**Filosofía Lean**

Es un enfoque más viejo que el agile, y surge en Japón en la industria automotriz (toyotismo). Dicha filosofía se basa en 7 principios; y bastantes años después se trajo dicha filosofía al software, en el libro Lean Software Development. El principal de todos los principios es el primero de ellos.

**7 principios Lean**

**1) Eliminar Desperdicios:** la idea detrás de este principio es tener un proceso eficiente, y que sume valor en todas sus partes, la intención es la mejora de procesos en forma continua. Por ende, si tenemos partes del proceso que no contribuyen a generar valor, son desperdicio, y dicho desperdicio debe ser eliminado. En general, todo aquello que no genera valor es desperdicio. El desperdicio también se puede entender como cualquier cosa que interfiere con darle al cliente lo que el valora, en tiempo y lugar donde le provea más valor.

Este principio está relacionado con el principio ágil de software funcionado y el de simplicidad (maximizar el trabajo no hecho). El 20% del software que entregamos debe contener el 80% del valor

Como dicho principio surgió en la industria automotriz, hay ciertas cosas que son desperdicio para la misma, como (Producción en exceso, stock, esperas, etc.), y dichas cosas que son desperdicio en la industria automotriz se deben trasladara la industria del software. Brevemente podemos decir que hay **7 desperdicios lean:**

* **Producción en exceso**: En industrias tangibles es fácil de interpretar, ya que implica mayores costos de almacenamiento, gasto innecesario de insumos, etc. A la hora de trasladarlo a industrias intangibles (software), esto se puede interpretar como **características extra**, es decir, desarrollar funcionalidades de más, o más funcionalidades de las que necesito actualmente, o también como especificación en exceso (upfront specification, perder tiempo especificando requerimientos que no son necesarios actualmente)
* **Stock**: en industrias tangibles el stock es costo, ya que hay fechas de vencimiento, desgaste por humedad, por calor, etc., y todo eso genera costos. En el software, el concepto de stock per se, no existe, sino que, se entiende como **trabajo a medias**, es decir, funcionalidades desarrolladas a medias, US no terminadas, todo esto es una manifestación del desperdicio. Otro ejemplo sería el acumular especificaciones de requerimientos que puedan ser susceptibles al cambio (desperdicio)
* **Proceso extra:** también entendido como pasos extra en un proceso, generan desperdicio. En la industria del software tenemos un problema relacionado a esto, y es que vamos de un extremo a otro, de nada de proceso a demasiado proceso. Por otra parte, los procesos muy estructurados obligan a hacer ciertas actividades que no agregan valor.
* **Movimiento**: El movimiento de productos en industrias tangibles genera un costo que es desperdicio, sin embargo, no tenemos un desperdicio análogo en la industria del software, aunque podemos decir que es similar al desperdicio producido por la espera, por ej.: compra de licencias (esperar a que las mismas sean entregadas), esperar la aprobación del cliente, demoras en la adquisición de recursos.
* **Defectos**: los defectos generan **retrabajo**, lo cual es un desperdicio, y esto es válido para cualquier industria. Para evitar el retrabajo se implementan otras actividades de aseguramiento de calidad.
* **Esperas**: la espera implica que necesito algo que me lo tiene que otorgar otra persona. En industrias tangibles se puede interpretar como que necesito que una tarea termine para poder arrancar otra, y la espera entre medio es desperdicio. En la industria del software esto se soluciona y se materializa mediante los equipos multidisciplinarios. Esto es debido a que, al saber todo lo necesario para realizar el trabajo, una persona puede, por ejemplo, implementar una US de principio a fin, y no tiene que esperar nada de nadie (aunque en la realidad esto no es tan cierto). En cambio, en los procesos definidos, las personas son silos funcionales, generando que existan esperas, y por lo tanto, desperdicios, por ejemplo: necesito que el DBA termine de armar la estructura de BD para poder arrancar a programar el Backend.
* **Cambio de tareas**: Por ejemplo, darle a otro equipo la responsabilidad de continuar con el desarrollo que estaba haciendo un equipo, esto es desperdicio. Darle a otra persona una US que estás desarrollando, eso es desperdicio. Esto se puede interpretar de esta manera, a que todo el tiempo que una persona tiene que usar, para entender lo que estabas desarrollando, y lo que falta por desarrollar, es desperdicio. El multitasking es desperdicio, ya que existe un tiempo de seteo entre poner la cabeza en una tarea intelectual y luego en otra, y dicho tiempo es desperdicio.

La búsqueda de información, si no hay SCM, corre el riesgo de gastar horas buscando artefactos, lo que me genera desperdicio. La SCM me permite eliminar estos desperdicios.

**2) Amplificar Aprendizaje:** Muchas veces los individuos tenemos mucho **conocimiento implícito**, que es nuestro, propio y no genera valor al equipo. Lo ideal es tener mucho **conocimiento explícito**, de todo el equipo, y esto se logra compartiendo nuestros conocimientos, es decir, haciendo explícito el conocimiento implícito.

Lo que busca este principio es crear y mantener una cultura de mejoramiento continuo y solución de problemas. Esto quiere decir, que el proceso debe hacer foco en crear conocimiento, y dicho conocimiento debe estar codificado de manera tal que sea accesible a todo el equipo u organización. Muchas veces los procesos definidos hacen difícil introducir en ellos mejoras. Por ejemplo, los procesos definidos definen el diseño completo de forma prematura, lo cual no permite que el diseño evolucione y esté en constante mejora, por lo que no se creará conocimiento.

La mejor forma de mejorar el proceso de desarrollo es amplificar el aprendizaje, aprender iterando, experimentado y obteniendo feedback.

**3) Embeber la integridad conceptual:** está relacionado con el principio ágil de “la calidad no se negocia”. No importa cuanto se enfoque el proceso en otras cosas, no se debe perder de vista las expectativas y necesidades del cliente, las cuales deben ser satisfechas. El objetivo es construir el producto con calidad desde el principio, no tratar de integrarla después.

Existen dos tipos de inspecciones, una que se realiza luego de que los defectos ocurren y otra para prevenir los defectos. Si uno quiere un producto de calidad, se debe inspeccionar antes no después. Si esto no es posible, se debe inspeccionar luego de pasos pequeños.

Consiste en encastrar todas las partes del producto o servicio, que tenga coherencia y consistencia. La integración entre las personas hace el producto más integro. Hay dos tipos de integridad:

* Integridad Percibida: el producto total tiene un balance entre función, uso, confiabilidad y economía que le gusta a la gente.
* Integridad Conceptual: todos los componentes del sistema trabajan en forma coherente en conjunto

El software requiere de un nivel adicional de integridad, y es que pueda mantener su utilidad en el tiempo. Se espera que el software evolucione mientras se adapta al futuro. Un software con integridad tiene una arquitectura coherente, tiene una alta usabilidad y adaptabilidad a su propósito, y debe ser mantenible, adaptable y extensible.

**4) Diferir Compromisos:** Consiste en postergar las decisiones hasta el último momento responsable (momento en el cual todavía estamos a tiempo). Es decir, postergar las decisiones hasta que tengamos que hacerlas, posterguemos las decisiones y evitemos la upfront specification. La idea es postergar hasta obtener más información para tomar la decisión, ya que si nos anticipamos tenemos información parcial o incompleta.

Se relaciona con el principio ágil de decidir lo más tarde posible pero responsablemente, no hacer trabajo que no va a ser utilizado. Se enlaza con el principio anterior de aprendizaje continuo, mientras más tarde decidimos más conocimiento tenemos.

Esto no significa que todas las decisiones deben postergarse, y se debe tratar de tomar decisiones reversibles, de forma tal que pueda ser fácilmente modificable. Las mejores estrategias de diseño de software están basadas en dejar abiertas opciones de forma tal que las decisiones irreversibles se tomen lo más tarde posible.

**5) Dar poder al equipo:** Implica que los equipos tengan la capacidad de tomar decisiones. Está relacionado con los equipos autogestionados de agile. La idea es no tratar a las personas de un equipo como si no tuvieran capacidades intelectuales, sino que les demos la libertad para tomar decisiones y que hagan su trabajo. Es decir, debemos respetar a la gente, fomentar una buena ética laboral, y delegar decisiones y responsabilidades del producto en desarrollo al nivel más bajo posible.

Las prácticas de Lean implican técnicas tipo pull, y no push (explicado en kanban)

**6) Ver el todo:** Tener una visión holística, de conjunto del producto a construir y lo que nos falta o es necesario para construir el producto. Hace foco en algo más grande que solo lo que yo tengo que hacer. Hay que lograr la visualización del todo. Un problema que puede surgir si falla este principio, es plantear objetivos individuales sin considerar los globales. Es decir, a veces el bien común es perjudicado si las personas atienden primero sus intereses particulares.

**7) Entregar lo antes posible:** Implica satisfacer al cliente con entregas frecuentes de software funcionando. Hay que diferenciar entre:

* Entregar rápido: Estabilizar ambientes de trabajo a su capacidad más eficiente y acotar los ciclos de desarrollo.
* Entregar rápidamente: eso hace que se vayan transformando n veces en cada iteración, mediante incrementos pequeños de valor. Llegar al producto mínimo que sea valioso y salir pronto al mercado