UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

Diplomatura en Metodologías Ágiles

Módulo 2

Lean, KanBan y Scrum

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 2 - Unidad 8

Frameworks para Escalar Ágiles

Presentación:

El Módulo 3 introduce a los participantes a desafíos de Agile avanzado.

Se busca que comprendan los desafíos de Agile en la transformación de grandes organizaciones, mejoras continuas a largo plazo, métricas. También una visión a futuro que habla que, en 15 años, 2035, todos los productos dentro de su núcleo tendrán una parte de Sistemas.

Dentro de Agile avanzado otro de los puntos es la aparición del Coaching como un nuevo rol clave en las organizaciones.

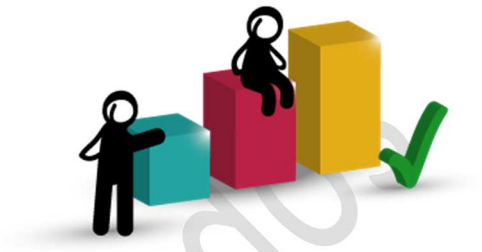
Objetivos:

Que los participantes logren:

- Conocer los principales marcos ágiles para organizaciones.
- Entender la visión Agile en toda la organización.
- Pensar la mejora continua a mediano y a largo plazo.

Bloques temáticos:

1. Introducción a DevOps. Automatización. Delivery Continuo
2. SAFe: Roles, artefactos, ceremonias, prácticas. Ventajas y limitaciones.
3. LeSS: Roles, artefactos, eventos y prácticas
4. El modelo Spotify



Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

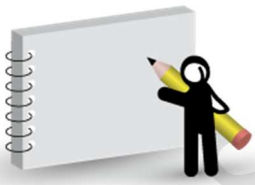
- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

** El MEC es el modelo de E-learning colaborativo de nuestro Centro.*



Tomen nota:

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

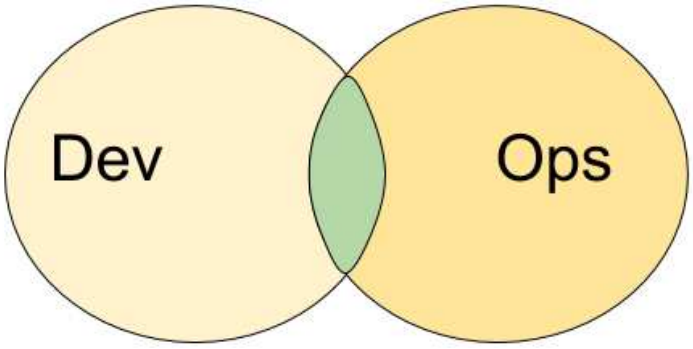
Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Introducción a DevOps. Automatización. Delivery Continuo

Filosofía

Comenzaremos con lo que podría darse como una definición, ya que hay muchas y distintas, de lo que es **DevOps**. Este concepto puede estar enmarcado como un movimiento basado en los principios Lean y en el manifiesto ágil, para abordar la colaboración del área de desarrollo (*development*) con las áreas de operaciones / infraestructura (*operations*), y que tiene como visión la ***entrega temprana, de valor y calidad a los clientes***. De la combinación del nombre de dichas áreas se creó la nominación DevOps.

Surge como respuesta a los inconvenientes intrínsecos de las estructuras organizacionales basadas en silos funcionales, como por ejemplo las mencionadas áreas de desarrollo y operaciones, que buscan la eficiencia de cada una de ellas, pero no así en el sistema completo de la organización.
Entonces, **¿Qué no es DevOps?** No es una persona en particular, no es un desarrollador haciendo el trabajo de operaciones, tampoco lo es al revés, una persona de operaciones realizando la actividad de un desarrollador, tampoco es un conjunto de herramientas avanzadas ni un rol particular, ni un sector, ni un Gerencia. Es una corriente, una cultura que se basa en la **colaboración** mutua por parte de las personas que integran, principalmente, dos áreas fundamentales en TI (Tecnología de la Información) que es **Desarrollo** y **Operaciones**. El movimiento se basa en crear una cultura de colaboración para acelerar los tiempos de entrega de software en manos de cliente/usuario, y para ello se sustenta en una visión y una serie de principios, los cuales irán modificando y transformando la organización con el soporte de las herramientas y las prácticas que se ven hoy en día. Lo que se intenta transmitir es que esa cultura busca promover los comportamientos y valores que posee la organización y las personas que la integran.



Por lo tanto, la incorporación de una serie avanzada de herramientas no instalará una cultura DevOps en la organización, pero sí podría verse como una aproximación a que los equipos puedan interactuar de otra manera, pero las herramientas por sí solas no podrán alentar a un cambio cultural en la organización. Dicha interacción es la que facilita el pulido de las asperezas que puedan existir entre Desarrollo y Operaciones, beneficiando de manera holística a la organización con entregas tempranas y de calidad de software.
Lo que transmite y promueve la cultura DevOps dentro de la organización son una serie de componentes que veremos a continuación.

Componentes principales

Comunicación abierta

La cultura de la organización se debe basar en el debate y en la discusión a través de canales claros de comunicación, evitando lo más posible medios, herramientas y procesos que no alienten esto. Por ejemplo, sistema de tickets para implementar, procesos y flujos documentados nunca conversados cara a cara, cadenas de mails, minutas detalladas, etc. Los equipos y las organizaciones vinculadas a DevOps debaten sobre el tiempo total de entrega del producto, la calidad del software, qué construir, medición del impacto de lo que se entrega, cómo hacer esta medición, hacer un uso más eficiente de los recursos, etc.

Alinear los incentivos y la responsabilidad

Los equipos y las diferentes áreas deben estar impulsadas por un propósito, por una visión que permita alinear los objetivos específicos de cada una de ellas. Esa visión es la de construir productos de alta calidad, con un alto valor para el cliente y entregarlos de la manera más rápida posible. Dada una visión así en la organización, pierde un poco el sentido de los objetivos en los silos funcionales de las organizaciones. Los desarrolladores no serán los únicos responsables por generar el código para que el producto sea bueno, las personas de operaciones no serán castigadas por hacer un despliegue a producción que no ocurrió como se esperaba y los testers no serán los únicos

responsables de agregar calidad. En lugar de esto, todos tienen la visión de construir el producto lo mejor posible y todos aportan en la creación de un ambiente, una estructura que promueva la colaboración, valiéndose de los éxitos y de los fracasos en conjunto.

Respeto

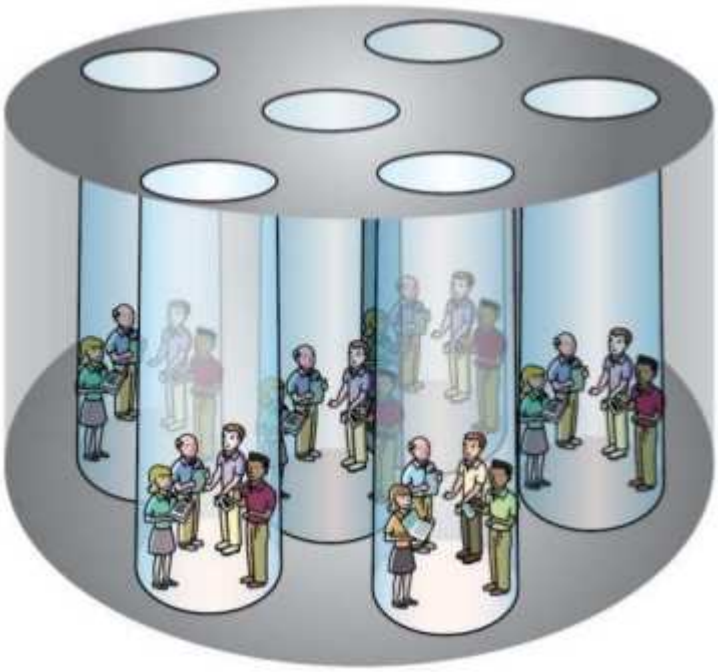
Los miembros de la organización se deben respetar unos a otros, no importa si están en el mismo equipo o no. No es necesario que una persona le agrade a la otra, pero sí que entre ellas puedan trabajar en un ambiente de respeto y apertura hacia distintos puntos de vista. Cambiar el paradigma tradicional que el respeto va por el lugar en la pirámide imaginaria organizacional y respetar a cada persona por el rol que cumple, por el valor que aporta, por su desempeño, por su actitud colaborativa y por el hecho de ser persona.

Confianza

Este es un componente fundamental para crear una cultura DevOps. Operaciones debe confiar en que los Desarrolladores harán el mejor código a su alcance, los Desarrolladores deben confiar en que los testers no levantan incidentes porque están conspirando contra ellos, los Líder de Proyecto deben confiar en que Operaciones está haciendo su trabajo y el feedback de información que les brinda es los suficientemente objetivo.

Reducción de Silos Organizacionales

El concepto de silos en las organizaciones se entiende como la dificultad (a veces, incluso imposibilidad) para trabajar eficientemente entre las distintas áreas o unidades de negocio, básicamente cada "cual" mira lo suyo.



Los silos son un "clásico" en organizaciones tradicionales y son un síntoma de una excesiva y mal equilibrada organización. Es lo que se conoce como una Organización muy "Verticalista". Si bien esta verticalización tiene ventajas (por ejemplo, un gran foco en cada "silo") en general en la mayoría de las organizaciones frecuentemente imposibilita que las cuestiones entre áreas se resuelvan a niveles medios o bajos. Las situaciones tienden a elevarse a lo alto del silo para su resolución en un nivel muy alto de la pirámide.

En el mundo de IT, Operaciones y Desarrollo, en el pasado eran considerados "Silos". Claramente DevOps busca reducir o eliminar las separaciones duras entre áreas.

Aceptar "fallas" como algo habitual

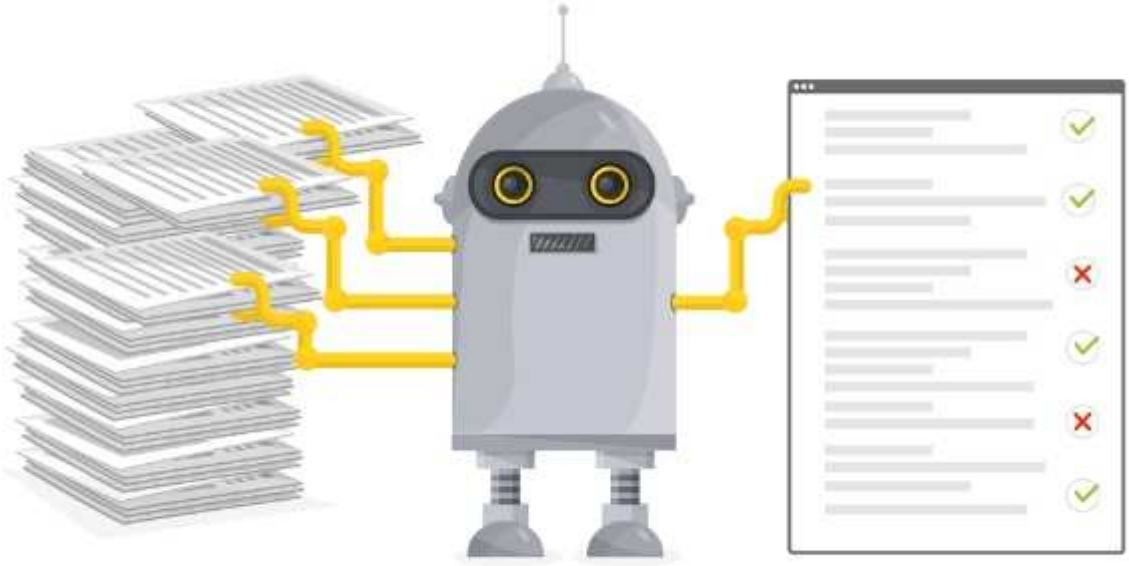
En la filosofía DevOps las fallas o problemas son algo habitual y no deben tener una connotación negativa.

De acuerdo a la Real Academia Española, la palabra "problema" debe ser entendida como: una "cuestión que se trata de aclarar. Una proposición o dificultad de solución dudosa. Un conjunto de hechos o circunstancias que dificultan la consecución de algún fin. Un planteamiento de una situación cuya respuesta desconocida debe obtenerse a través de métodos científicos" o bien "un disgusto, una preocupación".

No hay nada mejor que enfrentar un problema dado que los problemas exigen que mejoremos, los problemas ponen a prueba la creatividad, el ingenio, la pasión, el estado actual de los datos que tengamos, etc.

Las fallas no deben tomarse como algo negativo, aunque pueden causar malestar o disgusto, sino que deben contemplarse como algo que ocurre y ocurrirá y por lo tanto debemos estar cada vez mas preparados y lograr que se pueda aprender de cada situación.

Automatización



El movimiento DevOps es reconocido por la instauración de herramientas de software para la automatización de pasajes de código en ambientes y la automatización de pruebas. En el comienzo del camino hacia la implementación de las prácticas de Integración y/o Entrega Continua se debe prestar mayor atención a lo que se va a automatizar. Lo primero que sugiero automatizar es el *build* de la aplicación y el *deploy* a un ambiente distinto. Debe ser posible de configurar la aplicación para que el servidor de **Integración Continua** "escuche" en un determinado *branch* de repositorio de versionado de código y que al ejecutar un *commit* un desarrollador realice el *merge* y el *build*. Al generar el *build* se obtendrán los binarios/ejecutables de la aplicación los cuales se copiarán a otro ambiente distinto del servidor de **Integración Continua** y lo más parecido a uno de producción en el cual se le aplicará la configuración necesaria para que la aplicación logre ejecutarse. De tener más ambientes para distintos propósitos, la automatización que se escoja debe permitir que al transferir el binario a los distintos ambientes no sea necesario recompilar (y no se debe hacerlo) sino que hay que aplicarle la configuración dependiendo del ambiente en el que se realice el despliegue. Para

lograr esto recomiendo fuertemente que se genere una aplicación básica al estilo “hola mundo” y se haga el despliegue al ambiente (esta tarea es conocida como *deploy* técnico) para comprobar lo más pronto posible las configuraciones que haya que ajustar. Todas las advertencias y errores que se fueron detectando a lo largo de la instalación y configuración es muy importante que sean anotadas para futuras ocasiones.

Una vez logrado el *merge* y la compilación automática del código, el paso siguiente es la automatización de las pruebas unitarias y el análisis de código. La configuración de las pruebas unitarias no requiere mucha complejidad es un paso más al procedimiento de automatización de la compilación y despliegue que se realiza antes de que se realice la integración de código. Recordemos que estas operaciones generalmente se realizan en el servidor de Integración Continua. Luego es conveniente instalar alguna aplicación que pueda leer el código fuente y realice un análisis de “salud” del mismo. Según el grado de madurez que se tenga con la práctica de Integración Continua en la organización y en el equipo en la cual se esté llevando a cabo se deberá configurar los valores que puedan ser alcanzables para métricas de análisis de código como por ejemplo “complejidad ciclotímica”, “posibles vulnerabilidades”, y algunas otras muy útiles. Esto quiere decir que el no alcanzar los índices de los parámetros de análisis de código se podría considerar como si hubiese fallado la compilación.

El paso siguiente, es configurar la automatización de las pruebas de aceptación. En esta etapa se ejecutarán las pruebas que fueron definidas tanto para el equipo como para el negocio / usuario las cuales nos aseguran que nuestro sistema continúa entregando valor a pesar de estar en constante cambio. La automatización debería abarcar no sólo pruebas funcionales (la que entrega valor al negocio) sino también pruebas no funcionales tales como pruebas de capacidad, pruebas sobre el consumo de servicios, pruebas de seguridad, pruebas de fallas de infraestructura, etc. Comenzar con un conjunto, aunque sea pequeño, de estas pruebas al comienzo del desarrollo del pipeline es de gran beneficio.

Es necesario diseñar la arquitectura de la aplicación para que sea posible la automatización del conjunto de datos necesario para que la aplicación funcione y para que las pruebas puedan ser ejecutadas. La mayoría de los pequeños programas que construyen los desarrolladores para poder probar su desarrollo son candidatos a que sean scripts automatizables y que sean incorporados al ciclo de Integración Continua. Así mismo, las actualizaciones a la base de datos debieran estar bajo el mismo sistema de control de versiones y que la incorporación de algún cambio propague con algún script de automatización. En este curso no abordaremos el tema de la gestión de los datos en

la práctica de Integración Continua ya que es un tema por demás extenso y a su vez la implementación de esta característica depende de la arquitectura de la aplicación en cuestión. En ocasiones, es posible versionar el cambio que se realiza en la base de datos, pero si esta se encuentra en un producto que ya está en producción es muy difícil volver el cambio atrás.

La ventaja de contar con un proceso automatizado es que ante cualquier error será más visible su origen y además cualquier procedimiento automático será menos propenso a errores que si fuera manual. Tomemos por ejemplo los escenarios en los que una organización realiza los despliegues a producción de manera manual, está se encuentra con un alto riesgo de introducir errores, con lo que será más costoso detectarlo ya que si fuese automático existiría un registro del estado de cada operación posibilitando determinar la causa del inconveniente que se haya producido. En DevOps se busca eliminar cualquier paso manual que sea repetitivo y/o que posea riesgo para la organización, ya que es propenso a introducir errores. Tal como se ha mencionado anteriormente, es útil obtener métricas de tareas o pasos del proceso para poder saber lo que hay que optimizar, incluidas las tareas manuales y con ello se tendrá información suficiente para diseñar una estrategia de mejoras.

Delivery Continuo

La Integración Continua excede la instalación y configuración de herramientas de software, es una práctica y como tal hay ciertos aspectos que hay que tener en cuenta. Las herramientas de software por sí solas no proporcionan el cambio necesario para implementar un modelo de DevOps en la organización, así como tampoco por realizar la compra de una licencia de software que haga mención a Integración Continua, sin poseer los conceptos intrínsecos se estará destinado a resultados mediocres.

La práctica de Integración Continua surge de la metodología XP (Extreme Programming). Como se podrá observar si bien la práctica de Integración Continua se está fomentando mucho más en los últimos años, no es algo nuevo. El concepto principal de esta práctica es que el código esté siempre en un estado estable para ser pasado a producción y la manera de que esto suceda sin demasiado riesgo y costo es cuando se integra rápidamente código nuevo, asegurando a través de las pruebas escritas que en el nuevo código integrado no se están introduciendo fallas. Parece bastante obvio, pero lo que generalmente sucede son dos escenarios, no son los únicos, pero si los que más he visto. Uno, es que los equipos de desarrollo realizan sus cambios en repositorios de código distintos o bien en ramas de trabajo divididas por equipo (con un fuerte acoplamiento de código) por lo cual, al finalizar el trabajo, por ejemplo, de dos o más equipos se requiere que se integre lo desarrollado en un único repositorio o rama de código (branch) que los unifique. Este escenario es el que se conoce como **fase de integración** y conlleva a múltiples inconvenientes. El más sustancial es que en esta fase en general no se conoce cuando finaliza ya que los problemas que podrían existir por la integración son desconocidos hasta el momento de llevarla a cabo y esos inconvenientes la mayoría de las veces son costosos de resolver ya que se requiere volver hacia atrás mucho trabajo realizado. El segundo escenario, es que el trabajo de cada miembro del equipo se divide en ramas distintas de código (branch por feature) o no integran sus cambios frecuentemente a la rama principal, con lo cual cuando se requiera unificar el trabajo de varias personas suele ser bastante similar a una fase de integración, pero con la posibilidad de volver los cambios hacia atrás, en el caso de que estén en un mismo repositorio.

La práctica de Integración Continua se basa en una serie de reglas y acuerdos con el equipo de desarrollo, pero antes de incursionar en la mención de estos temas veremos qué requisitos se necesitan. Se requiere de tres ámbitos para la realización de esta práctica:

1. **Tener un repositorio de control de versiones** (GIT, Mercurial, SVN, etc). Con respecto de poseer un repositorio de control de versiones. Lo que hay que tener en cuenta es que todo

proyecto, por más que se crea que es pequeño, debe tener un repositorio de control de versiones y de ser posible versionar todo lo que incluye el proyecto (código, scripts de base de datos, scripts de deploy, documentación, etc).

2. **Poder generar un build (compilación de la solución del código) de manera automática.** La automatización es fundamental, pero en lo posible debe ser independiente del software que utilizan los programadores para el desarrollo (IDE). Esto no quiere decir que no se utilicen, por lo contrario, úsenlos ya que tienen una versatilidad importante, pero se debe prestar atención a que la ejecución del proceso de Integración Continua sea posible instanciarlo desde múltiples entornos.
3. **Tener un acuerdo con el equipo**. Tal como se mencionó anteriormente Integración Continua es una práctica, no una herramienta. Requiere de compromiso y disciplina por parte del equipo. En el acuerdo se debe incluir que los miembros del equipo de desarrollo realicen al menos un *commit* al día y lo publiquen, es decir integrando código continuamente. La prioridad más alta del equipo es que si hay un fallo en la línea principal dejaran lo que están desarrollando para solucionar el inconveniente. Si no existe un compromiso por parte del equipo es poco probable que se obtengan los beneficios de calidad que provee esta práctica.

Hasta ahora, lo que hemos visto son temas de surgimiento de esta práctica, sus requisitos, lo interesante como también hemos mencionado es que la Integración Continua no se refiere sólo al software propiamente dicho, sino a las actividades que deben desenvolver los integrantes del equipo de desarrollo para que esta práctica sea efectiva. Para ello enumeraré algunos puntos claves y básicos.

- **Commits frecuentes**:
  Muchos equipos que conocí que decían hacer integración continua hacían un *commit* una vez por semana o lo hacían todos los días, pero cada desarrollador en ramas diferentes (Branch por *Feature*) y recién lo integraban al finalizar la iteración. Mientras más seguido hagamos el *commit* más continuamente estaremos integrando código, a su vez mientras más frecuente sea este cambio por cada desarrollador menor será la porción de código que subirá y de tener algún error menor será el costo ya que hacer un roll back (volver al estado anterior al error) será rápido y menos doloroso. Este concepto nos remite nuevamente a considerar al desarrollo con la noción ya vista de desarrollo orgánico. Por lo tanto, lo recomendable es que se suban los cambios al menos una vez al día.

- **Tener una batería automatizada de pruebas**:

Este punto es de mucha importancia, sobre todo para las organizaciones que recién están adoptando, o están pensando en hacerlo, la práctica de Integración Continua. Integrar el código no significa sólo que los cambios de los integrantes del equipo se integren (*merge*) y compile la aplicación (*build*), aunque como inicio es un buen comienzo. Lo importante es tener una batería de pruebas que se ejecuten al realizar un cambio, pero antes de que se haga el *merge* en el repositorio de control de versiones actuando como una red aseguramiento de calidad ya que, si el cambio introducido podría hacer compilar la aplicación, pero está provocando que fallen, por ejemplo, test unitarios o test de componentes ese cambio debe ser rechazado. La batería de pruebas que se debería ejecutar debieran ser, mínimamente, Unit Tests, Component Test y Acceptance Test como práctica más madura. La ventaja que nos dará tener esta batería de pruebas automatizadas es tener feedback instantáneo al realizar un cambio. Si falla una prueba, el desarrollador sabrá exactamente en qué lugar tiene que revisar y qué cambio fue el que produjo la falla.

- **Mantener el proceso de compilación y el conjunto de pruebas en un tiempo corto**:
  Este punto es una recomendación muy útil, ya que hay que tener presente cuánto demora en ejecutar la compilación y la batería de pruebas en el proceso de integración continua. Lo recomendable es que no demore más de diez minutos, dado que, si el proceso demora mucho tiempo, los desarrolladores tenderán a evitar incorporar cambios frecuentemente o bien se podrían encolar múltiples cambios provocando que el build falle, con lo cual dificultará determinar cuál de dichos cambios provocó la falla. Para reducir los tiempos de compilación y ejecución de pruebas se suele dividir el proceso de integración continua en distintas etapas y ambientes.

- **Los desarrolladores deben administrar sus entornos**:
  La aplicación debería poder ejecutarse en los entornos locales de los desarrolladores y estos a su vez deberían poder ejecutar el conjunto de pruebas automatizadas en su propio ambiente. Si el desarrollador no posee una administración completa de su entorno, ante un fallo se podría hacer un cuello de botella al esperar que venga la persona encargada de darle los permisos necesarios para que lo solucione o bien para que tome cartas en el asunto.

  Por otro lado, al ejecutar el proceso de integración continua, el desarrollador debe poder partir un punto común conocido. Esto quiere decir que cada vez que un desarrollador obtenga el código para trabajar sea desde un punto en donde se ejecutaron todas pruebas necesarias y todas pasaron, es decir que la aplicación está en un estado estable. Este tipo de

práctica es altamente recomendable ya que tiende a reducir ampliamente los fallos en el servidor centralizado, ahorrando tiempo al equipo entero de desarrollo.

- **No integrar sobre un build que falló**:
  Si ocurre un fallo al momento de publicar un cambio se debe prestar total atención a que se solucione por la persona que subió dicho cambio. Si dicha persona no lo puede solucionar en un lapso de tiempo corto (tiempo acordado por el equipo) el desarrollador debe hacer un roll back para dejar la aplicación en un estado saludable o bien pedir ayuda al equipo para solucionarlo, pero nunca dejar que los builds fallen y acostumbrarse a que estén así. Debe ser regla mantener en estado saludable el código de la aplicación.

Por último, la recomendación es que, si se está por comenzar un proyecto nuevo, la incorporación temprana de un ciclo de Integración Continua es lo menos doloroso y costoso. La implementación Integración Continua en una aplicación heredada (legacy) es laboriosa pero no imposible, lo cual igualmente recomiendo su incorporación ya que la organización poseerá más visibilidad sobre los errores (de los cuales podrá obtener métricas), el equipo incrementará su velocidad poco a poco y la calidad tenderá a aumentar gradualmente generando más confianza en el producto. El proceso de integración y despliegue de software no debe ser riesgoso y altamente costoso, si lo es la recomendación es que mientras más doloroso sea más veces hay que intentar implementar la práctica para revertir el síntoma.

La entrega continua es más que la compra de una herramienta de software que permita hacer entregas a nuestros clientes de una manera rápida. La entrega continua o Continuous Delivery es un paradigma distinto en el cual se sostiene el negocio de una organización digitalizada o que posee una necesidad de ello. El principal obstáculo que trae aparejado la Entrega Continua en una organización que no tiene el suficiente nivel de madurez para llevar adelante esta práctica es la falta de lineamientos dentro las líneas de management y los equipos (silos). Los equipos de desarrollo están presionados para entregar software de una manera rápida, performante, con calidad y en algunos casos siguiendo ciertas normas regulatorias y por otro lado, están los equipos de implementaciones (operaciones / infraestructura) que analizan de cada implementación en producción su riesgo y en el mejor de los casos obtienen ciertas métricas de experiencias anteriores, datos muy importantes para una práctica concisa de DevOps ya que sin la medición y el monitoreo, no se sabría la situación actual de la aplicación o si los features que se entregaron realmente están generando valor (recordar de unidades anteriores la visión Lean y su ciclo). En organizaciones con poca madurez en los aspectos de DevOps, existe una tensión entre los equipos de desarrollo y

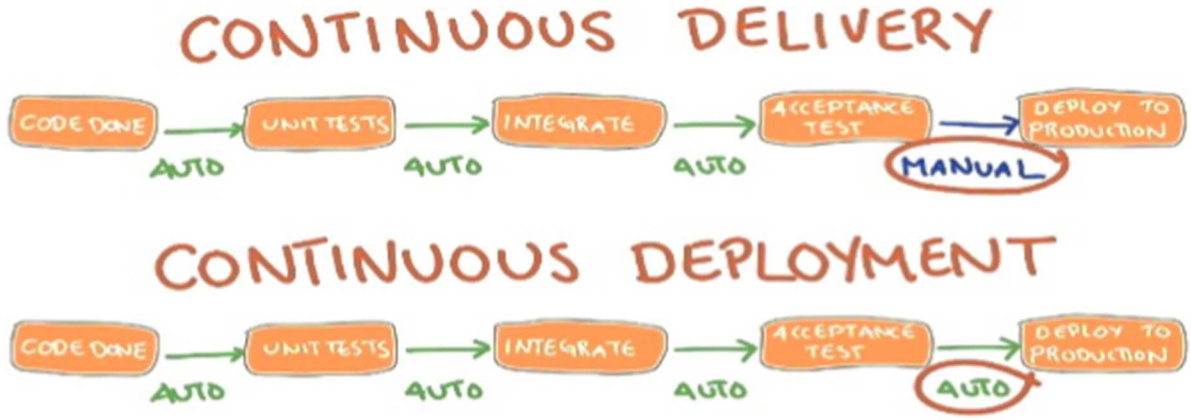
operaciones provocada por la falta de lineamientos estratégicos para obtener una ganancia de la manera más rápida posible, y a su vez, aprender de lo recientemente lanzado a producción (medirlo) para saber si hay que adaptar el producto (aprendizaje). Para ello, el pipeline de desarrollo se diseña de principio a fin involucrando a todas las partes de la organización, que van a agregar valor haciendo un lineamiento de objetivos acorde a como lo estableció el Negocio y Tecnología (IT). Esta manera de hacerlo, a través de un pipeline permite comenzar a diluir las barreras que existen en los silos de organizaciones medianas a grandes, mediante la colaboración mutua de equipos multidisciplinarios que realizan diferentes actividades.

La principal ventaja de las prácticas y técnicas vistas hasta aquí son utilizadas para diseñar un pipeline de desarrollo que permita disminuir el riesgo de las implementaciones, aumentar el feedack de una manera rápida y no tan costosa a través del desarrollo iterativo e incremental, con menos defectos y la automatización de pruebas e implementaciones reduciendo el ciclo de entrega.

La identificación de riesgos es un aspecto importante para la práctica de Entrega Continua y sobre todo para la organización que la adopte. Los principales aspectos que se deben tener en cuenta, no siendo los únicos, son los siguientes:

- Identificación de los riesgos principales del proyecto
- Estrategias para la mitigación de los riesgos identificados
- Estrategia para el seguimiento de los riesgos durante el curso del proyecto
- Definir una estructura para el reporte de estado de los equipos

Estos aspectos, en general, se suelen tratar en etapas iniciales del proyecto bastante antes del comienzo del desarrollo en la cual se conversan entre todos los involucrados de la organización, tanto el negocio como tecnología (comerciales, miembros de los equipos de desarrollo, de operaciones, de seguridad, de testing, ejecutivos, gerentes, etc.) y se deben continuar a lo largo de todo el proyecto, ya que los documentos que generen al inicio son documentos vivos, los cuales seguramente vayan cambiando a lo largo de la vida del proyecto y es suma importancia que se actualicen.



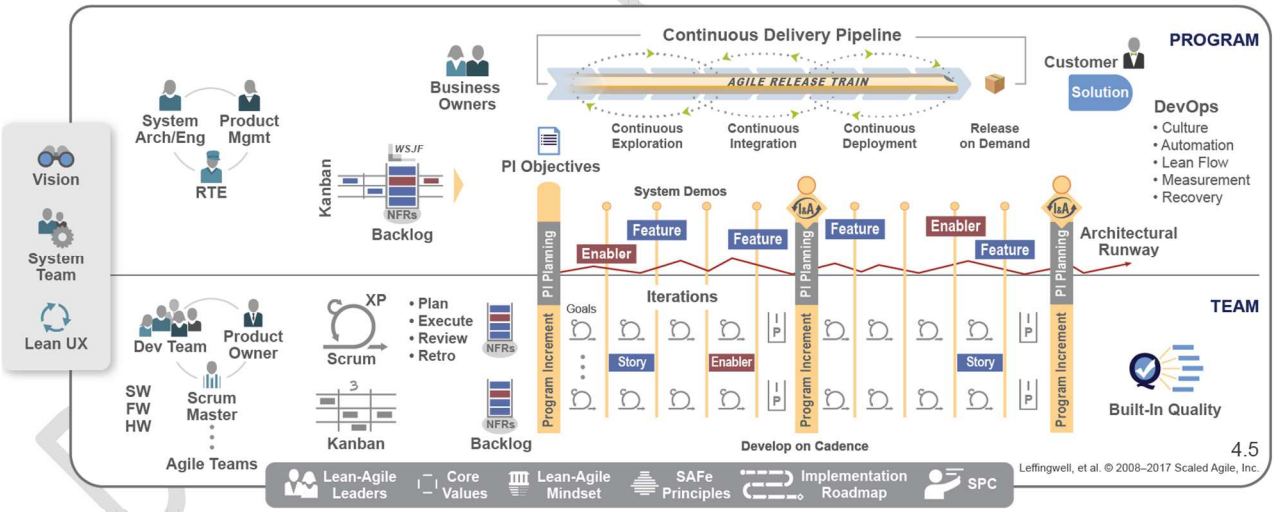
La esencia de la Entrega Continua es que la organización pueda tener la capacidad de entregar pequeños cambios de una manera incremental y continua, de manera de no sólo entregar software rápidamente, sino que además reduzca los riesgos asociados al introducir cambios en el sistema. La manera en que lo realizará será con lo el conocido "*One Click Deploy*" (implementar con un sólo click). Esto significa que se automatiza todo el pipeline de desarrollo incluyendo la salida a producción, pero con la diferencia de que alguien de la organización tendrá la posibilidad de elegir en qué momento implementar el software que está listo para salir a producción, es decir que para que se haga el pasaje a producción se requiere una aprobación.

La diferencia entre implementaciones continuas (Continuous Deployment) y Entregas Continuas es que en el caso de las primeras el ciclo está automatizado y es directo hasta la salida a producción. Es decir que no es necesaria la intervención de una persona para la salida a producción. Este tipo de práctica requiere un nivel de madurez muy alto en la organización asociado a la cultura ágil y de DevOps.

2. SAFe: Roles, artefactos, ceremonias, prácticas. Ventajas y limitaciones.

El concepto principal que introduce SAFe® para la coordinación entre equipos es el **Agile Release Train** (Tren de Entrega Ágil) o **ART**: un grupo estable de 5 a 12 equipos ágiles multi-funcionales (cada uno de ellos usando Scrum, XP, Kanban o combinaciones entre estos) que desarrolla y entrega incrementos de la **Solución** (el término que usa SAFe® para productos, servicios o sistemas) en iteraciones de tiempo fijo.

Este Tren describe el nivel de **Programa** (equipo de equipos, o conjunto de proyectos) en SAFe®, y por eso todos sus equipos entregan al final del ciclo un único **Incremento de Programa** (PI). El ciclo de PI se compone habitualmente de **cuatro iteraciones de desarrollo más una iteración final** de "Innovación y Planificación" (destinada tanto a mejoras como a solucionar problemas y errores). Para un sprint de 2 semanas (la duración más habitual) el Incremento de Programa implicaría un ciclo de **10 semanas** (o entre 8 y 12) para todos los equipos que trabajen sobre ese ART.



El esquema básico de SAFe®: Un conjunto de Equipos trabajando juntos dentro del mismo Agile Release Train, que ejecuta cierto Programa [*Nota: las imágenes incluidas no están traducidas al español porque la licencia de Scaled Agile Inc impide distribuirlas con modificaciones*].

Además de los roles existentes en Scrum a nivel equipo (Product Owner, Scrum Master, Equipo de Elaboración), SAFe® introduce algunos **roles a nivel Programa**, relacionados al **Agile Release Train**:

- **Release Train Engineer** (*RTE: Ingeniero del Tren de Entrega*): Es el coach y líder servicial del Tren (así como lo es el Scrum Master para los Equipos). Debe facilitar los eventos a nivel Programa, asistir a los equipos para que optimicen su proceso de entrega de valor, escalar los impedimentos grandes, administrar el riesgo y facilitar la mejora continua.
- **Product Manager**: Gestiona la Visión y el Roadmap (Hoja de Ruta) del Programa, así como también el contenido, priorización y valorización del Backlog de Programa (así como lo hace el Product Owner a nivel Equipos).
- **Arquitecto de Sistema**: Es una persona o equipo que define, diseña y evoluciona la arquitectura e infraestructura generales del Programa y/o la Solución, eligiendo (y validando) entre distintas alternativas para diseñar y mantener la *Pista de Despegue Arquitectural* (Architectural Runway): una cantidad de arquitectura suficiente para el mediano plazo, pero sin llegar al extremo de un diseño final completo e inflexible.
- **Business Owner** (*Responsable de Negocio*): Uno o más representantes (stakeholders) clave de las distintas partes interesadas, que tienen la responsabilidad principal (tanto técnica como de negocios) sobre la Solución que desarrolla ese ART en el marco de un Programa.

Además de esos roles también existen otros equipos o funciones que dan **soporte al Tren de Entrega**:

- **Equipo de Sistema**: Un equipo ágil especial, que da soporte a los otros para armar y usar los ambientes de desarrollo (con la infraestructura necesaria), las herramientas de Integración Continua y automatización (de testing o deploy), y también asiste con la integración entre equipos.
- **Servicios Compartidos**: Roles y servicios especializados necesarios para un Agile Release Train, pero sin dedicación de tiempo completo a un único Programa. Por ejemplo: *especialistas en seguridad informática*, *administradores de bases de datos*, *personal de operaciones*, *especialistas en documentación o traducciones*, etc.
- **Release Management** (*Gestión de Entregas*): Un equipo que se encarga de impulsar y aprobar versiones que se **implementarán** en Producción. En muchos casos esto lo gestionan directamente los equipos o los ARTs, pero algunas Soluciones deben cumplir con una gran

cantidad de normas y regulaciones que hacen necesaria una aprobación y gestión centralizadas.

Cuando el tamaño y/o la complejidad del producto o servicio requieren de **varios Trenes** (cada uno de ellos con 50-100 personas) trabajando en conjunto, SAFe® introduce el **nivel (opcional) de Solución Grande** (*Large Solution*): un conjunto integrado de ARTs (Tren de Solución), o sea un equipo de equipo de equipos, trabajando todos sobre una misma Solución.

Todos los equipos que participan de esta Solución comparten **la misma cadencia**: sus iteraciones a nivel equipo están sincronizadas, tienen **el mismo ciclo de Incremento de Programa** (8 a 12 semanas) entre los distintos Trenes, y comparten una misma demostración al final del ciclo, que en este caso se llama **Demo de Solución**.

Para administrar la complejidad extra que requiere coordinar el trabajo de cientos de personas (en una misma **Solución Grande**), SAFe® propone **tres roles adicionales**:

- **Solution Train Engineer** (*STE: Ingeniero del Tren de Solución*), equivalente al RTE pero a nivel Solución.
- **Solution Manager**, un rol análogo al del Product Manager pero para la Solución Grande (y que gestiona el Backlog de Solución).
- **Arquitecto de Solución**, el equivalente al Arquitecto del Sistema para este nivel.

Del mismo modo tanto el **Equipo del Sistema**, los **Servicios Compartidos** o el **Release Management** pueden tener equipos o personas destinadas específicamente al nivel de Solución, o coordinar entre las personas que dan soporte a los Trenes.

El **nivel más alto** de SAFe® (por encima de Equipo, Programa y Solución) es el de **Portfolio**. El objetivo de un Portfolio es **alinear la estrategia organizacional** con la planificación y ejecución de las tareas, organizando la empresa a través de una o más **Cadenas de Valor** (*Value Streams*). En general existe un sólo Portfolio, pero en empresas muy grandes puede haber varios (separados por ejemplo por distintas líneas de negocio), cada uno con sus respectivas Cadenas de Valor.

SAFe®: Ventajas y limitaciones

Ventajas:

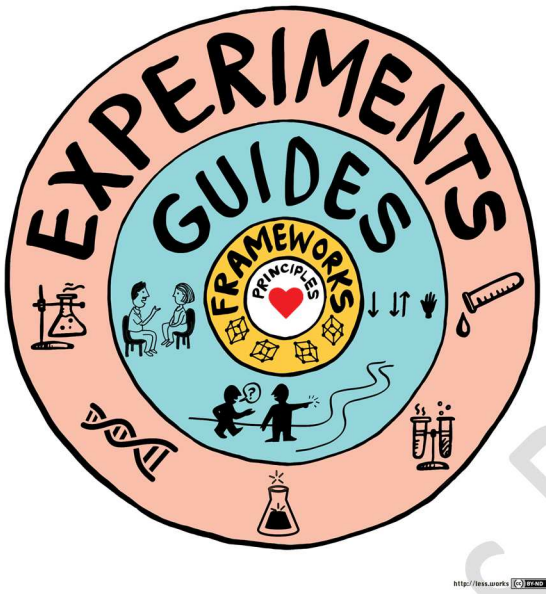
- Es un marco de trabajo **exhaustivo** y **muy completo**, que incluye e integra una gran recopilación de **buenas prácticas** ya existentes para el desarrollo de productos en forma ágil: Scrum, XP, TDD, Integración Continua, Lean UX, DevOps, etc.
- Es el framework con mayor cantidad de **documentación** "oficial", de **consultores** certificados, y de cursos de **entrenamiento** disponibles.
- Cubre algunos temas –de gran importancia para las organizaciones grandes– que otros métodos ignoran o prefieren evitar: por ejemplo **portfolios de programas**, **presupuestos y financiación**, gobernanza (**governance**) corporativa, **estrategia** organizacional. Y lo hace de un modo que las empresas grandes tradicionales pueden entender e implementar.

Desventajas (o Críticas):

- Su diseño y estructura son bastante **incompatibles con el espíritu de colaboración y auto-organización del agilismo**: propone un es**quema jerárquico de varios niveles**, donde los niveles superiores (arquitectos, managers, etc) toman la mayoría de las decisiones, y los niveles subalternos simplemente las ejecutan.
- Es un **esquema complicado**, donde los **procesos detallados**, los **consultores** "oficiales" y las **herramientas** específicas son lo más importante.

- Propone **ciclos de planificación y ejecución demasiado largos** (10-12 semanas) que incluso incorporan un "*sprint de estabilización*" final, una propuesta incompatible con los niveles de respuesta que necesita una organización realmente ágil.
- Como **no requiere de una transformación organizacional grande**, las empresas pueden implementarlo como esquema "prefabricado", sin cambiar su estructura de reporte verticalista ni su modelo de gestión tradicional (y por lo tanto, **sin obtener beneficios muy grandes**).

3. LeSS: Roles, artefactos, eventos y prácticas



El esquema de LeSS: Una serie de Principios esenciales de los que se desprenden las Reglas del framework, que pueden implementarse con las recomendaciones de sus Guías y probando los Experimentos sugeridos [*Nota: las imágenes incluidas no están traducidas al español porque la licencia de LeSS Company impide distribuirlas con modificaciones*].

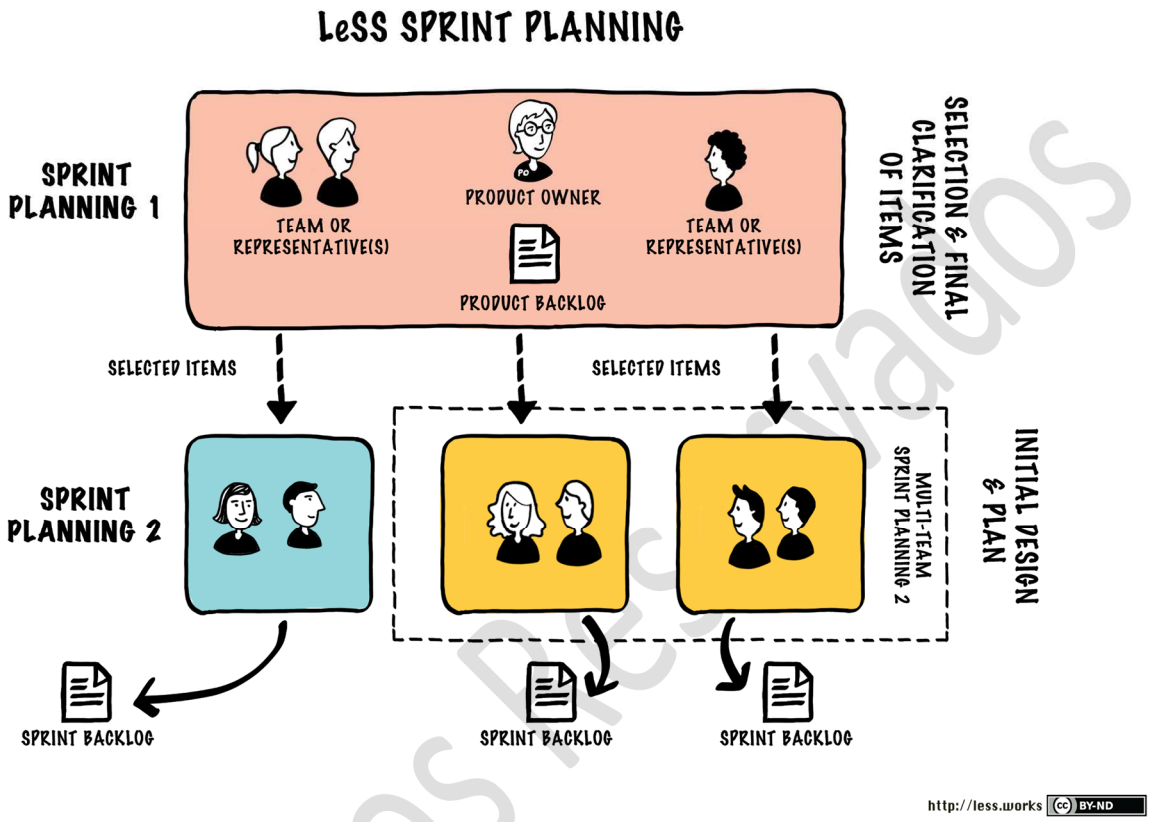
LeSS propone **Scrum**, **a varios equipos** de 3 a 9 personas (multi-funcionales, multi-componente, capaces de entregar valor al cliente c/u por sí mismo), **trabajando en conjunto** (con objetivos comunes) **sobre el mismo producto** de valor (una solución de cara al cliente, no un componente técnico o de infraestructura). Una descripción sorprendentemente similar a la de Scrum.

¿Qué propone LeSS para aplicar Scrum con varios equipos?

- ❏ Un **único product backlog** (porque el producto es el mismo).
- ❏ Una **única Definición de Hecho** (Definition of Done) para todos los equipos.
- ❏ Un **único incremento de producto** potencialmente implementable al final de cada sprint.
- ❏ Un **único Product Owner** (para todo el producto).
- ❏ **Varios** (de 2 a 8) **equipos** multi-funcionales, multi-componente, ubicado c/u en el mismo lugar físico, **todos orientados a funcionalidad de usuario**.
- ❏ Un **único sprint** para todos los equipos.

Al inicio de cada sprint se realiza una **Planificación** (**Sprint Planning**) al igual que en Scrum, con la única diferencia de que en LeSS se hace en **dos partes**: durante la **Sprint Planning 1** los equipos (completos, o algunos representantes) deciden junto al Product Owner cuáles son los **objetivos generales del sprint**, sobre qué temas (de mayor prioridad) va a trabajar cada equipo, y cuáles son las necesidades de **colaboración e integración** entre equipos para lograr esos objetivos.

Luego cada equipo hace su **Sprint Planning 2**, que es igual a la planificación de Scrum, pero en este caso en base a lo que acordaron los equipos en la parte 1. En los casos donde exista una necesidad fuerte de colaboración entre equipos para implementar una función determinada, la Planificación 2 debe incluir a esos equipos. Del resultado de la Planning 2 saldrá el **backlog del sprint** de cada equipo (igual que en Scrum).



La Planificación en LeSS: Sprint Planning 1 (con representantes de todos los equipos) para seleccionar objetivos generales, y después varias Sprint Planning 2 para generar el Sprint Backlog de cada equipo.

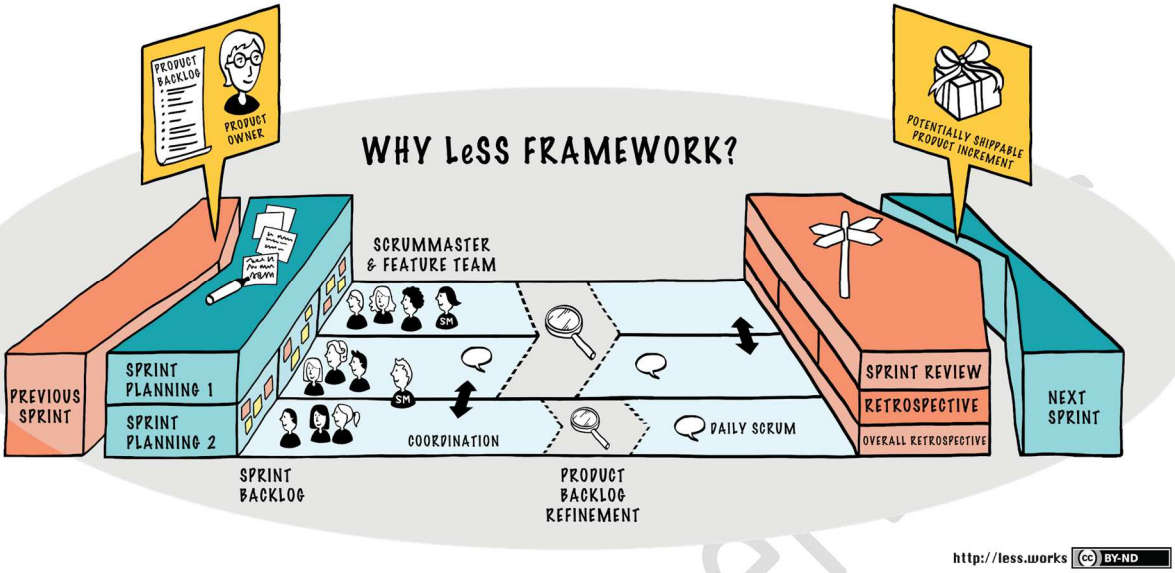
Luego comienza la **ejecución** propiamente dicha del sprint, que también es igual a la de Scrum (pero con el mismo calendario para todos los equipos), incluyendo la **sincronización diaria** (que es individual por equipo, aunque pueda visitar un observador de otro equipo si es necesario o útil). Puede existir también **Scrum de Scrums**, aunque está desaconsejado en LeSS como método de comunicación y coordinación entre los distintos equipos.

Una herramienta recomendada son las **Comunidades de Práctica** (*Communities of Practice*), grupos voluntarios compuestos por personas de distintos equipos, que se organizan **alrededor de un cierto componente** técnico o tipo de componente (ej: base de datos, motor de workflow) o de **un interés o especialidad funcional** (ej: arquitectura, requerimientos, diseño de experiencia de usuario) para **compartir conocimiento y experiencias**. Estas Comunidades promueven la comunicación frecuente, así como también la colaboración entre equipos sobre temas específicos.

El método más formal que propone LeSS para la comunicación y colaboración entre equipos son las **reuniones multi-equipo**: desde la *Planificación del Sprint* (que es compartida) hasta el *Refinamiento del Backlog* (que incluye personas de todos los equipos interesados en ese tema), incluyendo también **talleres de diseño** para que los equipos evolucionen colaborativamente la arquitectura en base a las necesidades.

Al final de cada sprint se realiza la **Revisión** (*Review* o *Demo*) al igual que en Scrum, donde los equipos demuestran colectivamente el incremento de producto (potencialmente implementable) que han logrado durante ese ciclo. Por eso es tan importante que todos los equipos compartan la misma **Definición de Hecho**: lo que se demuestra y revisa no es el trabajo de cada equipo sino el producto (único) de ese trabajo integrado.

Después cada equipo realiza su **Retrospectiva** para *reflexionar sobre su proceso de trabajo* y pensar posibles soluciones para los problemas o impedimentos (al igual que en Scrum). La diferencia es que después existe una **Retrospectiva Global**, donde todos los equipos juntos (o representantes de c/u) discuten los temas surgidos de las Retrospectivas de equipos que afecten a más de uno de ellos, o a todo el sistema.



El proceso de LeSS: varios equipos de funcionalidad trabajan sincronizadamente sobre el mismo sprint y mismo producto (con el mismo Product Owner), para generar en cada ciclo un incremento de producto.

De este modo cada equipo puede **entregar valor por sí solo al cliente final en cada ciclo**, sin esperar a que otro equipo complete otra fase, cumpliéndose así una de las premisas de Scrum.

En cuanto a **roles**, no existen más que en Scrum: Product Owner, Scrum Master, y miembros del equipo. Pero sí existen algunas diferencias en esos roles, necesarias para trabajar a nivel de varios equipos:

- El **Scrum Master**, al igual que en Scrum, es un coach y facilitador que ayuda al equipo a mejorar su funcionamiento (no un líder técnico ni un líder de proyecto). Pero a diferencia de Scrum, en LeSS debe estar mucho más atento al todo desde una **visión sistémica** (el sistema integrado que incluye a todos los equipos de producto) y no solamente a los equipos a los que presta servicio directamente.
- El **Product Owner**, al igual que en Scrum, administra y prioriza el backlog para maximizar el valor al cliente. Pero entre sus responsabilidades "*hacia adentro*" (ej: consultas y dudas del equipo) y las que son "*hacia afuera*" (ej: comunicación y validación con los usuarios) está

más **orientado "hacia afuera": la satisfacción de los usuarios y el retorno de inversión del producto**.

- ❑ LeSS puede usar un mismo Product Owner para varios equipos porque muchas de esas responsabilidades "hacia adentro" (clarificación de necesidades) las delega **empoderando al equipo para que se autogestione** en relación directa con los usuarios finales (con el Product Owner ejerciendo un rol de facilitador, no de intermediario).

LeSS Huge:

Cuando la coordinación necesaria implica más de 8 equipos, el esquema sugerido es distinto y se llama **LeSS Huge** (*LeSS Enorme*). Al igual que LeSS normal, existe **un solo Backlog** (general) de Producto, **una sola Definición de Hecho**, **un solo Product Owner** (general), **un solo sprint** simultáneo para todos los equipos, y **un solo incremento de producto** potencialmente implementable al final de cada sprint.

La estructura organizacional sugerida para LeSS Huge es muy parecida a la de LeSS, con algunas diferencias:

- ❑ **Departamento de No-Hecho** (*Undone Department*): grupos centralizados de funciones como *testing, arquitectura, homologación*, que idealmente no deberían existir por fuera de los equipos (ya que cada ciclo culmina con producto “Hecho” entregable), pero en esquemas muy grandes a veces resultan necesarios.
- ❑ **Soporte a los Equipos**: un grupo que centraliza algunas cuestiones de infraestructura como *gestión de configuraciones y de ambientes, herramientas de integración continua, herramientas de implementación* (pero lo centraliza como apoyo experto y soporte a los equipos, no como una imposición centralizada).
- ❑ **Entrenamiento y Coaching**: un grupo centralizado que provee entrenamiento y coaching sobre *prácticas técnicas y/o metodológicas*, en base a las necesidades de los equipos.

LeSS: Ventajas y limitaciones

Ventajas:

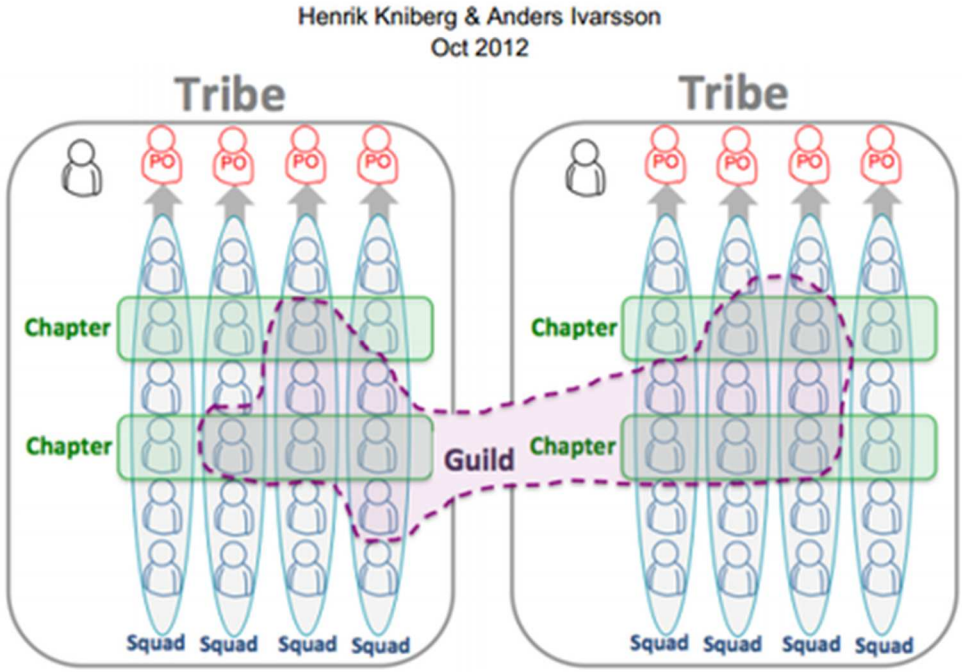
- Es probablemente el framework más **fiel al espíritu de Scrum y los valores del agilismo**, y por lo tanto el más apto para las organizaciones que quieran hacer transformaciones profundas o (re)diseñarse en forma radical.
- Es un **esquema bastante simple** (casi tan simple como Scrum) con muy pocos niveles, los mismos roles y eventos y artefactos, que no requiere "adaptación a medida" de procesos complejos por parte de consultores especializados.
- Prescribe y promueve **ciclos de entrega cortos**, donde en cada ciclo debería existir producto funcionando, con una orientación totalmente diseñada hacia la entrega de valor al cliente.

Desventajas (o Críticas):

- **No cubre determinados temas** (ej: *presupuestos, portfolios, programas de proyectos, integración con proveedores externos*, etc), y **no es tan exhaustivo** en sus prescripciones como otros marcos de trabajo más documentales o intensivos en proceso.
- Requiere de un **cambio organizacional muy profundo**, que muchas organizaciones probablemente no estén dispuestas a hacer, en especial las más grandes de gestión más tradicional.

"Todos los modelos están equivocados, pero algunos son bastante útiles" - George E.P. Box, Science and Statistics.

4. El modelo Spotify



Tribe:Tribu, Chapter:Consejo, Guild:Gremio, Squad:Brigada.

El modelo Spotify, en si no es un modelo como suele llamarse. Es una cultura de trabajo, que la han desarrollado con un contexto particular, y con estrategias prescriptivas y no prescriptivas. Para llegar al estado actual hicieron y hacen varios experimentos de prácticas y dinámicas. Algunos son exitosos, y otros no. Su cultura no es un estado definitivo, sigue evolucionando y cambiando como un organismo viviente. Debido a que es una cultura y no un modelo, son varios los que afirman que copiar su forma de trabajo puede ser contraproducente. Una comparación seria llevar de un día para otro la cultura de Estocolmo a Río de Janeiro, puede que no aplique, y menos en una implementación Big Bang. Los experimentos son exitosos en un contexto, en general no en todos.

Historia

En 2008 empezaron con un par de equipos Scrum. Luego en unos pocos años posterior se transformaron en varios equipos, y pusieron varias reglas de Scrum opcionales. Esto porque trascendieron a las prácticas y técnicas de scrum, llegando a otras de mayor valor para su nivel de expertise y contexto. Por ejemplo, la utilización de ciclos continuos, como Kanban y Lean.

"Ágil es de más valor que Scrum"

"Principios es de más valor que Prácticas"

Estas pautas, estos principios hizo que tomaran la opción de dejar fuera reglas de Scrum, cuando consideraban que lograban los principios ágiles mejor de otra forma.
Otra de sus situaciones desafiantes, fue crecer al doble de personas años tras año, su forma de trabajo soporto esto.

Misión

Si saber cuál es destino es fundamental para un viaje. Podemos decir que ese destino en la organización es su misión. Si hablamos de un sueño, muchas veces en el camino a concretar ese sueño, sabemos más de nosotros. Al saber más de nosotros modificamos ese sueño o lo cambiamos por otro de más satisfacción. También sucede esto con las organizaciones, al buscar la misión pueden aprender y definir otra misión de más valor.

Este es la misión de hoy de Spotify:

"Hacer Spotify el mejor lugar para descubrir música".

Piensan que no será la última misión.

Características:

- Cada equipo tiene su espacio físico para mejorar su interacción. Cara a Cara es la herramienta de comunicación más efectiva y eficiente.
- Equipos Autónomos
- Equipos Motivados
- Primero la misión y luego las sub-misiones. Primero el todo y luego el sector.
- Política cero.
- Ego cero.
- Comunicación por metas, anti-micromanagement.
- Copiar prácticas no por Estándar, por emular la Innovación. Copian lo mejor de otros equipos, porque funciona, no porque hay que hacerlo. Esto les da consistencia y Flexibilidad.
- Depende el framework, segun el equipo
- Colaboración en vez de Propiedad Privada.
- Para aumentar su velocidad, enseñan los responsables de una funcionalidad a que otros la sepan usar o modificar. Con esto reducen cuellos de botella.
- No forman silos
- No buscar hacer dueños del conocimiento, esparcen el conocimiento.

Equipos

El Ego es un enemigo a en una cultura de colaboración y la creación conjunta. Este es un principio que llevan a todos para marcarlo como una meta. Colaborar para encontrar la mejor solución, la base de esto es compartir, pedir ayuda y brindar ayuda. La diferencia con la competencia es que cada uno busca ser mejor que el otro, pero no necesariamente mejorar la organización. Esto puede llevar a una cultura de gente que oculte información, haga automarketing, como una forma de crecimiento individual, sin crecimiento de la organización, o sin aporte de valor.

No hay un jefe que te diga que hacer, se busca la mejor solución en equipo, o se puede tomar solo la decisión, dependiendo de la situación.

Foco en la motivación, el 94% dice que disfruta de trabajar en spotify. Entiende que las personas motivadas y contentas logran una muy superior contribución.

“Confianza es más importante que control”, esto aumenta la velocidad, la calidad y motivación.

Mentalidades incorporadas

Aprender del Error

- Explicitar que todos en algún momento vamos a fallar, nadie es infalible
- Foco en entender la experiencia para transformarla en un activo
- Fallar rápido y con bajo impacto, es la clave para aprender del error.

Comunidad

Comunidad más que en jerarquías. Esto para maximizar la interacción, aumentar la velocidad de respuesta, la adaptación al cambio, y la capacidad de ampliar capacidades en forma continua. Reduce la burocracia y la centralización de ideas. Para esto es bienvenida la informalidad.

Creatividad

“Si necesitas saber exactamente quien toma las decisiones, estas en el lugar equivocado.” Esto cultura no busca culpables, pretende hacer foco en la generación de ideas que surgen de inteligencia colectiva.

Lo difícil hacerlo muy seguido

Realizan pequeños releases para que se adquiera experiencia en el armado y se transforme en algo cotidiano y fácil. Pequeño y frecuente, rutina, aprendizaje y dominio.

Términos

Agile Coach: en vez de Scrum Master. (Otros practicas)

Brigada: en vez de Equipo Scrum. Equipo Cross auto organizado, entre 5 y 8 personas, puede no usar Scrum. Roles punta a punta de la cadena de producción, desde el diseño hasta el mantenimiento. No tienen un líder la brigada, son autoorganizados. Responden a un Dueño de Producto. La brigada tiene foco en "Que" y "Como", que entregar, validando con él "Para qué".

Tribu: son una agrupación de brigadas

Consejos: área de competencia, como testing, programación web o coaching. Hay un gerente por consejo responsable del desarrollo de los que están en el consejo. Un consejo está formado por gente que están en distintas brigadas de una sola tribu. Tiene el foco en el "Como", como hacerlo mejor.

Gremio: es una comunidad de intercambio entre tribus y consejos, que participan por un fin común, como Desarrollo Web, Entrega Continua, Diseño, etc. Similar al consejo, pero más global y sin gerente. Cualquiera que quiera participar se puede sumar.

Releases: despliegues a producción.

Trenes de Release, con funcionalidad en desarrollo, por cliente: Cada cliente tiene un ciclo de desarrollo de funcionalidad de 1, 2 o 3 semanas. Al finalizar el periodo se despliega todo lo desarrollado y lo que está en desarrollo a producción. La funcionalidad que está sin terminar es ocultada. Todo anteriormente fue probado y regresionado. Esta técnica adelanta problemas de integración y simplifica el versionado de código, evitando generar una nueva rama de versionado con lo que está sin terminar.

Tipos de brigadas

- **Funcionales**, mantiene un área funcional a través de las plataformas.
- **App Cliente**. Simplifica el despliegue en cada plataforma, son como los entrenadores técnicos las brigadas funcionales.
- **Infraestructura**: provee rutinas para entrega continua, Testing A/B, monitoreo y operaciones.

Foco de todos los tipos de brigadas en que la B. Funcional sea independiente. La B. Funcional hace sus propios despliegues, y monitorea su funcionalidad en producción.

Bibliografía utilizada y sugerida

https://www.scrumguides.org/. (2019). Retrieved from https://www.scrumguides.org/.

Institute, P. M. (2017). A Guide to the Project Management Body of Knowledge - PMBOK 6th. Project Management Institute.

Institute, P. M. (2017). Agile Practice Guide. Project Management Institute.

Institute, P. M. (2018). Organizational Project Management Maturity Model. Project Management Institute.

Nir, M. (2017). The Agile PMO. Amazon Digital Services.

Lo que vimos:

- Frameworks para Escalar Ágiles

Lo que viene:

- DevOps Transformation y SRE