INNOVA UNTREF
Revista Argentina de Ciencia y Tecnología

ICyTEC
Instituto de Ciencia y Tecnología
UNTREF

HACIA OTRO MODELO DE PROCESO DE DESARROLLO DE SOFTWARE

TOWARDS ANOTHER SOFTWARE DEVELOPMENT PROCESS MODEL

Fontela, Carlos[1]
Paez, Nicolás[2]

Fontela, C. y Paez, N. (2022). Hacia otro modelo de proceso de desarrollo de software. *Revista INNOVA, Revista argentina de Ciencia y Tecnología, 10.*

RESUMEN

Existe la noción de que en la industria del software lo que cambia es la tecnología: hardware, dispositivos, software de base, lenguajes y herramientas. Sin embargo, también vienen cambiando, desde hace décadas, los paradigmas, la organización de los equipos de trabajo, los roles en los mismos, la manera de entregar el producto y la forma en que se gestiona el desarrollo. En este artículo exploramos cómo se ha ido abandonando el modelo de desarrollo propio de las industrias manufactureras, utilizado en la industria del software durante mucho tiempo. En ese modelo, un producto se concibe, se construye y llega a los clientes a través de una serie de pasos que se realizan en forma sucesiva, en los cuales trabajan roles diferentes, incluso con intereses contrapuestos. Hoy se ha pasado a un modelo en que las distintas actividades se desarrollan de manera simultánea, con equipos plurifuncionales, mientras que el producto se entrega de manera incremental y continua, con los clientes y usuarios trabajando codo a codo con los desarrolladores a lo largo de todo el proceso. También analizamos por qué ocurre esto, qué impacto tiene en la forma de trabajo y presentamos evidencia de que esta evolución implica una mejora.

ABSTRACT

There is a notion that in the software industry what changes is the technology: hardware, devices, system software, languages and tools. However, for decades, the paradigms, the

[1] Laboratorio de Métodos de Desarrollo y Mantenimiento de Software, Universidad de Buenos Aires, Argentina – Universidad Nacional de Tres de Febrero, Argentina / cfontela@fi.uba.ar / ORCID 0000-0002-0286-9100
[2]Universidad de Buenos Aires, Argentina – Universidad Nacional de Tres de Febrero, Argentina / nicopaez@ieee.org / ORCID 0000-0002-0453-4259

organization of work teams, their roles, the way of delivering the product and the way in which development is managed have also been changing. In this article we explore how the development model of manufacturing industries, used in the software industry for a long time, has been abandoned. In that model, a product is conceived, built and reaches customers through a series of successive steps, in which different roles work, even with conflicting interests. Today we have moved to a model in which the different activities are carried out simultaneously, with cross-functional teams, while the product is delivered incrementally and continuously, with customers and users working side by side with developers throughout the process. We also analyze why this is happening, what impact it has on the way of working and we present evidence that this evolution implies an improvement.

PALABRAS CLAVE

Ingeniería de Software/ modelos de procesos de desarrollo/ evolución/ DevOps/ entrega continua.

KEY WORDS

Software Engineering/ development process models/ evolution/ DevOps/ continuous delivery.

INTRODUCCIÓN

- **Ingeniería y desarrollo de software**

La Ingeniería de Software, en su acepción más tradicional, es la disciplina que estudia el desarrollo, mantenimiento y operación del software, sus técnicas, herramientas y metodología (IEEE, 1990). Una definición más moderna dice que "es la aplicación de un enfoque empírico y científico para encontrar soluciones eficientes y económicas a problemas prácticos de software" (Farley, 2022)

La noción de una "ingeniería" del software surge a fines de la década de 1960 en respuesta a lo que en ese entonces se denominaba la "crisis del software"(Naur, 1968), cuando el desarrollo de software era visto solamente como sinónimo de una práctica mecánica de codificación. Uno de los supuestos básicos en la génesis de esta disciplina era que los costos crecientes del desarrollo y el mantenimiento del software se debían a la falta de procesos sistemáticos como los de las ingenierías. Por eso, una de las principales preocupaciones de ese momento fue definir formalmente procesos de desarrollo y mantenimiento.

En el marco de este artículo nos quedamos con la acepción más general del desarrollo. Por ello, llamamos desarrollo de software a todas las actividades que van desde el contacto inicial con un cliente que necesita una solución de software o desde una necesidad percibida de negocio que debe ser abordada desde el software, hasta que el producto está instalado en un ambiente productivo disponible para el usuario (en otras palabras, el producto está agregando valor a partir de solucionar el problema para el cual fue concebido). Dicho de manera más sucinta, a los fines de este artículo, incluimos como parte del desarrollo a todas las actividades necesarias desde la concepción del producto hasta su despliegue en los equipos en los que se va a ejecutar. Adquisición y gestión de requisitos, diseño, programación, gestión de equipos y personas, pruebas, despliegue, etc., son todas disciplinas que englobamos dentro del desarrollo de software.

- **El cambio en el proceso de desarrollo**

Los métodos y procesos de desarrollo de software están en constante cambio. Esto se debe, no sólo al avance tecnológico, sino también a que los procesos de negocio que soporta el software lo demandan.

Además, como el software está cada vez más presente en la economía, en los artefactos que usamos, en los usos del día a día, cada vez son más los procesos de negocio que demandan software, y cada vez sus necesidades son más perentorias, con cualidades crecientes de seguridad, usabilidad, confiabilidad, resiliencia y otros atributos de calidad (Forsgren et al 2018; Kelly, 2017; Andreessen, 2011).

En definitiva, no es el avance de la "tecnología" lo que hace cambiante a esta disciplina, sino la propia evolución de los métodos de construcción, que por su frecuencia de cambio la hacen única entre las ingenierías.

En este artículo analizaremos cuáles son los últimos avances en cuanto al proceso de desarrollo de software, y explicaremos qué hace que ese proceso esté en cambio constante, qué implica a nivel de equipos y personas, de gestión del desarrollo, de arquitectura, y mostraremos la evidencia existente de que realmente implica una mejora sobre modelos anteriores.

OBJETIVOS

Este artículo tiene como objetivos:

- Establecer las razones por las cuales el proceso de desarrollo de software continúa evolucionando.
- Explicar las principales características del modelo de proceso que se está imponiendo en la última década y su impacto en las diferentes disciplinas de la Ingeniería de Software.
- Dar a conocer la evidencia empírica que existe sobre las ventajas que el modelo actual ofrece sobre los enfoques anteriores.
- Dejar en claro que el nuevo modelo de proceso, si bien destaca favorablemente en la mayor parte de los escenarios, sigue coexistiendo con modelos anteriores en ciertos tipos de software.

BREVE RESEÑA DE ENFOQUES METODOLÓGICOS

- **Familias de métodos en el tiempo**

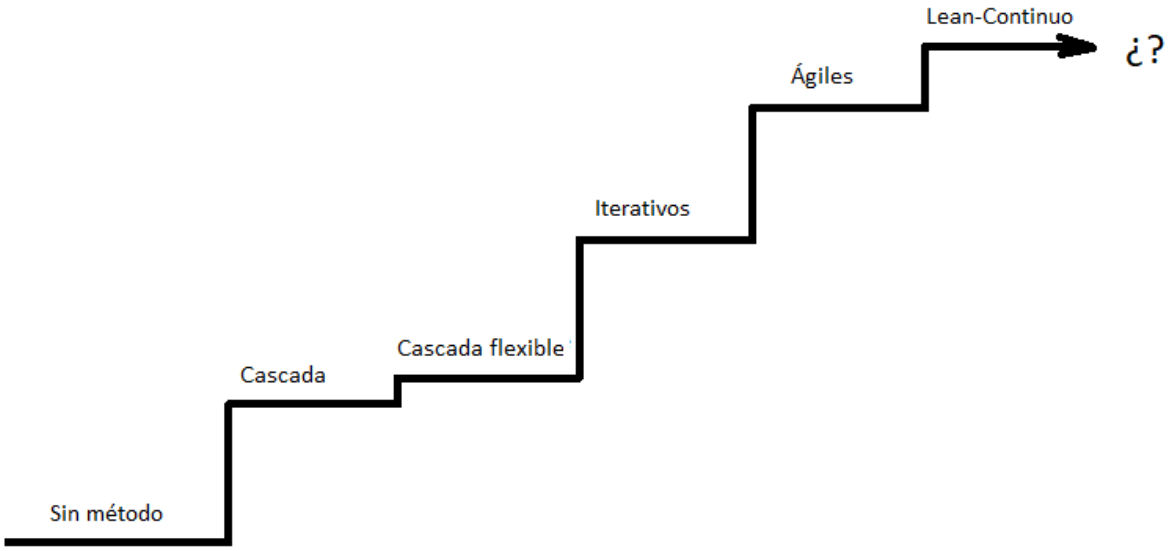
Muy groseramente, las familias de métodos de desarrollo que han prevalecido en las distintas épocas son los que se muestran en la Tabla I.

Tabla I: familias de métodos y épocas de predominio

Familia de métodos o procesos	Otra manera de referirnos	Época en que fue el método más corriente
Sin metodología	Sólo programación y pruebas al azar	Hasta mediados de los 1960
Cascada (Royce, 1987)	Desarrollo basado en etapas-actividades disjuntas	1960s - 1970s
Cascada flexibilizada	Cascada con vuelta atrás	1970s - 1990s
Primeros métodos iterativos	Iterativos basados en planes	~1995 - ~2005
Ágiles de primera generación	Iterativos - incrementales basados en el manifiesto ágil	~2000 - …
De flujo continuo	Lean-Continuo (ver luego)	2014 - …

No todos los cambios fueron igualmente importantes. La Fig. 1 muestra una evolución de los modelos según el autor.

Fig. 1 Evolución de los modelos de desarrollo de software (fuente: elaboración propia)



En efecto, los programas que se escribían hasta mediados de la década de 1960 eran todavía relativamente sencillos, y el costo de escribir programas era insignificante respecto del costo de las máquinas sobre las que esos programas se ejecutaban. Por eso el desarrollo de software consistía en escribir programas y probarlos de manera artesanal.

En la década de 1960 se produjo lo que en aquel momento se denominó la "crisis del software": los costos de programar subieron, los viejos programas requerían mantenimiento, y éste era costoso, y se empezó a percibir que, así como las ingenierías tenían procesos de fabricación de productos, el software también debería tenerlos. En ese momento se copió de alguna manera el modelo de cadena de montaje, que tan buenos resultados daba en la industria manufacturera: para construir un producto había que ir cumpliendo una serie de pasos sucesivos, cada uno ejecutado por trabajadores que asumían un determinado rol. De esa copia del modelo de cadena de montaje, surgió lo que luego se denominó modelo de cascada, que básicamente es un proceso de desarrollo que asume que un producto de software debe ser construido siguiendo una serie de etapas, cada una de las cuales implica uno o más roles y, al terminar la última de las etapas, se llega al producto final. Esas etapas, además, implicaban disciplinas diferentes: tanto es así que el nombre que se dio a esas etapas se correspondía con el nombre del trabajo predominante en la etapa o del resultado de la misma: requerimientos y análisis, diseño, construcción, pruebas, implantación o despliegue, etc. Como ya dijimos, se supone que cada etapa tiene uno o más especialistas: analistas, diseñadores, programadores, testers, especialistas en configuraciones e instalaciones de software. También se supone que las etapas no admiten vuelta atrás: por ejemplo, una vez que pasamos la etapa de diseño y están los programadores codificando el producto, no se vuelve más sobre el diseño. Esto es lo que dio a este modelo la denominación de "cascada". La única excepción a esta regla ocurría con la situación en la que, si durante las pruebas se encontraban problemas, había que volver sobre la construcción, aunque en algunos casos se solucionaba el problema conceptual llamándola etapa de "pruebas y depuración", entendiendo que "depuración" (o "debugging", que es el término original en inglés) era la actividad por la que se corregían los errores de construcción.

Al poco tiempo este modelo fue modificado para poder volver sobre etapas anteriores si lo que fallaba en las pruebas lo ameritaba. En efecto, si del resultado de las pruebas se infería que había un problema, no de construcción, sino de diseño, había que regresar, al menos con parte del producto, a la fase de diseño. Lo mismo ocurría con el análisis, y así sucesivamente. Pero todo se reducía a volver atrás con una parte y repetir la "cascada" desde allí. Incluso el mantenimiento, que era un problema cada vez más presente, se resolvía con un nuevo proyecto por cada cambio, que también se encaraba con la filosofía de cascada.

Los años ochenta, que tantos avances trajeron en los lenguajes y paradigmas de programación, no impactaron demasiado en lo metodológico.

En los noventa empezaron a surgir procesos de desarrollo iterativos, entre los que destaca el Proceso Unificado. La filosofía de los procesos iterativos de primera generación sostenía que el software se debía construir por iteraciones, en las cuales había un poco de cada actividad. Así, las etapas se convirtieron en iteraciones, pero ahora ya no atadas a una actividad en cada una, sino que en una misma iteración podía haber algo de requisitos, algo de diseño, algo de construcción, algo de pruebas, y hasta podía ser que en algunas de ellas, sobre todo una vez que el proyecto estaba encaminado, podía haber uno o más despliegues del producto. Hay algo más que quedó en evidencia con los métodos iterativos: el desarrollo de software permite el cambio de alcance durante los mismos proyectos, sobre todo porque a quien ve software funcionando se les disparan nuevas ideas para mejorarlo.

Pero comenzar a hablar de iteratividad, menor resistencia al cambio y entregas antes de terminar el proyecto, fue como abrir la caja de Pandora: al poco tiempo nacieron los métodos iterativos-incrementales, y detrás de ellos, los primeros métodos llamados "ágiles".

La filosofía del agilismo, resumida en pocas palabras, es que el desarrollo de software es más un problema de diseño que de fabricación (Farley, 2022), y que requiere menos papeleo, menos rigidez metodológica, más atención a las personas, a entregar valor a los clientes en incrementos pequeños de producto y más apertura a cambios (Beck, 2001). Los años 2000 fueron los años del agilismo, al punto que al día de hoy son pocos los que lo cuestionan y menos aún los que ponen en duda las premisas del manifiesto. Muchos colegas afirman que ya no tiene sentido hablar de métodos ágiles, porque ya están establecidos como la manera en que se construye el software. Como los métodos ágiles propugnan la entrega frecuente de producto, casi todos ellos plantean que a la salida de cada iteración, que se asume de unas pocas semanas, hay que estar en condiciones de entregar un incremento de producto.

Pero sin dejar de lado los principios ágiles, hace unos diez años comenzamos a dar otra vuelta de tuerca. Se empezó a cuestionar que el software se deba construir con un modelo de proyectos, que asume un principio y un fin. Así nació el ciclo de vida de flujo continuo que, contrariamente al modelo iterativo-incremental, no requiere la existencia de iteraciones para la entrega de producto, sino que el mismo debería estar en condiciones de ser entregado en cualquier momento. Así, pasamos de iteraciones de meses a semanas, de semanas a una semana y luego a nada, con el *timebox* discreto degenerando en un ciclo continuo (Henney, 2021). Las iteraciones siguen existiendo, y son el marco en el cual se realiza la planificación, las revisiones al proceso, algunas reuniones internas y con los clientes, pero ya no guardan la relación unívoca con las entregas que existía en los métodos ágiles de primera generación. Es decir, el nuevo modelo no requiere terminar una iteración para entregar producto, sino que constituye una línea de producción continua que empieza tomando una característica para

trabajar sobre ella, en la medida que el equipo tiene capacidad disponible, y termina con un incremento de producto en condiciones de ser desplegado en producción.

Por supuesto, el resumen anterior no quiere decir que en cada época sólo se usó una familia de métodos. El desarrollo en cascada ha seguido usándose luego de la aparición de los métodos iterativos y ágiles, aunque cada vez en menor medida, hasta su cuasi extinción hace unos diez años. Del mismo modo, los métodos iterativos-incrementales se siguen usando hoy en día, e incluso el modelo de flujo continuo mantiene la mayor parte de las características de los métodos ágiles de primera generación.

- **Disciplinas del desarrollo de software en el tiempo**

Los cambios en los procesos han impactado también en las disciplinas del desarrollo de software.

El cuadro de la Tabla II, muy esquemático y un tanto arbitrario, nos muestra tres épocas y cómo se fueron modificando las disciplinas de desarrollo.

Tabla II: épocas y enfoque de las disciplinas de desarrollo

Actividad	**En los primeros modelos**	**Una situación intermedia**	**Modelo actual**
Gestión del desarrollo	Inexistente	Modelo de proyectos (PMBOK, 2021)	Modelo de producto (Narayan, 2018)
Decisión sobre qué producto construir (concepción del producto)	Listas de "requisitos", Ingeniería de requisitos (Gane, 1977; Loucopoulos, y Karakostas, 1995)	Product Owner trabajando codo a codo con el equipo (Schwaber y Beedle, 2002)	Usuarios especifican las características del sistema en base a ejemplos en conjunto con el equipo (Adzic, 2009)
Mapeo de "requisitos" al diseño	Diseño estructurado (Stevens et al., 1974 y Yourdon, 1989)	Diseño orientado a objetos (Booch, 1990)	Diseño guiado por el dominio (Evans & Evans, 2004)
Diseño y arquitectura	Sistemas unitarios monolíticos	Arquitecturas en capas (Fowler, 2003), diseño hexagonal (Cockburn, 2005)	Microservicios (Fowler & Lewis, 2014)

Actividad	En los primeros modelos	Una situación intermedia	Modelo actual
Control de calidad	Como etapa final del proyecto	Como parte de cada iteración (Schwaber y Beedle, 2002)	Pruebas automatizadas integradas al flujo de trabajo, monitoreo y experimentos en producción (Humble y Farley, 2010)
Despliegue	Proceso complejo que se ejecuta a intervalos grandes, luego de pruebas exhaustivas de integración y sistema	Potencialmente al final de cada iteración (Schwaber y Beedle, 2002)	Despliegue muy frecuente en la nube
Equipos de trabajo	Roles por actividad/etapa	Equipos plurifuncionales (desarrollador = programador+tester) (Schwaber y Beedle, 2002)	DevOps y DevSecOps (Dyck et al, 2015; Myrbakken y Colomo-Palacios, 2017).

En apartados posteriores analizaremos algunas de estas cuestiones desde el punto de vista del modelo que vamos a presentar.

RAZONES DEL CAMBIO

Se ha analizado hasta el hartazgo por qué el producto software está sometido a tanta demanda de cambio (Myrbakken y Colomo-Palacios, 2017; Lehman, 1978). Sin embargo, no hay tantos estudios que expliquen por qué cambian también los métodos que se utilizan para desarrollarlo.

Veamos algunas cuestiones que pueden explicar este fenómeno.

- **El software en el centro del negocio**

Quedan cada vez menos negocios exitosos que no tengan al software como columna vertebral.

El conocido artículo de Marc Andreessen, de 2011, ya lo anunciaba al decir que “el software se está comiendo al mundo” (Andreessen, 2011). Otra forma de ver lo mismo, más centrada aún, dice que “toda compañía es una compañía de software” (Kelly, 2017). Tal vez sea exagerado si lo planteamos para la totalidad de la economía, pero al menos es algo que aplica para las compañías exitosas, que dependen del software para entregar productos y servicios, al punto de que sin el software no podrían competir.

Hay compañías que sólo son el producto de software: Booking no es una empresa hotelera, Uber y Cabify no tienen autos ni emplean choferes, Spotify no produce música. Otras compañías sí proveen algunos bienes y servicios, pero el software es su activo más

importante: Netflix, Tesla, Amazon, empresas de medios gráficos, etc. (Charette, 2021). Y las demás organizaciones exitosas, aunque su producto principal sea otro, dependen del software para poder mantenerse competitivas: empresas industriales, financieras y bancos, venta mayorista y minorista, telecomunicaciones, etc. Incluso los gobiernos y las organizaciones sin fines de lucro usan crecientemente el software para brindar servicios de calidad a sus beneficiarios. En estos casos, está claro que el desarrollo de software no es algo que se pueda considerar como un servicio externo, una especie de caja negra que resuelve problemas, o un centro de costos de la compañía cuyo negocio pasa por otro lado.

En este escenario, además, ya no funciona el desarrollo de nuevos productos y servicios mediante un modelo de proyectos a largo plazo. Por eso es que, cada vez más, estas organizaciones están conformadas por equipos pequeños que trabajan en ciclos cortos de desarrollo y entrega, a la vez que miden continuamente el feedback de los usuarios, para lograr o mantener un alto desempeño.

En pocas palabras, estas organizaciones, para seguir siendo competitivas, necesitan acelerar el proceso de entrega de bienes y servicios, mejorar el proceso de detección de errores y entender las nuevas demandas de clientes y usuarios, además de anticiparse a los cambios regulatorios que impacten a sus sistemas. En el centro de todas estas necesidades, está el software como motor de aceleración de las mismas.

- **El software debe poder cambiar cada vez más rápido**

El software puede cambiar, para adaptarse, para corregir problemas, para mejorar, y también para extenderlo. Estas propiedades, llamadas comúnmente *maleabilidad* y *extensibilidad*, son propias del software, y las personas y organizaciones que consumen software lo saben.

Debido a estas características, se demandan cada vez más cambios (y extensiones) a los productos exitosos de software. Ya decía Fred Brooks hace varias décadas que todo software exitoso va a ser cambiado (Brooks & Bullet, 1987). También Kent Beck planteaba, hace veinte años, que había que aceptar el cambio y acompañar el proceso con iteraciones cortas y entregas frecuentes, ajustando constantemente el plan, como quien ajusta la dirección de un vehículo (Beck & Fowler, 2001). Y esto que era cierto hace 40 y 20 años, se ha convertido en una presión creciente por obtener esos cambios y extensiones cada vez con mayor frecuencia, con menores costos y en menores tiempos.

A veces se debe a que esos cambios se necesitan urgentemente por razones regulatorias extremas o para no perder contra la competencia, pero lo cierto es que la presión por el cambio existe. Por eso, en la última década, las compañías basadas en software han pasado de liberar sus productos en forma trimestral o mensual a hacerlo por semanas, días e incluso horas.

El problema con el cambio no es que no se pueda hacer, como los consumidores de software descubrieron hace mucho tiempo. El problema es que realizar los cambios es difícil y costoso. Por eso desde siempre se han introducido nuevos paradigmas y nuevos lenguajes que supuestamente han venido facilitando el cambio. Y también por eso, que es lo que nos ocupa a nosotros, es que siempre estamos buscando nuevas maneras (métodos, procesos) de desarrollar software.

Por lo tanto, el proceso de desarrollo debe ir adaptándose para:

- Atender las demandas crecientes de cambios rápidos y seguros.
- Entregar los cambios con mayor frecuencia.
- Que el uso de los productos dé feedback sobre el éxito o fracaso de los cambios introducidos.

Dicho de otro modo, necesitamos procesos que permitan entregas frecuentes y de calidad, que hagan que los equipos de desarrollo se enfoquen en los resultados, que los pedidos de cambios puedan atenderse en cualquier momento y responderse enseguida, y tener mecanismos que permitan conocer cómo se está usando el producto minuto a minuto.

Mientras esto ocurra, vamos a seguir necesitando adaptaciones a los procesos de desarrollo de software.

En primer lugar, antes de considerar cualquier proceso, el propio software, por su naturaleza, nos facilita las cosas. Al ser un producto maleable y extensible, es posible cambiarlo sin necesidad de empezar de cero, particionarlo para permitir entregas incrementales, extenderlo y experimentar, a un menor costo que otros productos industriales.

Otra característica, que en general se considera que nos juega en contra, la intangibilidad del producto, refuerza el hecho de que la construcción incremental sea fundamental para poder recibir feedback a tiempo sobre el alcance, la calidad, etc.

Algunas prácticas, como la interacción colaborativa, la autoorganización y el potenciamiento de las habilidades sociales que se proponen para los equipos de trabajo, está fuertemente soportada y realimentada por la noción de que el software es materialización de conocimiento humano.

- **Los nuevos enfoques funcionan**

Por supuesto, no tendría sentido adaptar métodos y procesos de desarrollo si luego no van a funcionar. Pero, como veremos más adelante, algunos de estos nuevos procesos funcionan, y lo hacen muy bien.

En este artículo vamos a analizar un enfoque de desarrollo que algunos llaman Entrega Continua (CD), otros DevOps y otros de otras maneras, pero que está resultando exitoso desde hace algunos años.

UN ENFOQUE MODERNO Y CON LÓGICA INTERNA

- **El proceso en pocas palabras**

El modelo de proceso que vamos a analizar es una combinación de prácticas provenientes de distintos métodos marco que ha demostrado funcionar bien en el desarrollo moderno de software, más allá de su adopción total por la industria. Sus principales características son:

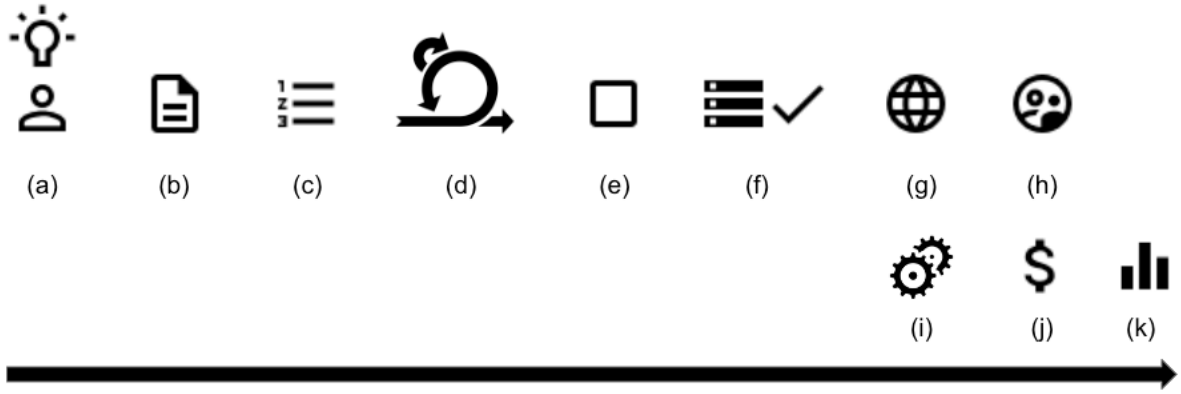
- Trabajar y entregar pequeños conjuntos de características cada vez, con valor para los clientes. Cada característica parte de un *backlog* (o lista de características) que se va elaborando entre los interesados y el equipo de trabajo (probablemente con la mediación de un *Product Owner*, en la terminología de Scrum), va avanzando por una línea de producción (o *pipeline*), que implica diferentes actividades como construcción, pruebas, integración en un ambiente pre-productivo, etc., hasta quedar en condiciones de ser desplegado en el ambiente productivo.

- El sistema, con los últimos cambios implementados en la rama principal del repositorio, debe estar en condiciones de ser desplegado en cualquier momento. En algunos casos, se puede usar la técnica de despliegue continuo, que implicaría que cada pequeño cambio se sube inmediatamente en producción, pero esto no es obligatorio en nuestro enfoque ni tampoco es tan habitual en la industria.
- Automatizar todo lo que sea factible automatizar. Esto implica las pruebas a distintos niveles, los *builds*, los despliegues en distintos ambientes y las configuraciones de ambientes (que deben ser tratadas como código).
- Cada equipo, a la vez que debe ser pequeño, es responsable de manera íntegra por el resultado de su trabajo. Esto hace que los equipos deban ser autogestionados y plurifuncionales. Típicamente, no hay contraposición de intereses entre programadores, testers, gente de negocio (*Product Owners* en la terminología habitual), personal de operaciones y especialistas en seguridad.
- Todos los equipos que trabajan en un producto deben colaborar entre sí en función de la entrega de valor. Sin embargo, los equipos trabajan de manera independiente, sin necesitar del permiso de nadie externo para hacer un cambio de implementación o un despliegue. Esto implica que no haya procesos complejos de aprobación de cambios o solicitudes de integración de cambios, privilegiando las prácticas del tipo de revisiones por pares, la programación en parejas de XP y Mob Programming (Beck, 2000; Zuill & Meadows, 2016).
- La cultura organizacional es típicamente colaborativa y orientada a los resultados. Es lo que Ron Westrum llama una cultura generativa, enfocada en la misión y en la cual la información fluye, los riesgos se comparten y hay transparencia en todos los niveles de la organización (Westrum, 2004)
- Trabajar con la práctica de integración continua. Al menos una vez por día se debe integrar el producto en la rama principal de desarrollo, haciendo un *build* y ejecutando pruebas a distintos niveles (incluso las de aceptación). Al mismo tiempo, el entorno en el que se despliega debería ser idéntico al de producción. Esto facilita las entregas frecuentes sin demoras ni largos procesos de integración.
- La seguridad se incorpora al producto desde etapas tempranas, y se testea en forma continua, con revisiones de seguridad en todas las entregas más o menos importantes al menos.
- Obtener feedback frecuente de los usuarios del sistema mediante el monitoreo en producción. Este monitoreo, en lugar de medir cuestiones técnicas, como habitualmente se ha hecho, debe enfocarse en reflejar la calidad del servicio desde la perspectiva del cliente.
- En cuanto a las prácticas de gestión, se siguen las de Lean Management (Forsgren et al, 2018). y las de Lean Startup (Reis, 2011). Para más detalles, ver más abajo.

La Fig. 2 muestra un esquema de las distintas actividades de desarrollo que quedan englobadas en el ciclo continuo. Todo comienza con una idea o necesidad del usuario (a), a partir de la cual definimos una visión que sienta las bases de la iniciativa (b) y generamos una lista de características del producto, o *backlog* en términos de Scrum (c). Luego, comenzamos a trabajar en forma iterativa (d), y vamos generando incrementos de producto (e), los cuales verificamos en un ambiente simil producción (f). Si está todo acorde a las expectativas, desplegamos a un ambiente productivo (g) donde lo ve nuestra comunidad de usuarios (h).

Una vez que el sistema está en producción debemos operarlo (i) para asegurar que estamos entregando valor (j) y finalmente obtenemos métricas (k) para luego volver a empezar.

Fig. 2 Ciclo de vida del producto de software (fuente: elaboración propia)



Debemos aclarar algo importante antes de seguir.

Lo que estamos presentando es un modelo, y como todo modelo implica una meta planteada como un escenario ideal. Si bien estas prácticas han demostrado su validez y su impacto en la mejora en el desempeño de las organizaciones basadas en software, no todas se aplican siempre en conjunto, e incluso hay algunas cuyo grado de aceptación en la industria aún no es completo. Por ejemplo, la programación en parejas y la integración continua, cuya efectividad y desempeño ha sido analizados a fondo y demostrados hace mucho (Williams et al, 2000; Cockburn & Williams, 2000) no han logrado todavía una adopción significativa: las empresas y las personas siguen prefiriendo programar en solitario y recurrir a ramas de desarrollo que se fusionan en la rama principal de desarrollo mediante solicitudes de integración de cambios (*merge-request* o *pull-request*, como las llaman las herramientas). Pero los resultados de las investigaciones llevadas a cabo, que muestran su adecuación y efectividad, hacen que las enumeremos como parte de nuestro modelo, con el cual guardan una relación coherente y lógica, como veremos más adelante en este artículo.

- **El dilema del nombre**

El enfoque que vamos a analizar, con pequeñas variaciones, es conocido con diversos nombres, según donde se ponga el acento. Veamos algunos:

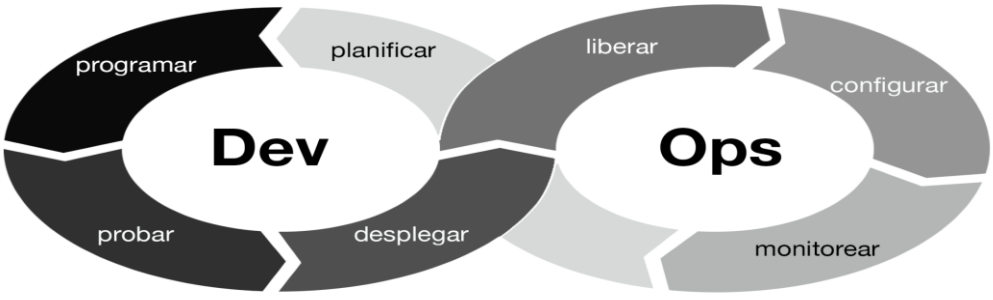
- *DevOps*: pone el acento en que los desarrolladores (*devs*) y la gente de operaciones (*ops*) son parte del mismo equipo, enfocados en la entrega de valor al cliente. Este es el nombre que utilizan, entre otros, Forsgren, Kim y Walls (Forsgren et al., 2018; Walls, 2013).
- *DevSecOps* (a veces *SecDevOps*): agrega al anterior la necesidad de que los especialistas en seguridad también sean parte del mismo equipo. Este nombre se utiliza mayormente para aprovechar el éxito del término DevOps e incorporar prácticas de seguridad que otros autores consideran parte de DevOps (Lehman, 1978).

- *Entrega Continua* (*CD*, en inglés, "*Continuous Delivery*"): pone el acento en que el proceso asegure que el producto esté en condiciones de ser entregado en cualquier momento. Este es el nombre que utilizaba Jez Humble luego de la publicación de su libro (Humble y Farley, 2010).
- *Digital continuo* (en inglés, "*Continuous Digital*"): pone el acento en la naturaleza continua del proceso y en el hecho de que el software está en el centro de los modelos de negocio de nuestro tiempo. Este es el nombre que utiliza Allan Kelly (2017)
- *Lean Software Development*: toma el nombre de un proceso industrial (*Lean Manufacturing*) que pone el foco en hacer más con menos recursos y menos desperdicio, entregando lo que el cliente desea en el menor tiempo posible, a la vez que se obtiene feedback todo el tiempo, incluso desde el sistema en producción. Este es el nombre que utilizaron Mary y Tom Poppendieck para un proceso un poco menos prescriptivo que el que estamos presentando aquí (Poppendieck & Poppendieck, 2003).
- *Lean-Kanban*: pone el foco en el apego a la filosofía Lean, agregándole el nombre de un tablero que se usa para registrar el avance de cada característica y controlar el trabajo en progreso. Es el nombre que utiliza Henrik Kniberg en algunos de sus trabajos (Kniberg, 2011).

No todo el mundo está de acuerdo con que estos conceptos se refieran a lo mismo. La mayor parte de la literatura que trata de DevOps considera a la entrega continua como una capacidad de DevOps. Algo parecido ocurre con Lean Software Development, que se usa desde antes que DevOps y CD (Poppendieck & Poppendieck, 2003).

DevOps se ha convertido en el nombre más popular. Lo que ocurre es que DevOps, que ha empezado como una idea de poner a desarrolladores y gente de operaciones a trabajar en pos de un mismo objetivo, se ha convertido en un nombre para casi cualquier cosa. Un artículo de 2015 define DevOps de una manera bastante acotada y, a juicio de los autores, acertada: "DevOps es un enfoque organizacional que hace hincapié en la empatía y la colaboración interfuncional dentro y entre equipos - especialmente desarrollo y operaciones de TI - en organizaciones de desarrollo de software, para operar sistemas resilientes y acelerar la entrega de cambios." (Dyck et al, 2015). El libro de Bass et al., si bien amplía lo que DevOps había sido en sus comienzos, no es tan abarcativo: "DevOps es un conjunto de prácticas cuya intención es reducir el tiempo entre el *commit* de un cambio en un sistema y el momento en que el cambio es puesto en producción" (Bass et al, 2015). La Fig. 3 muestra el ciclo de DevOps.

Fig. 3 Ciclo de DevOps (fuente: elaboración propia)



Notemos algo de la relación entre DevOps y CD: con las primeras definiciones, DevOps es más bien un corolario de CD, que precisa de la colaboración entre roles. Sin embargo, hay autores más recientes para los cuales la entrega continua es simplemente una de las características de DevOps, que la engloba.

Pero, como decimos más arriba, hoy DevOps se usa como una palabra comodín sin un significado claro. Por ejemplo, hay "ingenieros DevOps", que cuesta distinguir de los antiguos "ingenieros de despliegue" de hace una década, cuya principal actividad es la de escribir scripts para automatizar tareas (Kerzazi y Adams, 2016). Los textos que tienen DevOps como parte de su título cada vez incluyen más prácticas dentro de la supuesta disciplina. Hay incluso productos comerciales que llevan DevOps como parte de su nombre.

Nuestro modelo no coincide con exactitud con ninguno de los procesos definidos arriba, sino que es una combinación basada en lo que ha dado mejores resultados en pos de la entrega eficiente de software (que permita entregas frecuentes y de calidad, enfocadas en los resultados, que facilite encarar los pedidos de cambio de manera ágil y que permita tener feedback sobre el uso del producto) y está avalado por investigaciones, tanto en la industria como en la academia. Tal vez se adapte bien a lo que Dave Farley llama "Ingeniería de Software Moderna" (Farley, 2022). A falta de un nombre único, y para facilitar el análisis sin sesgar hacia ninguno de los que se utilizan, en este artículo, cuando sea necesario, vamos a denominarlo *"Lean-Continuo"*, que enfatiza que se trata de un enfoque basado en principios Lean, que a la vez soporta un proceso de producción y entrega continuos.

- **Trabajos previos**

El proceso que estamos describiendo se basa en muchas de las premisas de los métodos ágiles. Por ejemplo, la integración continua es parte de XP, tal como lo definió Beck en sus primeras publicaciones (Beck, 2000); sin embargo, se planteaba que el producto debía ser integrado en el ambiente de desarrollo, no listo para ser desplegado por demanda. Las entregas frecuentes también aparecen entre las primeras ideas de XP (Beck & Fowler, 2001); pero no se planteaba que hubiera entregas antes de finalizar una iteración, aunque éstas se recomendaba que fueran cortas. La noción de que programadores, testers y otras personas que trabajan en la construcción del producto debían trabajar, no enfrentados, sino con los mismos objetivos, se puede rastrear sin problema en el equipo de desarrollo de las primeras versiones de Scrum (Schwaber y Beedle, 2002); sin embargo, los especialistas en operaciones y en seguridad quedaban por fuera del proceso, o al menos no se los mencionaba explícitamente. La automatización de pruebas a distintos niveles (unitarias y "de clientes") también se menciona en XP (Beck, 2000); pero no hay una pretensión de automatización de la línea de producción entera. Y así podríamos seguir.

Por eso, mientras los métodos ágiles existen desde los primeros años del siglo, muchas de las prácticas que mencionamos aquí debieron esperar una década más:

- El uso de los principios Lean para el desarrollo de software, con su énfasis en lograr feedback rápido desde los clientes a lo largo de todo el ciclo de vida, fue presentado por Mary Poppendieck y Tom Poppendieck en un libro en 2003 (Poppendieck y Poppendieck, 2003)
- En 2009 Patrick Dubois, un practicante del agilismo, organizó una conferencia en Gante, Bélgica, que denominó DevOps Days, introduciendo el término y la idea de que

los distintos roles asociados a la producción de software debían trabajar en pos de un objetivo común.
- En 2010, Jez Humble y David Farley popularizaron CD con la publicación de un libro que haría historia al ampliar el horizonte de la integración continua (Humble y Farley, 2010).
- El término “despliegue continuo” se popularizó con el paper "Climbing the Stairway to Heaven" (Holmstrom-Olsson et al, 2012).

Luego de estos primeros trabajos seminales, hubo muchos trabajos científicos y libros que se publicaron. Sólo por nombrar los que han tenido mayor impacto:

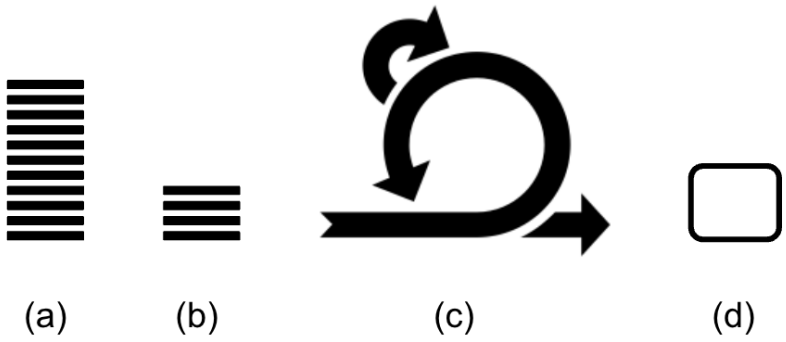
- Henrik Kniberg publicó un libro en el que explica un caso práctico del uso de Lean y Kanban en un proyecto de gran escala en Suecia (Kniberg, 2011)
- Len Bass y otros dos autores publicaron un libro en el cual exploran las decisiones que deben hacer los arquitectos de software que deseen utilizar DevOps y cómo su trabajo resulta impactado por otros interesados (Bass et al, 2015)
- Varios autores escribieron “The DevOps Handbook”, un libro conteniendo varios casos de estudio de la aplicación de DevOps en diversas industrias (Kim et al, 2016)
- Nicole Forsgren, Jez Humble y Gene Kim publicaron su libro “Accelerate”, que luego veremos que es un extenso informe de un proyecto de investigación sobre capacidades de organizaciones de alto desempeño (Forsgren et al, 2018)

- **Impacto en los modelos de gestión**

El enfoque de Lean-Continuo ha sido disruptivo respecto de anteriores modelos de gestión.

En efecto, si bien toma elementos del agilismo, y es compatible prácticamente con todos los principios del Manifiesto Ágil, plantea algunas diferencias que no resultan del todo compatibles con algunos de los métodos ágiles de primera generación. Por ejemplo, en las primeras versiones de Scrum se plantea un trabajo en iteraciones en las cuales no se pueden agregar nuevas funcionalidades una vez comenzadas ni tampoco se prevén entregas en cualquier momento que no sea el fin de una iteración. Asimismo, muchos indicadores típicos de los primeros métodos ágiles, como los *burn charts* y la medición de velocidad, no están pensados para una lógica de entrega continua, mientras otros, como los puntos de historia, son una medida de tamaño y no de valor. La Fig. 4 muestra el ciclo típico de las primeras versiones de Scrum y su entrega por iteraciones: el backlog de producto contiene toda la lista de funcionalidades del proyecto por construir (a), en cada iteración el equipo trabaja sobre una porción de funcionalidades que se denomina backlog de sprint (b), el trabajo se desarrolla en iteraciones de tiempo fijo con reuniones diarias de sincronización (c), al final de cada iteración tenemos un incremento de producto potencialmente entregable (d). No obstante, han habido muchos cambios a los métodos ágiles, al punto que se los ha ido adaptando para hacerlos compatibles con lo que llamamos el modelo Lean-Continuo (Schwaber & Sutherland, 2021).

Fig. 4 Ciclo de Scrum (fuente: elaboración propia)



En lo que sigue nos centraremos en las novedades que plantea nuestro modelo. Las prácticas de Lean Management (Forsgren, 2018; Reis, 2011). y de Lean Startup (Reis, 2011) han demostrado llevarse bien con varios de los supuestos del proceso. Por eso, nuestro modelo de gestión de basa en las mismas:

- Limitar el trabajo en ejecución (o *WIP*, por su acrónimo en inglés)
- Gestión visual
- Feedback desde producción
- Gestión simple y liviana de cambios
- Trabajo en pequeños lotes de características
- Visibilizar el flujo de trabajo
- Recolectar el feedback de los clientes e implementar lo que se obtenga de allí
- Experimentar continuamente

Todo el proceso de desarrollo tiene que gestionarse alrededor de la idea de valor, que es el parámetro a maximizar. Si bien este enunciado parece sencillo, no es tan fácil definir qué significa "valor", tanto para las empresas que buscan maximizar beneficios como para las organizaciones sin fines de lucro.

Por ello, una cuestión importante es la de los indicadores de gestión del producto. Forsgren, Humble y Kim plantean en su libro cuatro indicadores de salud del proceso de desarrollo, que indirectamente miden productividad, calidad, estabilidad y rendimiento, tanto del proceso como del equipo (Forsgren, 2018). Los indicadores propuestos son *Tiempo de espera* (en inglés "*Lead Time*"), desde que se inicia el proceso de desarrollo de una característica hasta que está en condiciones de ser entregada al cliente; *Frecuencia de despliegue*, en un ambiente de producción, *Porcentaje de fallas en puestas en producción*, *Tiempo medio para restaurar el servicio luego de una falla*.

- **Gestionando en base a criterios de aceptación**

Más arriba mencionamos que en modelos más antiguos se había trabajado bastante en la cuestión de cómo definir qué funcionalidades y características debía tener el sistema a construir, llegando a constituir una disciplina de "ingeniería de requisitos". El problema con este enfoque es múltiple, pero tal vez lo más cuestionable sea que supone que lo que

necesitan los clientes está allí, en forma de "requisitos", y no es algo que debamos ayudar a encontrar. Además, todas las técnicas que se usaban suponían que podíamos acceder a los supuestos clientes y usuarios. Los métodos ágiles mejoraron mucho esta cuestión, con las historias de usuario de XP, mejor aún si estaban acompañadas por criterios y casos de aceptación, con el *product backlog* de Scrum, y otros artefactos que remedian algunos de los problemas.

Cuando usamos Lean-Continuo, las herramientas que nos brindan los métodos ágiles están bastante cerca de la filosofía del método. Sin embargo, hay cosas que se pueden hacer para mejorarlas.

Cuando Kent Beck presentó XP hizo especial hincapié en especificar mediante historias de usuario acompañadas por pruebas de cliente (Beck, 2000). Está bastante claro en sus primeras publicaciones que Beck se refiere a pruebas de aceptación. Pero con el tiempo, la práctica fue derivando en lo que se conoció como *TDD (Test Driven Development)*, basado en pruebas unitarias que sirven como especificación del comportamiento de pequeñas porciones de código (Beck, 2003). Esta transición ha sido desafortunada, ya que – si bien es una buena práctica guiar el diseño mediante pruebas unitarias – ha minimizado la importancia de las pruebas de aceptación para derivar el comportamiento del sistema, más allá de que el propio Beck ha recomendado regresar a los orígenes, desarrollando en base a pruebas a varios niveles (Beck, 2011).

Con el tiempo, fueron surgiendo algunas prácticas que buscaron subsanar este problema. Bastante tempranamente, Bret Pettichord y Brian Marick comenzaron a usar la expresión *Agile Acceptance Testing* para implementar las pruebas de cliente de las que hablaba Beck (Pettichord y Marick, 2022). Luego, Dan North presentó *BDD (Behavior Driven Development)* como una práctica para hacer bien TDD y fue convergiendo a una manera de especificar el comportamiento esperado mediante escenarios concretos que se puedan automatizar como pruebas de aceptación (North, 2006). Mugridge acuñó el acrónimo *STDD (Storytest Driven Development)* (Mugridge, 2008). Otros hablaron de *ATDD (Acceptance Test Driven Development)* (Koskela, 2007; Gärtner, 2012). Y finalmente llegaría *SBE (Specification By Example)*, que fue presentada originalmente como una práctica para mejorar la comunicación entre los distintos roles de un proyecto de desarrollo, en especial entre los desarrolladores y los clientes y usuarios, y luego ha ido convirtiéndose en una práctica colaborativa de construcción basada en especificaciones mediante ejemplos que sirven como pruebas de aceptación (Adzic, 2009; Adzic, 2011). Si bien las diferencias entre los distintos enfoques son sutiles, y ya se han analizado en un trabajo de hace una década (Fontela, 2012), vamos a quedarnos con la idea de SBE, más interesante para este trabajo.

En definitiva, se plantea que hay que partir de ejemplos que acompañan a las historias de usuario, y que sirven tanto como especificaciones del usuario, guías para el desarrollo y casos de aceptación a probar. De esta manera, se va construyendo el sistema de manera incremental en base a historias de usuario que tienen valor para el cliente.

El éxito de las pruebas de aceptación es el que nos permite conocer mejor cuándo se ha satisfecho un requisito, tanto para el desarrollador como para el cliente: simplemente hay que preguntar si todas las pruebas de aceptación de la historia de usuario están funcionando. Adicionalmente, da una visibilidad mayor de los avances parciales para ambos.

SBE trabaja mediante un enfoque de afuera hacia adentro, partiendo de funcionalidades y sus casos de aceptación (que son ejemplos de uso futuro del producto) para luego ir refinando el diseño (Freeman y Pryce, 2009).

En definitiva, los ejemplos de SBE presentan ciertas ventajas, de las cuales las más importantes son:

- Sirven como herramienta de comunicación.
- Son más sencillos de acordar con los clientes, al ser más concretos.
- Evitan que se escriban ejemplos distintos, una y otra vez por roles distintos, con su potencial divergencia.
- Sirven como pruebas de aceptación: más aún, las pruebas de aceptación son la especificación.

Si bien SBE se ha planteado más como una práctica colaborativa de construcción de conocimiento, sin demasiado énfasis en la automatización, en nuestro contexto, el uso de SBE implica automatizar todas las pruebas que se generen.

- **Un lenguaje ubicuo**

Una de las recomendaciones básicas de SBE es que las especificaciones se construyan en talleres multidisciplinarios, de los cuales participan todos los interesados, con especial énfasis en los roles de usuario, analista de negocio, desarrollador y tester (Pugh, 2010).

Otro de los objetivos de los talleres de especificaciones que plantea SBE es establecer un lenguaje común más cercano al del negocio. Ese lenguaje es el que usan los participantes del taller, luego los testers al escribir los casos de aceptación y los programadores en su código. En ese sentido, es análogo al lenguaje ubicuo que plantea *DDD (Domain Driven Design)(Evans y Evans, 2004)*. Evans sugiere que el éxito de un proyecto está fuertemente relacionado con la calidad del modelo de dominio, que a su vez está soportado por un lenguaje ubicuo que se usa en ese dominio. De esta manera, el lenguaje de los interesados es el que se utiliza durante todo el desarrollo, desde los requerimientos hasta las pruebas, pasando por los nombres de clases, métodos, funciones, etc., de modo tal que se convierte en un proceso de construcción de conocimiento (Park, 2011).

Ahora bien, ¿qué es este lenguaje ubicuo?

Se trata de una serie de conceptos clave que definen el dominio y el diseño. El dominio, para evitar las confusiones en el uso de términos similares que significan cosas distintas. Respecto del diseño, estos conceptos, por un lado definen una jerga del proceso de desarrollo, y por el otro van a servir para dar nombres a las clases, métodos, módulos, o cualquier artefacto del diseño.

En realidad, el lenguaje ubicuo lo define el propio equipo de desarrollo ampliado (esto es, incluyendo a clientes, usuarios y expertos del dominio): no es necesariamente el lenguaje del negocio, ni un estándar de la industria, ni la jerga que usan los expertos, sino un conjunto de términos, con sus propiedades y significados, que utiliza este equipo amplio. Ello se debe a que es el lenguaje colaborativo que usa el equipo para capturar los conceptos y términos de un dominio y lograr una comprensión compartida. Por eso, no es ubicuo para todo un proyecto ni una organización, sino solamente dentro del equipo y dentro de un contexto.

- **Controlando (y monitoreando) la calidad del producto**

Como dijimos más arriba, el proceso que llamamos Lean-Continuo implica automatizar todas las pruebas que puedan automatizarse, en todos los niveles: unitarias, técnicas de integración, de comportamiento, de aceptación de usuarios e incluso, si se puede, también pruebas de seguridad y de desempeño, dejando afuera solamente a las pruebas exploratorias y de usabilidad, y algunas otras que requieran intervención humana. El automatizar las pruebas, además de que hace repetible la ejecución de las mismas a un costo mínimo, favorece también las pruebas de regresión y la corrección de errores en producción.

Por una cuestión de costo de las pruebas a distintos niveles, podemos usar la pirámide de automatización de pruebas como guía (Cohn, 2010). Como las buenas prácticas indican, las pruebas unitarias deberían ser las más abundantes, seguidas de las de integración, luego las de comportamiento a través de una capa de servicios y sólo finalmente las pruebas de aceptación a través de la interfaz de usuario.

Además de las pruebas, que se utilizan como control de calidad dentro de la línea de producción de Lean-Continuo, hay otros controles de calidad que pueden hacerse sobre el sistema en vivo: son las que llamamos actividades de monitoreo.

Una posibilidad que se suele implementar son las *pruebas A/B* o los patrones *Blue/Green Deployment* (Fowler, 2010) o *Canary Release* (Sato, 2014) para analizar la aceptación en producción de la nueva implementación. Estas técnicas, de una manera u otra, permiten desplegar a un subconjunto de nuestros usuarios y probar la aceptación en producción antes de desplegarlo a todos ellos.

Finalmente, existen técnicas para analizar la resiliencia del sistema en producción, siendo la más popular la de *Chaos Engineering*, que consiste en experimentar de manera controlada si el sistema responde de la manera esperada ante fallas insertadas intencionalmente (Rosenthal y Jones, 2020). De paso, Chaos Engineering sirve también para recalibrar los modelos mentales que tenemos de los sistemas (Jones, 2018).

- **Impacto en la arquitectura**

La arquitectura del sistema también va a ser afectada por el uso del proceso que hemos llamado Lean-Continuo. Para entender por qué, recordemos que necesitamos facilitar los despliegues independientes, que los equipos puedan trabajar autónomamente y decidir cómo realizar la construcción del producto, que se puedan escribir pruebas automáticas de la manera más desacoplada posible, entre otras cosas.

Todo lo anterior nos lleva a que un requisito fundamental de nuestra arquitectura es que se base en componentes cohesivos con el menor acoplamiento posible, y que permita que esos componentes se desplieguen independientemente. Una posible arquitectura que soporte esto es microservicios, que separa la implementación de componentes detrás de interfaces flexibles y bien definidas, alentando el bajo acoplamiento y la reusabilidad (Richards, 2015 y Newman, 2021).

La arquitectura de microservicios es un enfoque para desarrollar una aplicación como un conjunto de componentes, cada uno corriendo en su propio proceso y comunicándose mediante mecanismos livianos (Fowler y Lewis, 2014) Estos servicios son construidos en correspondencia con capacidades de negocio y pueden desplegarse independientemente mediante procesos totalmente automatizados. A la vez, pueden ser escritos en diferentes lenguajes de programación y usar distintas tecnologías de almacenamiento de datos. Cada

microservicio puede verse como un componente cohesivo y autónomo trabajando en conjunto con otros microservicios para entregar valor.

La idea de los microservicios se ha visto popularizada por el auge de las aplicaciones en la nube y sus necesidades (escalar fácilmente, desplegar a una alta frecuencia, etc.).

No obstante las ventajas de los microservicios, tengamos en cuenta que no es la única arquitectura posible: lo que en definitiva necesitamos es cualquier arquitectura desacoplada para permitir bajar el ancho de banda de las comunicaciones entre equipos y permitir despliegues frecuentes y autónomos. Un monolito modularizado con las mejores prácticas de bajo acoplamiento y fuerte cohesión podría ser suficiente en muchos casos.

Además, microservicios no es una arquitectura exenta de dificultades. Cualquier arquitectura distribuida es más compleja de desarrollar y probar que una monolítica. La depuración (*debugging*) de una aplicación con microservicios, al no poder tener todo junto el ambiente, el estado del sistema y el flujo de ejecución, es mucho más compleja que la que se hace sobre un sistema monolítico. Algo parecido ocurre con las refactorizaciones que deban cruzar límites de servicios. La seguridad misma se ve afectada por la heterogeneidad de plataformas y los múltiples puntos de ataque. Y desde ya el desempeño es un inconveniente de cualquier sistema distribuido. Por todo lo anterior, hay quienes sostienen que no debe adoptarse para sistemas u organizaciones pequeñas (Hansson, 2016). Un conocido artículo de Fowler recomienda introducir microservicios después de que el sistema ya ha evolucionado y está suficientemente maduro (Fowler, 2015).

Una posibilidad que no exploraremos, pero que va en la misma línea es la de *micro-frontends* (Jackson, 2019) que aboga por los mismos principios de los microservicios aplicados a la perspectiva del usuario que ve a la IU como una unidad conceptual.

- **Propuestas para los equipos de trabajo**

Los equipos ágiles siempre fueron plurifuncionales. Ya las primeras versiones de Scrum proponían que todas aquellas personas necesarias para desarrollar un incremento de funcionalidad fueran un único equipo de *desarrolladores*, enfocados en la entrega de valor en cada iteración (Schwaber y Beedle, 2002). Pero en el escenario actual es aún más importante: ya no se pueden hacer entregas continuas a alta frecuencia si tenemos límites funcionales que fuerzan esperas entre distintas áreas que se controlan entre sí en forma sucesiva, con el viejo principio administrativo de “control cruzado por contraposición de intereses”. Así, lo que era inadecuado hace veinte años es aún más inadecuado hoy. Precisamente lo que falla es la propia “contraposición de intereses” cuando lo que se alienta es la colaboración.

En Lean-Continuo, cada equipo tiene que trabajar en un mismo producto o unidad de negocio de punta a punta, empezando por definir lo que hay que construir y terminar con cada característica puesta en producción. Eso implica que no haya especializaciones, sino equipos realmente multidisciplinarios, que incluyen programadores, testers, analistas de negocio, gestión, especialistas en experiencia de usuario y hasta operaciones y seguridad. Eso disminuye las comunicaciones entre equipos y permite a cada uno ser más autosuficiente. Por eso, se propone DevOps como modelo de equipo. Y esos equipos asignados a productos o unidades de negocio permanecen durante todo el ciclo de vida del producto, y no asociados a proyectos con un comienzo y un fin (más sobre esto en el próximo ítem).

Un planteo interesante en este sentido es el de las organizaciones podulares (Gray, 2013). En las mismas, el control de la organización está distribuido en equipos pequeños (*pods*) que

funcionan de manera autogestionada, habilitados y empoderados para tomar decisiones de manera autónoma. Estos equipos/pods experimentan y se adaptan rápidamente. Su responsabilidad se mide por los resultados, sin importar cómo se organicen internamente.

Hay que aclarar un punto con este tema de los equipos plurifuncionales. Es importante destacar que se requiere que el equipo sea plurifuncional, no cada uno de los miembros (Cox et al, 2019). En efecto, el desarrollador *full-stack*, que tanto se ha puesto de moda en las descripciones de puestos y las búsquedas laborales, es difícil de encontrar y difícil de mantener actualizado, debido a la proliferación de herramientas y tecnologías. Es más realista pensar en equipos plurifuncionales formados por personas con funciones diferentes en el mismo equipo.

Podría pensarse que hay una tensión entre la idea de equipos con múltiples especialistas y la idea de que los equipos sean pequeños. Sin embargo, hoy existen varias capacidades que se ofrecen como servicio en la nube con la provisión de infraestructuras virtualizadas que se ocupan de ruteo, balanceo de carga, servidores de aplicaciones, almacenamiento, etc. Podría, sí, existir un equipo de plataforma que se encargue de proveer estos servicios a los demás equipos plurifuncionales en forma centralizada (Cox et al, 2019).

En cuanto al balanceo de trabajo dentro de los equipos, la mejor propuesta es tener profesionales especializados en una o dos funciones o áreas, pero que sean capaces de ayudar a otros roles cuando se requiera (por ejemplo, un programador que oficia de tester cuando hay mucho para probar, en la medida que no pruebe lo mismo que él programó; o un programador de interfaces de usuario que pueda ayudar programando objetos de negocio cuando sea necesario). También ayuda rotar roles en forma periódica y compartir conocimiento dentro del equipo.

Para determinar la estructura de la organización en su conjunto, hay quien ha previsto utilizar lo que se ha dado llamar en la *maniobra inversa de la Ley de Conway*. En efecto, Conway postuló, en 1968, que "las organizaciones que diseñan sistemas (...) producen diseños que son copias de las estructuras de comunicación de estas organizaciones" (Conway, 1968). Ruth Malan fue más allá y afirmó, unos 40 años más tarde: "si la arquitectura del sistema y la arquitectura de la organización están en desacuerdo, gana la arquitectura de la organización" (Malan, 2008). La ley de Conway cuenta con abundante evidencia empírica en su favor.

Fue James Lewis, de Thoughtworks, en 2015, quien al querer pasar a una arquitectura de microservicios decidió aplicar lo que él denominó "la maniobra inversa a la ley de Conway": armar la estructura de la organización y organizar los equipos para que la arquitectura salga como se desea (Fowler y Lewis, 2017; Parsons, 2015).

Como derivación de las ideas previas, Matthew Skelton y Manuel Pais presentan un enfoque que llaman *Team-First Thinking* (Skelton y Pais, 2019). Partiendo de la premisa válida de que la organización de los equipos es un factor clave del éxito (De Marco y Lister, 2013) y de la validez de la maniobra inversa a la Ley de Conway, plantean comenzar a definir la arquitectura a partir de la definición de la estructura de la organización que la produce.

Entre los temas que plantean Skelton y Pais está el de limitar las comunicaciones entre equipos que realizan tareas operativas (no así en tareas experimentales y de descubrimiento, como las de los talleres de SBE, que sí necesitan comunicación entre equipos) y definir estrictamente las interfaces entre ellos. Dando un paso más, plantean que cada parte del sistema tiene que tener como dueño sólo a un equipo, mientras que los demás equipos no

deben hacer cambios en esa parte, sino a lo sumo enviar *pull requests* o sugerencias, o a lo sumo formar parejas de ambos equipos en casos extraordinarios.

En el mismo sentido, abordan la cuestión del tamaño de los equipos en función de la carga cognitiva (Sweller, 2010) que cada equipo puede manejar, definiendo que cada servicio debe ser propiedad absoluta de un equipo con la capacidad cognitiva suficiente para construirlo y operarlo.

- **Del modelo de proyecto al modelo de producto**

Una consecuencia de nuestro modelo de proceso es el abandono del modelo de gestión por proyectos, tan usual en varias ingenierías, por el modelo de gestión de productos (PMBOK, 2021).

Hace 20 años contraponíamos los métodos ágiles a los que denominábamos "métodos basados en planes". La razón para esto no era abjurar de la planificación: de hecho, la planificación puede ayudarnos en la gestión si la usamos como guía. Tampoco se descartaba de plano la necesidad de definir alguna forma de cronograma de corto plazo ni de considerar cotas presupuestarias. Lo que negábamos era la validez de modelos en los cuales los planes rígidos habían sido la norma y en los cuales el éxito venía dado por el cumplimiento del plan, mientras los cambios eran problemas que resistíamos mediante costosos procesos de gestión del cambio, que nos permitieran volver lo más pronto posible a nuestro plan ideal.

Ahora, al menos para una gran parte de los desarrollos de productos de software, estamos empezando a dejar de lado la propia noción de proyecto, con su carga de planificación de tiempos, costos, fechas de finalización, etc. Ello ocurre porque la noción de proyecto viene asociada a un emprendimiento temporario, con un comienzo y un final, mientras que, en una economía en la que el software está en el centro del negocio, pensar que el producto de software se termine es como pensar que esa línea de negocio se termine, cuando lo deseable es que siga viviendo, evolucionando y extendiéndose.

Por eso es que estructurar el desarrollo de software alrededor de la idea de proyecto ya no tiene sentido en muchos casos. Han habido algunas ideas de reformular la definición de proyecto para ajustarla al desarrollo de software, que necesariamente requiere mayor flexibilidad en el control de ciertas variables, acepte correcciones constantes en base al aprendizaje y las necesidades de cambio (Rothman, 2007). Pero ni siquiera esos enfoques logran desterrar la idea de finalización.

Lo que se necesita es un modelo que acepte que el software está en el centro del modelo de negocio, y que por lo tanto es una actividad continua y no algo que debe terminar mientras tengamos resultados de negocio positivos. De hecho, hemos planteado que en el marco de Lean-Continuo el alcance se va ajustando en base a consideraciones de valor para el negocio.

Otra característica del modelo de proyectos que queremos evitar es la tendencia a medir productos de software o artefactos intermedios. La idea de Lean-Continuo es medir resultados y valor para el negocio. Por eso, más que controlar costos o cronogramas, nos interesa monitorear los beneficios del negocio y la velocidad con la que respondemos a los requerimientos en pos de esos beneficios. No buscamos cumplir con costos de proyecto, sino financiar cadenas de valor que se ajustan en base a resultados de negocio.

En última instancia, si algo se quiere fijar en el enfoque de productos, podemos manejar una cota presupuestaria definiendo el tamaño de equipo. Incluso se puede hacer una planificación de entregas, mientras no nos vayamos tan lejos en el tiempo.

De todas maneras, queda una situación en la cual es importante seguir usando una lógica de proyectos: el trabajo necesario para llegar a un *MVP* (acrónimo en inglés de producto mínimo viable: un producto que se construye con la menor cantidad de características posibles para hacer una prueba de concepto), para el cual se tiene al menos un alcance aproximado, una idea de costos y de tiempos. Pero una vez superada esa etapa y decidido que se va a seguir construyendo un producto, solemos pasar a una lógica de gestión del producto.

- **Un modelo con coherencia interna**

El modelo presentado hasta aquí se basa en un conjunto de prácticas que atañen a diferentes disciplinas y actividades del desarrollo de software. Sin embargo, hay que hacer notar que estas prácticas están fuertemente entrelazadas entre las distintas disciplinas.

Empecemos por los ejemplos de uso del sistema. Los ejemplos de uso surgen como un subproducto de la aplicación de SBE, pero no sólo sirven como especificaciones, sino que los mismos se pueden automatizar, quedando como especificaciones ejecutables que luego sirvan en la línea de producción de entrega continua como pruebas de aceptación automatizadas. Además, son herramientas de control de avance, pues definen criterios de completitud de funcionalidades o características que el sistema deba tener. Incluso se convierten en documentación viva del sistema.

Pero los ejemplos de uso no quedan allí: están escritos con un lenguaje ubicuo que luego se utilizará para el desarrollo del producto. Ese lenguaje ubicuo, a su vez, está limitado por contextos en los cuales se aplica, y que pueden servir para encontrar los límites a los microservicios, que a su vez pueden convertirse en límites entre equipos de trabajo. El mismo lenguaje ubicuo sirve en el contexto de los equipos de trabajo para referirse a conceptos, actividades, etc., del sistema. Y aparece también relacionado con una técnica de diseño que lleva casi dos décadas de existencia: *Domain Driven Design* (*DDD*) (Evans y Evans, 2004). Esto resulta interesante en nuestro contexto, ya que DDD también está relacionado con SBE por el uso del lenguaje ubicuo y además propugna que el diseño arquitectónico se base en los contextos del lenguaje, los mismos que numerosos autores proponen para limitar los microservicios (Fowler y Lewis, 2014).

La propia entrega continua se basa en una línea de producción automatizada, en la cual es fundamental contar con partes del sistema que sean desarrolladas y desplegadas por equipos plurifuncionales, que hablan el lenguaje ubicuo tanto hacia dentro del equipo como hacia clientes, usuarios y otros interesados. Esas partes deben ser lo más independientes posible para permitir trabajar con escasas dependencias con otros equipos. Por eso surgen aquí nuevamente los microservicios, que a su vez están limitados en contextos determinados por el lenguaje ubicuo.

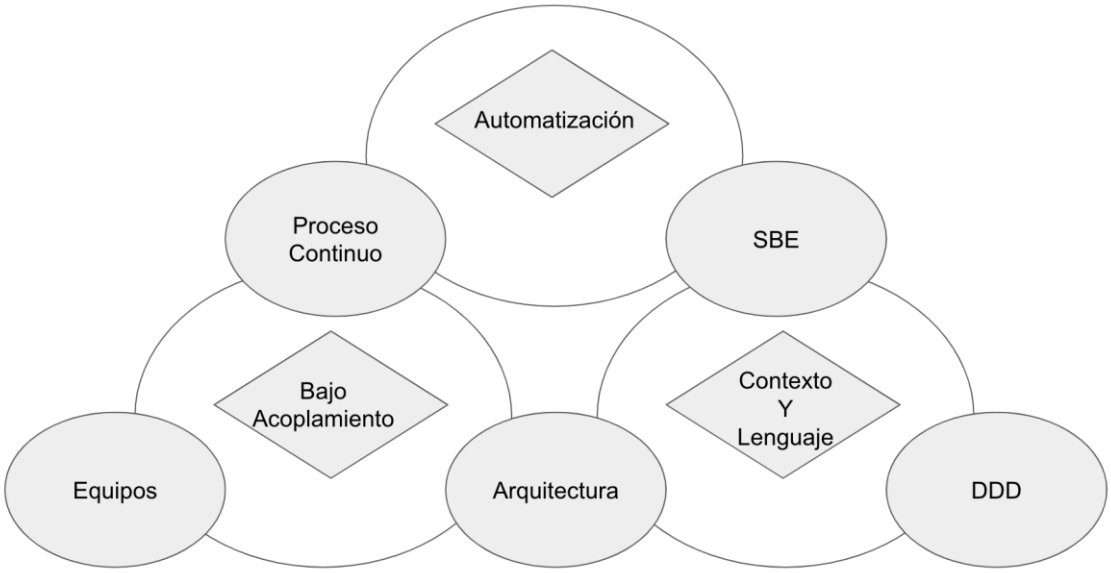
Los equipos que desarrollan, despliegan y operan el sistema, siguen la lógica de DevOps (o DevSecOps si se quiere usar el neo-neologismo), en el sentido de que son plurifuncionales, abarcando programadores y testers (desarrolladores en la jerga de los primeros tiempos de Scrum), más gente de operaciones, de seguridad, de plataforma, etc. Asimismo, el bajo acoplamiento de la arquitectura se ve facilitada por el bajo acoplamiento entre equipos.

La automatización cruza también varias disciplinas. Si bien SBE no requiere que se automaticen los ejemplos, es una ayuda para hacerlo. CD requiere automatizaciones varias en la línea de producción. Los microservicios son un buen soporte a la automatización, al igual que los equipos autogestionados, ya que hacen que cada tarea se pueda automatizar sin molestar a los demás servicios y equipos. Las pruebas automatizadas incluso favorecen el trabajo de los equipos en forma independiente, ya que brindan una red de contención para problemas derivados de cambios.

La propia idea de monitoreo y experimentación en producción es factible por el relativo bajo acoplamiento entre partes del sistema. En efecto, si cada parte está implementada como un microservicio y cada microservicio es responsabilidad de un equipo, esto facilita la experimentación - y la vuelta atrás cuando es necesaria - con el sistema en producción.

La Fig. 5 muestra esta interrelación entre prácticas dentro del modelo.

Fig. 5 Interrelación de prácticas y disciplinas (fuente: elaboración propia)



Lo que acabamos de mostrar tiene un corolario bastante evidente: conviene usar todas estas prácticas de manera conjunta. Por eso, si bien se pueden incorporar las prácticas de a una, y tal vez sea aconsejable si se está migrando desde otro modelo de proceso, hay que tender a usar todas juntas apenas sea posible, ya que el conjunto, con su coherencia interna, va a facilitar la implementación del modelo.

● EVIDENCIA EMPÍRICA: EL MODELO FUNCIONA

Hasta ahora hemos descrito un modelo. Sin embargo, no hemos probado que el mismo cubra las necesidades planteadas más arriba, de favorecer la respuesta a las demandas crecientes de cambios rápidos y seguros, permitir entregar los cambios con mayor frecuencia y facilitar el feedback proveniente del uso de los productos.

Existe un trabajo muy interesante de investigación, realizado por Nicole Forsgren, Jez Humble y Gene Kim y difundido mediante un exitoso libro (Forsgren, 2018) que nos permite analizar esta cuestión.

El trabajo de investigación es de suma importancia porque, por un lado, fue desarrollado por practicantes de la industria, con la credibilidad adicional de basarse en métodos académicos rigurosos. A la vez, mientras los autores por un lado realizan publicaciones en revistas académicas con revisión de pares (Wiedemann, 2019; Forsgren y Kersten, 2018), lo ponen a disposición de la industria en general a través de libros (Kim at al, 2016), charlas en congresos empresariales (como el DevOps Enterprise Forum) y redes sociales.

El trabajo de investigación se desarrolló a lo largo de cuatro años, entre 2013 y 2017. Recolectaron información a través de encuestas en organizaciones de todos los tamaños, desde startups hasta grandes corporaciones, en todas las industrias, en todo el mundo. Trabajaron sobre organizaciones muy innovadoras, y también sobre otras que operan en entornos muy regulados, como la banca, el gobierno o la salud. Los sistemas que desarrollan las organizaciones han sido, desde sistemas legados hasta los desarrollados con las últimas tecnologías, en sistemas de sólo software hasta sistemas de software embebido y sistemas ciberfísicos complejos. Recolectaron 23.000 respuestas provenientes de 2.000 organizaciones distintas. A lo largo de los cuatro años trabajaron en distintos aspectos de un proceso que ellos llaman DevOps (y que se parece mucho a nuestro modelo Lean-Continuo), basado en 24 capacidades y prácticas que impulsan el desempeño de la entrega de software (esas mismas 24 capacidades fueron surgiendo como resultados de la investigación).

Es de esperar que pronto haya más grupos de investigación que puedan validar los resultados a los que llegaron usando enfoques diferentes o replicando este.

Los principales hallazgos a los que han llegado son:

- El desarrollo y la entrega de software se pueden medir de una manera estadísticamente significativa. Lograron definir métricas para medir el desempeño de la entrega de software, que son las que hemos mencionado más arriba.
- Las organizaciones de alto desempeño mantienen el nivel de desempeño en forma consistente y significativamente mejor que la mayor parte del resto de las organizaciones. Además, la brecha entre las organizaciones de alto y bajo desempeño se agranda cada vez más.
- Hay una correlación positiva, probablemente no intuitiva para muchas escuelas de gestión, entre la mejora del desempeño y el logro de altos niveles de calidad y estabilidad. Esto es, la velocidad, la estabilidad y la calidad se realimentan positivamente.
- Las prácticas técnicas, incluida la automatización, impactan el desempeño de la entrega de software.
- Determinaron qué prácticas arquitectónicas impulsan mejoras en el desempeño de la entrega de software.
- Las prácticas de Lean Management también impactan positivamente en el desempeño de la entrega de software.
- Integrar la seguridad en el desarrollo y la entrega de software ayudan al proceso y mejoran la velocidad de entrega.
- La cantidad de retrabajo disminuye al utilizar las 24 capacidades y prácticas.

- Tener buenas prácticas técnicas contribuye a una mayor lealtad hacia la organización.
- Observaron una realimentación positiva entre las variables organizacionales de desempeño, identidad y cultura.
- El modelo logra entregar producto más rápido, con menores niveles de agotamiento de las personas (*burnout*) y mayor satisfacción con el trabajo.
- Exploraron el rol de los líderes para medir el impacto transformador en las organizaciones.
- Se han visto efectos positivos en los resultados no comerciales, tales como efectividad, eficiencia y satisfacción de los clientes.

● UN ESCENARIO POCO EXPLORADO: SISTEMAS CIBERFÍSICOS

Llamamos *sistemas ciberfísicos* a aquellos sistemas complejos en los cuales los componentes son dispositivos controlados y monitoreados por el software. En estos sistemas, que cada vez descansan más en el software para proveer sus funcionalidades, el foco del desarrollo está pasando del de los componentes mecánicos y electrónicos al del desarrollo de software. Por lo tanto, en esta época no podemos analizar un enfoque metodológico para el software dejando de lado estos sistemas complejos e interdisciplinarios.

Pero al plantearnos si nuestro enfoque es adecuado al desarrollo de sistemas ciberfísicos nos encontramos con unos cuantos desafíos. Destacan los siguientes (Kreutz, 2021):

- Dependencia del hardware. Agregar nuevas características necesita adaptaciones a la plataforma de hardware. Por lo tanto, no puede analizarse todo el desarrollo como cambios incrementales a un producto no terminado cuando el hardware se actualiza en ciclos de varios meses o años.
- Complejidad del entorno de operación. Pretender que los entornos de prueba sean similares a los de producción resulta poco realista, y hasta imposible en ocasiones, dado que el entorno de producción de los sistemas ciberfísicos es el mundo real, con sus comportamientos imprevistos, e incluso imprevisibles en ocasiones.
- Comportamiento no determinista de los sensores y actuadores debido al ruido. Esto hace que a lo sumo se puedan ejecutar algunas pruebas que aseguren la confiabilidad con cierto grado de significancia estadística.
- Conectividad a los sistemas ya instalados para el monitoreo en producción. Esto se ve limitado en muchas industrias que quieren evitar los riesgos de tener sus dispositivos continuamente conectados a Internet.
- La cultura de responsabilidad conjunta por la entrega de valor suele estar en riesgo por la participación de ingenieros de distintas disciplinas, cada una con sus particularidades culturales.
- Problemas con los clientes. Los clientes corporativos suelen esperar conocer el presupuesto del sistema de antemano y no quedar atados a contratos de mantenimiento de por vida. Más aún en el caso de los consumidores finales, que compran el sistema físico sin esperar que deban pagar luego por las actualizaciones de software.
- Regulaciones de seguridad. Como en muchos casos los sistemas ciberfísicos operan en industrias altamente reguladas por cuestiones de seguridad, incluso seguridad de la vida humana, los productos suelen requerir certificaciones que resulta complicadas

de obtener en medio de un gran número de pequeñas actualizaciones incrementales. Hay algunas propuestas de introducir de manera incremental las certificaciones de seguridad, aunque esto es algo que está lejos de ser globalmente aceptado (Fassbinder, 2018 y Abrahamsson, 2020).

- Pruebas en producción. Muchas de las pruebas en producción, o técnicas como Blue-green deployment o Canary Release pueden ser impracticables en sistemas cuyo mal funcionamiento ponga en riesgo la vida.

Con todos estos desafíos en mente, lo cierto es que el enfoque Lean-Continuo no puede adoptarse sin adaptaciones al desarrollo de sistemas ciberfísicos. Y ello, al menos en parte, provoca que los enfoques tradicionales sigan vivos.

Hay algunas adaptaciones obvias, como separar los ciclos de vida del hardware y del software, introducir capas de abstracción del hardware, simular partes del entorno que lleven menos esfuerzo.

Hay también algunas primeras experiencias, sobre todo en la Unión Europea (Gartziandia et al., 2021; Combemale y Wimmer, 2019 y Eramo et al, 2021). En general se basan en microservicios específicos, desplegados en la nube, que hacen tareas de monitoreo, para entender el comportamiento del sistema en tiempo de ejecución, para facilitar recuperación ante fallas y detección de incertezas. Gran parte de estos desarrollos están basados en *MDE (Model Driven Engineering)*, un enfoque que en la Ingeniería de Software pura había quedado de lado por impopular, pero que resulta de ayuda para modelar sistemas físicos, brindando una abstracción mayor que permite mejorar el trabajo conjunto de ingenieros de diferentes disciplinas.

CONCLUSIONES

En este artículo hemos explicado por qué el proceso de desarrollo de software es tan cambiante como el software mismo y la tecnología de hardware sobre la que se ejecuta. También hemos mostrado el que la tendencia indica que va a ser el modelo dominante en esta década, que hemos llamado Lean-Continuo, con sus características, su impacto en distintas disciplinas de la Ingeniería de Software y algo de evidencia que respalda su idoneidad. Como ha venido pasando a lo largo de la historia de los distintos modelos, éste también exhibe una coherencia interna que hace que las distintas prácticas se realimenten entre sí.

Una costumbre que suele aparecer cada vez que emerge un nuevo modelo es desdeñarlo diciendo que es una moda pasajera. Esta reacción es lógica para quienes han resuelto problemas complejos con modelos previos y no ven la necesidad de un cambio, pero no debería serlo para aquellos que día a día experimentan las limitaciones de los modelos actuales. De todas maneras, la mejor refutación que puede hacerse al concepto de moda es la evidencia empírica de que el nuevo modelo funciona mejor que los anteriores. Por supuesto que nunca los cambios son totales, sino que la industria se va a adecuando de manera incremental, pero la adaptación progresiva es el camino.

Otras tentaciones surgen del lado de los innovadores. Están aquellos que sostienen que un modelo se debe aplicar como un todo sin fisuras o no tendrá éxito. Están también los que

creen que cada nuevo modelo es definitivo y perdurará en el tiempo durante décadas. Ambas son posiciones extremas, en las que debemos evitar caer. Muchas veces es imposible para una organización cambiar de modo inmediato y radical sus métodos de desarrollo, por lo que es necesario experimentar distintas maneras de transicionar. Por otro lado, si siempre hemos visto cambios de modelos, ¿por qué pensar que el que vivimos es el definitivo?

Nos parece importante también destacar que el modelo, tomado como un todo, todavía no ha demostrado su factibilidad en todos los escenarios de desarrollo. Ya vimos que los sistemas ciberfísicos plantean desafíos que aún no han sido resueltos. También hemos dicho que arquitecturas distribuidas, como la de microservicios, son demasiado costosas en pequeñas organizaciones, más allá de que en la teoría son las más adecuadas al modelo. La noción de que la gestión por proyectos va quedando atrás ni siquiera sirve en muchos desarrollos, e incluso no se puede desterrar de las primeras fases de la construcción de productos. La tercerización del desarrollo se seguirá aplicando en algunas áreas de las empresas que no estén en el centro del negocio, y en esos casos es dudoso que el modelo se pueda aplicar de punta a punta.

Tampoco todas las prácticas se utilizan con la misma frecuencia en la industria. SBE, la programación en parejas y algunas automatizaciones, han tenido menor adopción que otras, aun cuando existe sólida evidencia de sus beneficios.

Por todo lo anterior, consideramos a este nuevo modelo como un producto en construcción, que se irá adoptando gradualmente en la medida en que se vaya demostrando su proporcionalidad, y que podrá cambiar de modo incremental, llegando incluso el día en que debamos reemplazarlo por otro. No por nada la sociedad nos considera la más dinámica de las profesiones.

REFERENCIAS BIBLIOGRÁFICAS

Abrahamsson, P., et al. (2020). Towards a secure devops approach for cyber-physical systems: An industrial perspective. *International Journal of Systems and Software Security and Protection* (IJSSSP) 11(02), 38-57.

Adzic, G. (2011). *Specification By Example. How successful teams deliver the right software*. Manning Publications.

Adzic, G. (2009). *Bridging the communication gap: specification by example and agile acceptance testing.* Neuri Limited.

Andreessen, M. (2011). Why software is eating the world. *Wall Street Journal* 20.2011: C2.

Bass, L., Weber, I., & Zhu, L. (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.

Beck, K. (2011). The History of JUnit and the Future of Testing with Kent Beck. *Software Testing Magazine*, April 1, 2011. https://www.softwaretestingmagazine.com/knowledge/the-history-of-junit-and-the-future-of-testing-with-kent-beck/ visitado en noviembre de 2022.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beck, K. (2001). *Agile Manifesto. https://agilemanifesto.org/iso/es/manifesto.html*

Beck, K. (2000). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Beck, K. & Fowler, M. (2001). *Planning extreme programming*. Addison-Wesley Professional.

Booch, G. (1990). *Object oriented design with applications*. Benjamin-Cummings Publishing Co., Inc.

Brooks, F. P., & Bullet, N. S. (1987). Essence and accidents of software engineering. *IEEE computer*, 20(4), 10-19.

Charette, R.N. (2021). *How Software is Eating the Car*. https://spectrum.ieee.org/software-eating-car

Cockburn, A. (2005). *The Pattern: Ports and Adapters*. https://alistair.cockburn.us/hexagonal-architecture/

Cockburn, A., & Williams, L. (2000). The costs and benefits of pair programming. Extreme programming examined, 8, 223-247.

Cohn, M. (2010). *Succeeding with agile: software development using Scrum*. Pearson Education.

Combemale, B., Wimmer, M. (2019). Towards a model-based devops for cyber-physical systems. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer, Cham.

Conway, M.E. (1968) How do committees invent. *Datamation*, 14(4), 28-31.

Cox, J. et al. (2019). Full Stack Teams, Not Engineers. *IT Revolution*.

DeMarco, T., & Lister T. (2013). *Peopleware: productive projects and teams.* Addison-Wesley.

Dyck, A., Penners, R., & Lichter, H. (2015). Towards definitions for release engineering and DevOps. *In IEEE/ACM 3rd International Workshop on Release Engineering*. IEEE.

Eramo, R., et al. (2021). AIDOaRt: AI-augmented Automation for DevOps, a Model-based Framework for Continuous Development in Cyber-Physical Systems. In *24th Euromicro Conference on Digital System Design (DSD).* IEEE.

Evans, E., & Evans, E.J. (2004). *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional.

Farley, D. (2022). *Modern Software Engineering*. Addison-Wesley.

Fassbinder P. (2018). *Continuous Quality - Fulfilling Industry Regulations and Quality Expectations in a World of Continuous Delivery*. Sofware Quality Days.

Fontela, C. (2012). *Estado del arte y tendencias en Test-Driven Development*. Universidad Nacional de La Plata. http://163.10.34.134/bitstream/handle/10915/4216/Documento_completo.pdf?sequence=1.

Forsgren, N., Humble, J., Kim, G. (2018). Accelerate: the science of lean software and DevOps: building and scaling high performing technology organizations. *IT Revolution.*

Forsgren, N., Kersten, M. (2018). DevOps metrics. *Communications of the ACM,* 61(4), 44-48.

Fowler, M. (2003). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Fowler, M. (2015). *Monolith First.* ThoughtWorks. https://martinfowler.com/bliki/MonolithFirst.html visitado en noviembre de 2022.

Fowler, M. (2010). *Blue-Green Deployment*. https://martinfowler.com/bliki/BlueGreenDeployment.html visitado en noviembre de 2022

Fowler, M., & Lewis, J. (2014). *Microservices. ThoughtWorks*. http://martinfowler.com/articles/microservices.html

Freeman, S., Pryce, N. (2009). *Growing object-oriented software, guided by tests*. Pearson Education.

Gane, S. (1977). *Structured Systems Analysis/Chris Gane, Trish Sarson: Structured Systems Analysis: tools & techniques*.

Gärtner, M. (2012). *ATDD by example: a practical guide to acceptance test-driven development.* Addison-Wesley.

Gartziandia, A., et al. (2021). Microservices for Continuous Deployment, Monitoring and Validation in Cyber-Physical Systems: an Industrial Case Study for Elevators Systems. In *IEEE 18th International Conference on Software Architecture Companion* (ICSA-C) (pp. 46-53). IEEE.

Gray, D. (2013). A Business within the Business. *NHRD Network Journal*, 6(2), 51-57.

Hansson, D.H. (2016). *The Majestic Monolith*. https://medium.com/signal-v-noise/the-majestic-monolith-29166d022228.

Henney, K. (2021). *"Agility ≠ Speed"*. https://kevlinhenney.medium.com/agility-speed-96057078fe40

Holmstrom-Olsson, H., Alahyari, H., Bosch, J. (2012). *Climbing the Stairway to Heaven -- A Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software*. Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications. IEEE Computer Society: 392–399.

Humble, J., Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

IEEE Standard Glossary of Software Engineering Terminology (1990). *IEEE std* 610.12-1990.

Jackson, C. (2019). *"Micro Frontends"*. https://martinfowler.com/articles/micro-frontends.html

Jones, N., et al. (2018). *The InfoQ eMag: Chaos Engineering*. Web: https://www.infoq.com/minibooks/emag-chaos-engineering/

Kelly, A. (2017). *Continuous Digital: An Agile Alternative to Projects*.

Kerzazi, N., Adams, B. (2016). *Who needs release and devops engineers, and why? Proceedings of the International Workshop on Continuous Software Evolution and Delivery*.

Kim, G., et al. (2016). *The DevOps Handbook: How to Create World-Class Agility*. Reliability, and Security in Technology Organizations.

Kniberg, H. (2011). *Lean from the trenches: Managing large-scale projects with Kanban*. Pragmatic Bookshelf.

Koskela, L. (2007). *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Simon and Schuster.

Kreutz, A., et al. (2021). *DevOps for Developing Cyber-Physical Systems*. Fraunhofer Institute for Cognitive Systems IKS. Magazino GmbH.

Lehman, M. M. (1978). Programs, cities, students—limits to growth?. *Programming Methodology* (pp. 42-69). Springer.

Lehman, M.M. (1996). *Laws of software evolution revisited*. European Workshop on Software Process Technology. Springer.

Loucopoulos, P., Karakostas, V. (1995). *System requirements engineering.*

Malan, R. (2008). *Conway´s Law*. https://eavoices.com/2008/02/13/conways-law/

Mugridge, R. (2008). *Managing agile project requirements with storytest-driven development*. IEEE software (25)1: 68-75.

Myrbakken, H., Colomo-Palacios, R. (2017). DevSecOps: a multivocal literature review. In *International Conference on Software Process Improvement and Capability Determination.* Springer, Cham.

Narayan, S. (2018). *Products Over Projects.* https://martinfowler.com/articles/products-over-projects.html.

Naur, P. (1968). *Software engineering-report on a conference sponsored by the NATO Science Committee Garmisch, Germany*. http://homepages. cs. ncl. ac. uk/brian. randell/NATO/nato1968.

Newman, S. (2021). *Building Microservices.* O'Reilly Media, Inc.

North, D. (2006). *Introducing BDD*. Better Software 12.

Park, S.S. (2011). *Communicating domain knowledge through example-driven story testing*. University of Calgary, Department of Computer Science.

Parsons, R. (2015). *Inverse conway maneuver*. https://www.thoughtworks.com/radar/techniques/inverseconway-maneuver,

Pettichord, B., Marick, B. (2022). Agile acceptance testing. In *Conference on Extreme Programming and Agile Methods.* Springer.

PMBOK® Guide 7th Edition (2021). https://www.pmi.org/pmbok-guide-standards/foundational/pmbok

Poppendieck, M., Poppendieck, T. (2003*). Lean software development: an agile toolkit*. Addison-Wesley.

Pugh, K. (2010). *Lean-Agile Acceptance Test-Driven Development.* Pearson Education.

Reis, E. (2011). The lean startup. *Crown Business,* 27: 2016-2020.

Richards, M. (2015). *Microservices vs. service-oriented architecture*. O'Reilly.

Rosenthal, C., Jones, N. (2020). *Chaos engineering: system resiliency in practice*. O'Reilly Media.

Rothman, J. (2007). *Manage it!: your guide to modern, pragmatic project management.* Pragmatic Bookshelf.

Royce, W.W. (1987). *Managing the development of large software systems: concepts and techniques*. Proceedings of the 9th international conference on Software Engineering.

Sato, C. (2014). *Canary Release*. https://martinfowler.com/bliki/CanaryRelease.html

Schwaber, K., & Sutherland, J. (2021). *La Guía Scrum.* https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Spanish-European.pdf

Schwaber, K., Beedle, M. (2002). *Agile software development with Scrum*. Vol. 1. Upper Saddle River: Prentice Hall.

Skelton, M., Pais, M. (2019). Team topologies: organizing business and technology teams for fast flow. *It Revolution*.

Stevens, W.P., Myers, G.J., & Constantine L.L. (1974). *Structured design. IBM Systems Journal*, 13(2), 115-139.

Sweller, J. (2010). *Cognitive load theory: Recent theoretical advances.*

Walls, M. (2013). *Building a DevOps culture*. O'Reilly Media, Inc.

Westrum, R. (2004). A typology of organisational cultures. *BMJ Quality & Safety,* 13.suppl 2: ii22-ii27.

Wiedemann, A., et al. Research for practice: the DevOps phenomenon. *Communications of the ACM,* 62(8), 44-49.

Williams, L., Kessler, R. R., Cunningham, W., & Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE software*, 17(4), 19-25.

Yourdon, E. (1989). *Modern structured analysis*. Yourdon press.

Zuill, W., & Meadows, K. (2016). Mob programming: A whole team approach. In *Agile 2014 Conference, Orlando, Florida* (Vol. 3).

Fecha de recepción: 4/11/2022

Fecha de aprobación: 5/12/2022