equipo o culpabilizándolo de problemas), se puede decidir removerlo de la Retrospectiva. En esa situación, se debería evaluar fuertemente si se quiere seguir con ese *PO*, pues la necesidad de removerlo va en contra de los principios ágiles al no tratarlo como un miembro del equipo.

Otras partes del proceso deberían participar de la retro sólo si son invitadas por el equipo, pues si el mismo no se siente con la confianza abierta para hablar de problemas reales y encontrar mejoras por la presencia de ciertos individuos (por ej *stakeholders*), la retro pierde sentido.

19.2. Trabajo Previo

19.2.1. Definir el Foco de la Retrospectiva

El foco de la retro en general suele ser abierto, como revisar los aspectos relevantes del proceso de *Scrum* que se usaron en éste *Sprint*. Sin embargo, puede ser útil focalizarlo en algo más concentrado, para atacar problemas de forma más directa (por ejemplo, mejorar *TDD* o mejorar comunicación con cliente). Establecer el foco de la retro de antemano le permite al equipo de *Scrum* prepararse mejor para la misma -pensando ideas o recopilando información útil-, ahorrando tiempo. Establecer focos más concentrados para retrospectivas mas cortas es especialmente útil en equipos maduros de *Scrum*, donde se supone que ya hay un buen manejo de la estructura general del proceso, pero si se podría mejorar en áreas específicas.

19.2.2. Seleccionar Ejercicios y Recolectar Información

Los “ejercicios” de la retro tienen como objetivo que los participantes discutan, piensen y decidan juntos. Una retro típica puede tener a cada miembro compartiendo sus intuiciones de mejores, y luego una votación para la decisión final de cuales aplicar. Otro ejercicio muy típico para ciclos sin retrospectiva altos se conoce como la *Retrospective Timeline*¹⁹, en el cual se hace una línea de tiempo con todos los eventos notables positivos, problemáticos y significativos desde la última retro, y se utiliza esa información para visualizar tendencias en el transcurso del *Sprint*, con el objetivo de mejorar las negativas.

¹⁹<https://www.thekua.com/rant/2006/03/a-retrospective-timeline/>

Los ejercicios a decidir en la etapa previa a la retro son aquellos que necesiten recolección de datos a lo largo del *Sprint*, para no tener que hacer ésto en la reunión y perder tiempo. Para los demás, suele ser mejor realizarlos justo a tiempo en base a lo que los participantes crean que funciona mejor. La recolección se debería realizar sobre datos objetivos (no opiniones), como por ejemplo contar el *WIP*, los eventos, etc.

19.2.3. Estructurar la Retrospectiva

La duración de la retro depende de varios factores como la cantidad de personas, si están remotos, su experiencia, etc. En general, es muy difícil tener una retro útil en menos de 60 minutos, por lo que dedicarle aproximadamente 1.5 horas para un *Sprint* de 2 a 3 semanas suele ser suficiente. El lugar de la Retrospectiva no es tan importante, siempre y cuando se tenga fácil acceso a la información necesaria y se pueda hablar con libertad de las problemáticas a enfrentar. Como último, debería decidirse el facilitador. Si bien éste suele ser el *ScrumMaster*, realmente cualquier miembro puede actuar de éste rol (y a veces es positivo rotar un poco). En retrospectivas donde se toquen temas mas sensibles, puede ser una buena alternativa traer un facilitador externo al equipo (por ejemplo otro *ScrumMaster*). Como último, para múltiples equipos con muchos *SM*, es un buen ejercicio que un *SM* actue de facilitador para otro equipo, por ejemplo.

19.3. Enfoque

La *Sprint Retrospective* es, como todos los rituales, un proceso. Por lo tanto, podemos definir ciertos *inputs* y *outputs* mismo. Como *inputs*, sabemos que tendremos el foco y la data objetiva recolectada previamente. También, cada miembro traerá consigo datos subjetivos (percepciones del *Sprint*). Además, podemos considerar a los ejercicios de la retro como *input*, como también **un *Backlog* donde estén recolectados los resultados de retros previas**, llamado *Insight Backlog*. Éste Backlog se divide en 3 áreas: “Medidas a Seguir Haciendo”, “Medidas a Dejar de Hacer” y “Medidas a Intentar este *Sprint*”.

Como *outputs*, la retro produce entradas en el *Insight Backlog* representativas de las mejoras a tomar, además de un aumentado sentimiento de compañerismo en el equipo, producto de una buena autocrítica y una sensación de progreso constante dada la naturaleza iterativa del proceso. Cabe aclarar que ésto presupone que la retro se realiza con éxito y de forma pacífica, donde se quiere mejorar al producto de forma conjunta sin agendas personales en el medio.

Además de ésto, podemos definir una cierta estructura para el proceso, definiendo los pasos a continuación.

19.3.1. Crear la Atmósfera

La retro es un desafío que pone al equipo entero “bajo el microscopio”, con el fin de establecer mejoras iterativas en el desarrollo ágil. Para evitar que ésta sea una experiencia desagradable, se debe crear una atmósfera que permita e incentive la participación de todos, sin miedo a retribución por parte de otros miembros del equipo. Para lograr ésto, es conveniente establecer ciertas reglas que ayuden a entender que el foco está en la organización del equipo y del proceso, no en criticar individuos personalmente.

Habrán ciertas ocasiones donde los problemas serán entre personas: la Retrospectiva no es lugar para resolverlos. La retro sirve para mejorar el proceso de *Scrum* del equipo, no para asignar culpa individual a los miembros. Las reglas del proceso deben enfatizar éste punto por los demás. Además de eso, se debe generar una participación activa por parte de la mayor cantidad de miembros posible, para evitar que la gente tome un rol pasivo. Ésto puede lograrse utilizando preguntas disparadoras, con el objetivo de simplemente de hacer que la gente comience a hablar.

19.3.2. Compartir Contexto

Es importante que todos los miembros del equipos tengan una visión compartida de lo que ocurrió en el *Sprint*. Muy probablemente, cada uno tenga una perspectiva personal de los eventos que ocurrieron, y

es importante que estas no dominen por sobre la perspectiva compartida. Caso contrario, la retro podría transformarse en una sesión concentrada en opiniones en vez de una visión conjunta.

Para compartir el contexto, se debe focalizar la retrospectiva en una vista objetiva a gran escala del *Sprint*. Para esto, es importante compartir la data objetiva más importante relacionada al objetivo de la retro, como por ejemplo número de defectos, *PBI's* estimados, *PBI's* completados, etc. Esto no significa que la visión subjetiva de cada uno no debería ser tomada en cuenta, por lo contrario, también es importante. Para poder desarrollar una visión compartida de forma objetiva y subjetiva, existen dos ejercicios muy comunes: el *Event Timeline* y el *Emotions Seismograph*.

Event Timeline & Emotions Seismograph

Un buen enfoque para la *Event Timeline* consiste en trazar una línea en un pizarrón o pared dividida por los días del *Sprint* (equipos distribuidos pueden usar pizarra online). A cada día, cada miembro del equipo puede asignarle eventos significativos (con sticky-notes por ejemplo). Ejemplos de eventos pueden ser “X volvió de vacaciones”, “Se rompió la build de producción”, o cosas más leves como “Trabajo interrumpido para fixear bug de producción”. Para diferenciar la intensidad/importancia de los eventos, se pueden usar colores en las notas. Otro uso para los colores puede ser para categorizar el tipo: eventos técnicos, organizacionales, personales (pero que afectaron el *Sprint*), etc.

En conjunto con la línea de tiempo, muchos equipos crean algo llamado *Emotions Seismograph*. Éste se ubica debajo de la *Event Timeline*, y sirve para reflejar el estado emocional de los participantes en relación a como fue evolucionando el *Sprint* a lo largo de su duración. Se realiza dibujando una curva a lo largo de la línea de tiempo provista por la *Event Timeline*, en relación a tu estado emocional en ese momento. Como se puede ver, la *Event Timeline* representa una parte objetiva de los datos, mientras que el *Emotions Seismograph* toma en cuenta la parte subjetiva.

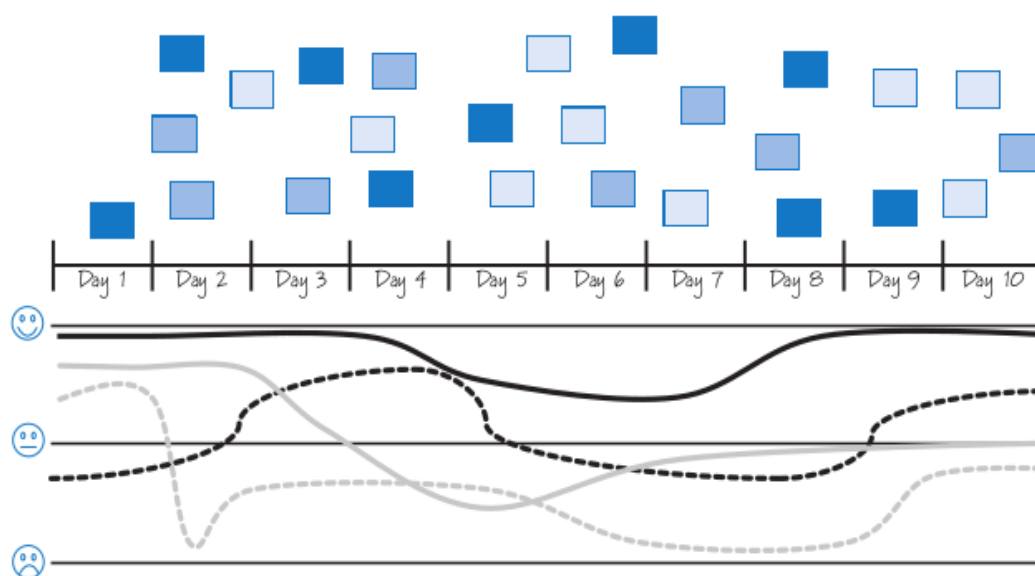


Figura 15: *Event Timeline* arriba y *Emotions Seismograph* abajo.

19.3.3. Identificar Percepciones, Mejoras y Conocimiento

Una vez que se tiene un contexto compartido, se puede empezar a buscar mejoras en el proceso. Para realizar esto, es importante poder mirar al proceso como un todo (visión *macro*) y no detener toda la atención en un detalle particular. Para comenzar, una buena estrategia es que todos los participantes hagan una especie de *brainstorming* de percepciones que se les ocurran, una vez que se haya analizado toda la data de la *Sprint Retrospective*. También se pueden utilizar los elementos del *Insight Backlog* provenientes de retros pasadas, como para pensar mejoras iterativas sobre mejoras pasadas. Todas estas nuevas mejoras que cada uno piensa se deberían bajar a algún formato organizable, para el paso siguiente.

Cada uno, por su cuenta, debería ahora agrupar las nuevas mejoras por similitud, en un ejercicio conocido como *Silent Grouping*. Esto luego puede ponerse en común y utilizarlo como pie para establecer en qué parte

del *Insight Backlog* irán las nuevas mejoras (si son cosas para probar, para hacer permanente o descartar). Una vez analizado ésto, debemos determinar acciones al respecto.

19.3.4. Determinar Acciones

Una vez que establecemos nuestras percepciones²⁰ o conocimientos de lo ocurrido en el *Sprint*, es necesario extraer de ellas acciones para realizar en la siguiente iteración. Éste también es un buen momento para revisar las acciones que iban a realizarse en el *Sprint* actual y determinar si se hicieron o tuvieron fruto. Si ni siquiera se comenzaron, podría decidirse priorizar el accionar sobre las medidas establecidas previamente, o tal vez descartarlas pues no valió la pena ni siquiera comenzar a realizarlas.

Eligiendo Percepciones

En una *Sprint Retrospective* habitual, suele ser común identificar mas lugares dónde se puede mejorar que los que el equipo puede atacar en un *Sprint*. Por lo tanto, es importante priorizar qué es lo que se quiere intentar mejorar, eligiendo criterios como “lo que el equipo considere más importante” ó “aquellas medidas que despierten mayor pasión e interés en el equipo”. Si bien las medidas más importantes es un buen criterio, lo mejor suele ser ir por aquellas que energizen y apasionen más al equipo, pues es mucho más probable que se generen fructíferas acciones para éstas (claramente con cierto cuidado, si algo es tan crítico que bloquea el funcionamiento normal del *Sprint*, seguramente se ataque eso). Una buen manera de decidir entre equipos se conoce como *Dot Voting*, método en el cual cada persona recibe varios puntos, con un color distinto para cada persona. Simultáneamente, todos ponen la cantidad de puntos que quieran sobre las percepciones a atacar, y ganan las que mayor cantidad de puntos tengan.

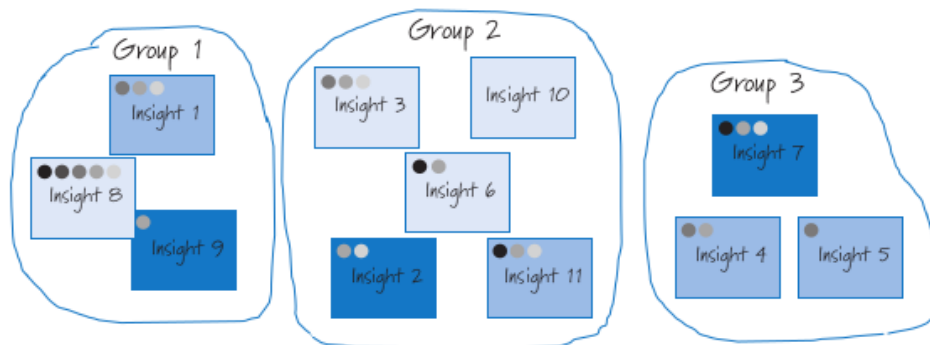


Figura 16: *Dot Voting* con *Insights* = Percepciones.

La respuesta de *cuántas* atacar a la vez depende mucho de la capacidad que el equipo quiera destinar a mejoras de la retro. Las cosas a considerar en éste análisis son las horas del siguiente *Sprint*, la criticidad del mismo (en relación a *features*), y otros temas como si se decide dedicar tiempo a implementar acciones de retros previas en éste *Sprint*. Para saber cuánto tiempo disponible se debería aloca, es importante el input del *PO*, por lo que es bueno que él participe de la retro también. Si el equipo de *Scrum* no dedica tiempo a accionar y monitorear éstas percepciones, es muy probable que queden en el olvido.

Decidiendo Acciones

Con la capacidad y percepciones a atacar ya aclaradas, se pueden decidir acciones a tomar sobre las mismas. Por ejemplo, si una percepción es “Tardamos demasiado tiempo en darnos tiempo cuando se rompe la *build* del código”, una acción puede ser “Que el servidor envíe un mail de forma automática cuando se rompe la *build*”. En general, las acciones suelen ser a nivel *task* de *Scrum*, aunque no todas tienen que necesariamente serlo. Por ejemplo, una percepción puede ser “Mejorar el respeto del tiempo del equipo llegando a la *Daily* a tiempo”, cuya acción correspondiente es “El equipo de *Scrum* debería esforzarse para llegar a tiempo”. Ésta acción no es como una *task* de *Scrum* y no reduce la capacidad del equipo, pero aún así representa una acción a realizar.

Otras acciones pueden ser bloqueantes del equipo a resolver por el *ScrumMaster* junto a otra persona o

²⁰Por ejemplo: “Estamos gastando demasiado tiempo en el proceso de *Code Review* sobre funcionalidades no críticas”

departamento. Por ejemplo, la percepción podría ser “No podemos pasar los *PBI's* a *Done* porque necesitamos la última versión de un software *third-party* para el testeo”, cuya acción asociada es “Nina (*SM*) con nuestro departamento de ventas para obtener la última actualización del vendedor”. Ésta acción no requiere tiempo del *devteam* pero sí del *SM*, por lo que bajará su capacidad disponible y tal vez requiera más de un *Sprint* al depender de terceros para su completitud.

Al determinar las acciones sobre las percepciones, es importante saber que es posible que no se puedan solucionar de forma inmediata. A veces, la mejor solución es investigar en base a la percepción en el *Sprint* actual para determinar una acción a realizar en el siguiente. Por ejemplo, con la percepción “Nos preocupa que dos componentes completamente testeados fallen al combinarse en una suite de tests multi-componente donde cada componente es, en teoría, individualmente testado”.

Insight Backlog

Muchos equipos crean un *Insight* (ó *Improvement*) *Backlog* para ubicar preocupaciones que no pueden ser atacadas en el *Sprint* actual. Con un poco de *grooming*, éste *Backlog* puede contener percepciones valiosas que pueden ser priorizadas frente a las nuevas que vayan ocurriendo. También puede usarse como historial de mejoras completadas, para evaluar el proceso de la retro a lo largo del tiempo. Algunos equipos deciden no hacerlo, pensando que si una percepción es realmente importante, se puede descartar sin temor pues simplemente volverá a aparecer en el siguiente *Sprint*.

19.4. Seguimiento sobre el *Sprint*

Para asegurarse que las acciones a realizar no queden solo en la retro, debería haber un seguimiento de las mismas por parte del equipo. Elementos como “que todo el equipo intente estar en el *Daily Scrum*” solo necesitan recordatorios por parte del *ScrumMaster*. Otras acciones que requieren tiempo del *devteam* deberían ser tomadas en cuenta en la *Sprint Planning*. Un enfoque que suele resultar es volcar en el *Sprint Backlog* tareas correspondientes a las acciones de la retro antes de empezar con las nuevas *features*. De ésta forma, el equipo puede medir de forma más precisa la capacidad que tienen para entregar nuevas *features* en el *Sprint*.

Los enfoques que no suelen funcionar son aquellos que tienen su “plan de mejoras” separado del flujo de *features* a entregar en el *Sprint*. Ésta mentalidad tiene el problema de que, en la mayoría de los casos, las mejoras quedarán completamente subordinadas a las *features*, por lo que hay más probabilidad de que mucho del trabajo hecho en la retro quede en el olvido. Para asegurarse que las acciones correspondientes a las mejoras se realicen, no se debe separar, sino que se debe integrar!

19.5. Problemas de la Retrospectiva

Si bien éste ritual es simple, puede tener una serie de problemas, como por ejemplo:

■ No hacer la retro ó tener poca asistencia:

1. Por ejemplo, personas asignadas a múltiples equipos podrían no poder asistir a éstas reuniones por problemas organizacionales de tiempo, los cuales deberían ser resueltos con los managers correspondientes lo antes posible.
2. De forma similar, si el equipo tiene integrantes de forma remota, se deberían acomodar los ejercicios o locación de la retro para intentar que sea lo menos incómoda posible.
3. Otra razón puede ser equipos que todavía no creen en la mejora continua de *Scrum* y piensan que todo tiempo fuera de código o testeo es desperdicio. Ésta actitud es errónea pues, dada la agilidad de *Scrum* y la urgencia de los *Sprints*, no se suele tener un momento para “parar la maquinaria y hacer introspección” que no sea la retro.
4. Por el contrario, también puede haber equipos que piensan que llegaron al “pináculo” de *Scrum*, por lo que ya no la necesitan más. Si bien es cierto que mientras más se mejora encontrar medidas puede ser más difícil, debería ser imposible tener un equipo “inmejorable” (ésto suele ser

confundido con simplemente estar cómodo con la forma de trabajar).

- **All fluff, no stuff:** Mucho trabajo en la retro pero sin medidas accionables al final. En éstas situaciones, es mejor traer un facilitador experto de otro equipo que ayude a mejorar los resultados finales de la retro con su guía y mirada externa.
- **Ignorar los puntos críticos:** Puede ocurrir que haya una situación de “elephant in the room”, donde todos saben que hay un problema crítico y bloqueante, pero nadie se anima a discutirlo. En éste caso, el SM debería tomar una posición de líder y trabajar para mejorar la seguridad del equipo para que se pueda discutir y solucionar de forma eficiente.
- **Facilitador pobre:** Ésta situación ocurre cuando un *ScrumMaster*, tal vez nuevo en su trabajo, intenta dar lo mejor pero no produce resultados. Como solución, es mejor traer un facilitador experto de afuera que actúe de *coach* para el nuevo SM.
- **Ambiente depresivo y agotador:** Puede ser que los *Sprints* estén saliendo mal de forma constante, y la retro sea para el equipo simplemente una forma de revivir aquello que ya para éste punto quieren evitar. Para solucionar esto, mas allá de los problemas fuera de la retro que lo estén causando, es útil pensar un mejor plan para crear una buena atmósfera para encarar las mejoras concentrándose en generar positividad para el equipo. También es posible que las retos sean deprimente porque algunos miembros las usan para echar culpas a los demás. El facilitador debe extinguir ésta actitud lo más rápido posible para prevenir una cascada de gente que se culpa entre sí.
- **Sesión de quejas:** Hay personas que pueden ver la retro como el momento para quejarse de como están las cosas actualmente. No tienen deseo de mejorar, simplemente necesitan hacer catársis en un lugar y utilizar la retro para quejarse. Como solución, es mejor invitar a la retro a aquellas personas que puedan impulsar un cambio positivo en el proceso de *Scrum*, y luego el facilitador puede hablar en privado con aquellas personas quejosas para lograr el cambio de actitud necesario para que se integren al ritual.
- **Demasiado ambiciosas:** Frecuente con equipos nuevos y energéticos, notan muchas áreas donde mejorar y ponen objetivos ambiciosos, sólo para decepcionarse cuando no llegan a cumplirlos más adelante. En ésta situación, es responsabilidad del *ScrumMaster* recordar a los participantes de la retro la capacidad disponible para realizar las mejoras y moderar sus ambiciones para efectivizar los cambios de forma concreta.
- **Sin seguimiento en el *Sprint*:** Éste suele ser el mayor problema de todos. Sucede cuando no se dedica tiempo a trabajar en las mejoras definidas en la retro, lo cual invalida el ritual el general. EL *ScrumMaster* tiene el rol de liderazgo en el equipo en la ayuda que debe proveer para que mejoren su proceso ágil, por lo que debe trabajar agresivamente con el equipo para identificar la causa que ésta haciendo que no se trabaje sobre las mejoras planeadas para removerla.

20. Fuentes

- *Essential Scrum* : <https://www.amazon.com/Essential-Scrum-Practical-Addison-Wesley-Signature/dp/0137043295>
- *Scrum: a Breathtakingly Brief and Agile Introduction*: <https://www.amazon.com/Scrum-Doing-Twice-Work-H/dp/038534645X>
- *The Mythical Man Month*: <https://www.amazon.com/Mythical-Man-Month-Software-Engineering-Anniver/dp/0201835959>